

Richard S. Wright, Jr. and Benjamin Lipchak

SAMS



CD-ROM included



OpenGL[®]

SUPERBIBLE

THIRD EDITION

OpenGL® SuperBible, Third Edition

By Richard S. Wright Jr., Benjamin Lipchak

Publisher: Sams Publishing

Pub Date: June 30, 2004

ISBN: 0-672-32601-9

Pages: 1200

OpenGL SuperBible, Third Edition is a comprehensive, hands-on guide for Mac and Windows programmers who need to know how to program with the new version of OpenGL. This book will help readers master and expand their knowledge of 3D graphics programming and OpenGL implementation. Seasoned OpenGL programmers will also find this learning tool serves as a reference that can be used time and again. This book explains how to draw lines, points, and polygons in space; move around in a virtual world; utilize raster graphics and image processing in OpenGL. In addition readers learn how to use fog and blending special effects; explore the use of OpenGL extensions; use shaders to create procedural textures; write vertex programs for dynamic special effects.

Copyright

Copyright © 2005 by Sams Publishing

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

Library of Congress Catalog Card Number: 2003116629

Printed in the United States of America

First Printing: July 2004

07 06 05 04 4 3 2 1

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the CD or programs accompanying it.

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside the U.S., please contact

International Sales
1-317-428-3341
international@pearsontechgroup.com

Credits

Associate Publisher

Michael Stephens

Acquisitions Editor

Loretta Yates

Development Editor

Sean Dixon

Managing Editor

Charlotte Clapp

Project Editor

Dan Knott

Copy Editor

Chuck Hutchinson

Indexer

Erika Millen

Proofreader

Suzanne Thomas

Technical Editor

Nick Haemel

Publishing Coordinator

Cindy Teeters

Multimedia Developer

Dan Scherf

Book Designer

Gary Adair

Page Layout

Michelle Mitchell

Dedication

Dedicated to the memory of Richard S. Wright, Sr.

I Thessalonians 4:16

Thanks, Dad, for just letting me be a nerd.

—Richard S. Wright, Jr.

To my parents, Dorothy and Mike, for providing me top-quality genetic material and for letting me bang away on the TRS-80 keyboard when I should have been outside banging sticks against trees. Those crude 128x48 monochrome images have come a long way. Mom and Dad, thanks for seeing my future in those blocky pixels.

—Benjamin Nason Lipchak

About the Authors

Richard S. Wright, Jr., has been using OpenGL for nearly 10 years, since it first became available on the Windows platform, and teaches OpenGL programming in the game design degree program at Full Sail in Orlando, Florida. Currently, Richard is the president of Starstone Software Systems, Inc., where he develops multimedia simulation software for the PC and Macintosh platforms using OpenGL.

Previously with Real 3D/Lockheed Martin, Richard was a regular OpenGL ARB attendee and contributed to the OpenGL 1.2 specification and conformance tests. Since then, Richard has worked in multidimensional database visualization, game development, medical diagnostic visualization, and astronomical space simulation.

Richard first learned to program in the eighth grade in 1978 on a paper terminal. At age 16, his parents let him buy a computer instead of a car, and he sold his first computer program less than a year later (and it was a graphics program!). When he graduated from high school, his first job was teaching programming and computer literacy for a local consumer education company. He studied electrical engineering and computer science at the University of Louisville's Speed Scientific School and made it halfway through his senior year before his career got the best of him and took him to Florida. A native of Louisville, Kentucky, he now lives with his wife and three children between Orlando and Daytona Beach. When not programming or dodging hurricanes, Richard is an avid amateur astronomer and a Sunday School teacher.

Benjamin Lipchak graduated from Worcester Polytechnic Institute with a double major in technical writing and computer science. "Why would anyone with a CS degree want to become a writer?" That was the question asked of him one fateful morning when Benj was interviewing for a tech writing job at Digital Equipment Corporation. Benj's interview took longer than scheduled, and he left that day with job offer in hand to work on the software team responsible for DEC's Alpha Workstation OpenGL drivers.

After Compaq bought DEC and laid off most of the graphics group, Benj quit one Friday and came

back the following Monday on a more lucrative contracting basis. Despite his new status, Benj continued his leadership role at Compaq, traveling to customer sites and managing projects related to the PowerStorm UNIX and NT OpenGL drivers. Workstation apps were the primary focus, but GLQuake ran on AlphaStations at speeds never before witnessed.

Benj took a two-year hiatus and founded an image search start-up company during the Internet boom. The Internet bust had Benj begging on hands and knees for his first love, OpenGL, to take him back. And she did, in the form of ATI Research in Marlboro, Massachusetts. Recently, he has been most active in the area of fragment shaders, adding support to the Radeon OpenGL drivers. Benj chaired the OpenGL ARB working group that generated the GL_ARB_fragment_program extension spec, and has participated in ATI's OpenGL Shading Language efforts. In his fleeting spare time, Benj tries to get outdoors for some hiking or kayaking. He also operates an independent record label, Wachusett Records, specializing in solo piano music.

Acknowledgments

Where do I begin? I thank God for every door of opportunity He has opened for me throughout my life, and for the strength to make it through when I wasn't smart enough to say no. I thank my wife and family for putting up with me during the most insane periods of my life while trying to get this book out yet the third time. LeeAnne, Sara, Stephen, and Alex all were put on hold, promises were broken, weekends lost, voices raised. Daddy went insane for a little while, and when it was all over, they were still there waiting for me. This book should have their names on it, not mine.

The people at Sams have been terrific. I want to thank Loretta Yates for putting up with my missed deadlines, and congratulate her for the birth of her "real" baby just as we were finishing the book. Sean Dixon, the content editor; Nick Haemel, the technical reviewer; Chuck Hutchinson, my copy editor; and Dan Knott, the project editor, all worked hard to make me look like the literary and technical genius that I'm really not.

Special thanks also go to Benjamin Lipchak and ATI Technologies, Inc. Benjamin joined late as a coauthor, and rescued some of the most important additions to the book. Thank you, ATI, for donating some of Benj's work time, the use of sample code, and for early access to OpenGL drivers. Thank you also for your continued commitment to the OpenGL standard.

Finally, many thanks also to Full Sail for their support over the past few years by allowing me the privilege of teaching OpenGL on a part-time basis. I come in for a few hours, I get to talk about what I really love to talk about, and the audience has to act as though they enjoy it and pay attention. I even get paid. How on earth I keep getting away with this is beyond me! Many people at Full Sail have helped me especially, either on the book directly or just by offering general support and making life easier for me. Thanks to Rob Catto for your understanding and support on many issues. Tony Whitaker took several student projects and cleaned them up for the demo directory and helped make sure all our code ran on Linux. Thanks to Bill Galbreath, Richard Levins, and Stephen Carter for demo materials. Finally, thanks to Troy Humphreys and Brad Leffler for being the two best lab specialists I could ask for and filling in with next-to-no notice when I had some emergency or another to take care of.

—Richard S. Wright, Jr.

I'd like to begin by acknowledging my colleagues at ATI who have unselfishly given their valuable time and advice reviewing my chapters, and leaving me to take all the credit. These folks are masters of the OpenGL universe, and I am fortunate to work side by side with (or sometimes hundreds of miles away from) this caliber of individual on a daily basis. In particular, I'd like to thank Dan Ginsburg, Rick Hammerstone, Evan Hart, Bill Licea-Kane, Glenn Ortner, and Jeremy Sandmel. Thanks to technical editor Nick Haemel, also from ATI, for the unique perspective of someone previously unexposed to my chapters' subject matter. What better way to test the material for our readers? Most of all, thanks to manager and friend Kerry Wilkinson, and ATI, for dedicating the time and equipment for me to work on this book. I hope the end product proves to be mutually beneficial.

Richard, thanks for the opportunity to work with you on this project. I'm honored to have my

name listed beside your own, and welcome any and all future "Wright & Lipchak" collaborations.

Thanks to the team of editors and other support staff at Sams Publishing for transforming my lowly text into something I'm proud of. Your eagle eyes spared me from sweating the details, making writing hundreds of pages much less strenuous.

Thanks to WPI professors Mike Gennert, Karen Lemone, John Trimbur, Susan Vick, Matt Ward, Norm Wittels, and others for the solid foundation I lean on every day. A shout out to all my friends at GweepNet for distracting me with PC game LAN parties when I was burnt out from too much writing. Special thanks to Ryan Betts for coining the title of Chapter 21. To my entire extended family, including Beth, Tim, Alicia, Matt, and Jen, thanks for tolerating my surgically attached laptop during the winter months. To brother Paul, your success in everything you do provides me with nonstop healthy competition. To sister Maggie, you redefine success in my eyes every time I see you. You both make me proud to have you as siblings. Last but not least, I'd like to thank Jessica for keeping herself so busy that my cross to bear rarely seemed the heaviest. The day we have a moment to breathe will be the day we realize there's time for yet another side project. Maybe next time we'll work on one together!

—Benjamin Lipchak

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an associate publisher for Sams Publishing, I welcome your comments. You can email or write me directly to let me know what you did or didn't like about this book-as well as what we can do to make our books better.

Please note that I cannot help you with technical problems related to the topic of this book. We do have a User Services group, however, where I will forward specific technical questions related to the book.

When you write, please be sure to include this book's title and authors as well as your name, email address, and phone number. I will carefully review your comments and share them with the authors and editors who worked on the book.

Email: feedback@samspublishing.com

Mail: Michael Stephens
Associate Publisher
Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

For more information about this book or another Sams Publishing title, visit our Web site at www.samspublishing.com. Type the ISBN (excluding hyphens) or the title of a book in the Search field to find the page you're looking for.

Introduction

I have a confession to make. The first time I ever heard of OpenGL was at the 1992 Win32 Developers Conference in San Francisco. Windows NT 3.1 was in early beta (or late alpha), and many vendors were present, pledging their future support for this exciting new graphics technology. Among them was a company called Silicon Graphics, Inc. (SGI). The SGI representatives were showing off their graphics workstations and playing video demos of special effects from some popular movies. Their primary purpose in this booth, however, was to promote a new 3D graphics standard called OpenGL. It was based on SGI's proprietary IRIS GL and was fresh out of the box as a graphics standard. Significantly, Microsoft was pledging future support for OpenGL in Windows NT.

I had to wait until the beta release of NT 3.5 before I got my first personal taste of OpenGL. Those first OpenGL-based screensavers only scratched the surface of what was possible with this graphics API. Like many other people, I struggled through the Microsoft help files and bought a copy of the *OpenGL Programming Guide* (now called simply "The Red Book" by most). The Red Book was not a primer, however, and it assumed a lot of knowledge that I just didn't have.

Now for that confession I promised. How did I learn OpenGL? I learned it by writing a book about it. That's right, the first edition of the *OpenGL SuperBible* was me learning how to do 3D graphics myself...with a deadline! Somehow I pulled it off, and in 1996 the first edition of the book you are holding was born. Teaching myself OpenGL from scratch enabled me somehow to better explain the API to others in a manner that a lot of people seemed to like. The whole project was nearly canceled when Waite Group Press was acquired by another publisher halfway through the publishing process. Mitchell Waite stuck to his guns and insisted that OpenGL was going to be "the next big thing" in computer graphics. Vindication arrived when an emergency reprint was required because the first run of the book sold out before ever making it to the warehouse.

That was a long time ago, and in what seems like a galaxy far, far away...

Only three years later 3D accelerated graphics were a staple for even the most stripped-down PCs. The "API Wars," a political battle between Microsoft and SGI, had come and gone; OpenGL was firmly established in the PC world; and 3D hardware acceleration was as common as CD-ROMs and sound cards. I had even managed to turn my career more toward an OpenGL orientation and had the privilege of contributing in some small ways to the OpenGL specification for version 1.2 while working at Lockheed Martin/Real 3D. The second edition of this book, released at the end of 1999, was significantly expanded and corrected. We even made some modest initial attempts to ensure all the sample programs were more friendly in non-Windows platforms by using the GLUT framework.

Now, nearly five years later (eight since the first edition!), we bring you yet again another edition, the third, of this book. OpenGL is now without question the premier cross-platform real-time 3D graphics API. Excellent OpenGL stability and performance are available on even the most stripped-down bargain PC today. OpenGL is also the standard for UNIX and Linux operating systems, and Apple has made OpenGL a core fundamental technology for the new MacOS X operating system. OpenGL is even making inroads via a new specification, OpenGL ES, into embedded and mobile spaces. Who would have thought five years ago that we would see Quake running on a cell phone?

It is exciting that, today, even laptops have 3D acceleration, and OpenGL is truly everywhere and on every mainstream computing platform. Even more exciting, however, is the continuing evolution of computer graphics hardware. Today, most graphics hardware is programmable, and OpenGL even has its own shading language, which can produce stunningly realistic graphics that were undreamed of on commodity hardware back in the last century. (I just had to squeeze that in someplace!)

With this third edition, I am pleased that we have added Benjamin Lipchak as a coauthor. Benj is primarily responsible for the chapters that deal with OpenGL shader programs, and coming from the ARB groups responsible for this aspect of OpenGL, he is one of the most qualified authors on this topic in the world.

We have also fully left behind the "Microsoft Specific" characteristics of the first edition and have embraced a more multiplatform approach. All the programming examples in this book have been tested on Windows, MacOS X, and at least one version of Linux. There is even one chapter apiece on these operating systems, with information about using OpenGL with native applications.

What's in This Book

The *OpenGL SuperBible* is divided into three parts. In the first, we cover what we call *Classic OpenGL*. This is the fixed pipeline functionality that has been a characteristic of OpenGL from the beginning. Many of these chapters have been greatly expanded since the second edition because OpenGL has had a number of revisions since version 1.2. Fixed pipeline programming will still be with us for a long time to come, and its simple programming model will still make it the choice of many programmers for years.

In these chapters, you will learn the fundamentals of real-time 3D graphics programming with OpenGL. You'll learn how to construct a program that uses OpenGL, how to set up your 3D-rendering environment, and how to create basic objects and light and shade them. Then we'll delve deeper into using OpenGL and some of its advanced features and different special effects. These chapters are a good way to introduce yourself to 3D graphics programming with OpenGL and provide the conceptual foundation on which the more advanced capabilities later in the book are based.

In the second part, three chapters provide specific information about using OpenGL on the three mainstream operating system families: Windows, MacOS X, and Linux/UNIX.

Finally, the third part contains the newest features not just of OpenGL, but of 3D graphics hardware in general today. The OpenGL Shading Language, in particular, is the principal feature of OpenGL 2.0, and it represents the biggest advance in computer graphics in many years.

Part I: Classic OpenGL

Chapter 1—Introduction to 3D Graphics and OpenGL

This introductory chapter is for newcomers to 3D graphics. It introduces fundamental concepts and some common vocabulary.

Chapter 2—Using OpenGL

In this chapter, we provide you with a working knowledge of what OpenGL is, where it came from, and where it is going. You will write your first program using OpenGL, find out what headers and libraries you need to use, learn how to set up your environment, and discover how some common conventions can help you remember OpenGL function calls. We also introduce the OpenGL state machine and error-handling mechanism.

Chapter 3—Drawing in Space: Geometric Primitives and Buffers

Here, we present the building blocks of 3D graphics programming. You'll basically find out how to tell a computer to create a three-dimensional object with OpenGL. You'll also learn the basics of hidden surface removal and ways to use the stencil buffer.

Chapter 4—Geometric Transformations: The Pipeline

Now that you're creating three-dimensional shapes in a virtual world, how do you move them around? How do you move yourself around? These are the things you'll learn here.

Chapter 5—Color, Materials, and Lighting: The Basics

In this chapter, you'll take your three-dimensional "outlines" and give them color. You'll learn how

to apply material effects and lights to your graphics to make them look real.

Chapter 6—More on Colors and Materials

Now it's time to learn about blending objects with the background to make transparent (see-through) objects. You'll also learn some special effects with fog and the accumulation buffer.

Chapter 7—Imaging with OpenGL

This chapter is all about manipulating image data within OpenGL. This information includes reading a TGA file and displaying it in an OpenGL window. You'll also learn some powerful OpenGL image-processing capabilities.

Chapter 8—Texture Mapping: The Basics

Texture mapping is one of the most useful features of any 3D graphics toolkit. You'll learn how to wrap images onto polygons and how to load and manage multiple textures at once.

Chapter 9—Texture Mapping: Beyond the Basics

In this chapter, you'll learn how to generate texture coordinates automatically, use advanced filtering modes, and use built-in hardware support for texture compression. You'll also learn about OpenGL's powerful texture combiner functionality.

Chapter 10—Curves and Surfaces

The simple triangle is a powerful building block. This chapter gives you some tools for manipulating the mighty triangle. You'll learn about some of OpenGL's built-in quadric surface generation functions and ways to use automatic tessellation to break complex shapes into smaller, more digestible pieces. You'll also explore the utility functions that evaluate Bézier and NURBS curves and surfaces. You can use these functions to create complex shapes with an amazingly small amount of code.

Chapter 11—It's All About the Pipeline: Faster Geometry Throughput

For this chapter, we show you how to build complex 3D objects out of smaller, less complex 3D objects. We introduce OpenGL display lists and vertex arrays for improving performance and organizing your models. You'll also learn how to create a detailed analysis showing how to best represent large, complex models.

Chapter 12—Interactive Graphics

This chapter explains two OpenGL features: selection and feedback. These groups of functions make it possible for the user to interact with objects in the scene. You can also get rendering details about any single object in the scene.

Part II: OpenGL Everywhere

Chapter 13—Wiggle: OpenGL on Windows

Here, you'll learn how to write real Windows (message-based) programs that use OpenGL. You'll learn about Microsoft's "wiggle" functions that glue OpenGL rendering code to Windows device contexts. You'll also learn how to respond to Windows messages.

Chapter 14—OpenGL on the Mac OS X

In this chapter, you'll learn how to use OpenGL in native MacOS X applications. Sample programs show you how to start working in Carbon or Cocoa using the Xcode development environment.

Chapter 15—GLX: OpenGL on Linux

This chapter discusses GLX, the OpenGL extension used to support OpenGL applications through the X Window System on Unix and Linux. You'll learn how to create and manage OpenGL contexts as well as how to create OpenGL drawing areas with several of the common GUI toolkits.

Part III: OpenGL: The Next Generation

Chapter 16—Buffer Objects: It's Your Video Memory; You Manage It!

In this chapter, you'll learn about OpenGL 1.5's vertex buffer object feature. Buffer objects allow you to store vertex array data in memory that can be more efficiently accessed by the GPU, such as local VRAM or AGP-mapped system memory.

Chapter 17—Occlusion Queries: Why Do More Work Than You Need To?

Here, you'll learn about OpenGL 1.5's occlusion query mechanism. This feature effectively lets you perform an inexpensive test-render of objects in your scene to find out whether they will be hidden behind other objects, in which case you can save time by not drawing the actual full-detail version.

Chapter 18—Depth Textures and Shadows

This chapter covers OpenGL 1.4's depth textures and shadow comparisons. You'll learn how to introduce real-time shadow effects to your scene, regardless of the geometry's complexity.

Chapter 19—Programmable Pipeline: This Isn't Your Father's OpenGL

Out with the old, in with the new. This chapter revisits the conventional fixed functionality pipeline before introducing the new programmable vertex and fragment pipeline stages. Programmability allows you to customize your rendering in ways never before possible using shader programs.

Chapter 20—Low-Level Shading: Coding to the Metal

In this chapter, you'll learn about the low-level shader extensions: `ARB_vertex_program` and `ARB_fragment_program`. You can use them to customize your rendering via shader programs in a language reminiscent of assembly code, offering full control over the underlying hardware.

Chapter 21—High-Level Shading: The Real Slim Shader

Here, we discuss the OpenGL Shading Language, the high-level counterpart to the low-level extensions. GLSL is a C-like language that gives you increased functionality and productivity.

Chapter 22—Vertex Shading: Do-It-Yourself Transform, Lighting, and Texgen

This chapter illustrates the usage of vertex shaders by surveying a handful of examples, including lighting, fog, squash and stretch, and skinning.

Chapter 23—Fragment Shading: Empower Your Pixel Processing

Again, you learn by example—this time with a variety of fragment shaders. Examples include per-pixel lighting, color conversion, image processing, and procedural texturing. Some of these examples also use vertex shaders; these examples are representative of real-world usage, where

you often find vertex and fragment shaders paired together.

Conventions Used in This Book

The following typographic conventions are used in this book:

- Code lines, commands, statements, variables, and any text you type or see onscreen appear in a **computer** typeface.
- Placeholders in syntax descriptions appear in an *italic computer* typeface. Replace the placeholder with the actual filename, parameter, or whatever element it represents.
- *Italics* highlight technical terms when they first appear in the text and are being defined.

About the Companion CD

The CD that comes with the *OpenGL SuperBible* is packed with sample programs, toolkits, source code, and documentation—everything but the kitchen sink! We dig up stuff to put on this CD all the way until press time, so check out the `readme.txt` file in the root of the CD for a complete list of all the goodies we include.

The CD contains some basic organization. Off the root directory, you'll find

\Examples— Beneath this directory, you'll find a directory for each chapter in the book that has programming examples. Each sample has a real name (as opposed to **sample 5.2c**), so you can browse the CD with ease and run anything that looks interesting when you first get the book.

\Tools— A collection of third-party tools and libraries appears here. Each has its own subdirectory and documentation from the original vendor. Sample programs throughout the book use some of these tools (GLUT in particular).

\Demos— This directory contains a collection of OpenGL demo programs. These programs all showcase the rendering capabilities of OpenGL. Some are free; some are commercial demos.

For support issues with the CD, please contact Sams Publishing (www.samspublishing.com). Other OpenGL questions, more samples and tutorials, and, of course, the inevitable list of errata are posted on the book's Web site at

<http://www.starstonesoftware.com/OpenGL>

Building the Sample Programs

All the sample programs in this book are written in C. They should be easy to move to C++ for those so inclined, but C usually provides a larger audience for a book such as this, and is readily understood by anyone who might prefer C++. Most programs also use the **glTools** library, which is simply a collection of OpenGL-friendly utility functions written by the authors. The **glTools** header file (**gltools.h**) and source **.c** files are listed in the **\common** directory beneath the **\Examples** directories on the CD.

Beneath **\Examples**, you will find a directory for each OS platform supported. Windows project files are written in Visual C++ 6.0. The reason is that many have chosen not to upgrade to the new .NET-biased version of the tools, and those who have (myself included on many projects) can easily import the projects.

MacOS X sample project files were made with Xcode version 1.1. Anyone still using Project Builder should upgrade...it's free, and it will add years to your life. All Mac samples were tested on OS X version 10.3.3. At the time of printing, Apple had not released drivers that support the OpenGL

shading language. Latent Mac bugs could show up when it does. Check the Web site referenced in the preceding section for any necessary updates.

Linux make files are also provided. There are 10,392,444,224,229,349,244,281,999.4 different ways to configure a Linux environment. Well, maybe not quite that many. Without meaning to sound too harsh, you are pretty much on your own. Because Xcode uses a similar gnu compiler that many Linux environments use, you shouldn't have too many problems. I expect some additional notes and tutorials will show up on the book's Web site over time.

Dare to Dream in 3D!

Once upon a time in the early 1980s, I was looking at a computer at an electronics store. The salesman approached and began making his pitch. I told him I was just learning to program and considering an Amiga over his model. I was briskly informed that I needed to get serious with a computer that the rest of the world was using. An Amiga, he told me, was not good for anything but "making pretty pictures." No one, he assured me, could make a living making pretty pictures on his computer.

This was possibly the worst advice I have ever received in the history of bad advice. A few years ago, I forgot about being a "respectable" database/enterprise/yada-yada-yada developer. Now I write cool graphics programs, teach graphics programming, own my own software company, and generally have more fun with my career than should probably be allowed by law!

I hope I can give you some better advice today. Whether you want to write games, create military simulations, develop scientific visualizations, or visualize large corporate databases, OpenGL is the perfect API. It will meet you where you are as a beginner, and it will empower you when you become your own 3D guru. And yes, my friend, you *can* make a living making pretty pictures with your computer!

—Richard S. Wright, Jr.

Part I: Classic OpenGL

The following 12 chapters are about fixed pipeline 3D graphics rendering. In recent years, hardware on the PC platform has become quite mature, and sophisticated real-time graphics have become commonplace. The basic approach has not changed. We are still playing connect the dots and using triangles to make solid geometry. Performance now can be overwhelming, but by using the techniques in Part I of this book, you can create high-performance and stunning 3D graphics effects.

Although many advanced, sophisticated effects are built on the more flexible capabilities explored in Part III, "OpenGL: The Next Generation," the fixed pipeline capabilities of OpenGL remain the bedrock on which all else is built. We know that dedicated hardware is usually faster than flexible programmable hardware, and we can expect that for basic rendering needs the fixed pipeline functionality of OpenGL will remain with us for some time.

Chapter 1. Introduction to 3D Graphics and OpenGL

by Richard S. Wright, Jr.

What's This All About?

This book is about OpenGL, a programming interface for creating real-time 3D graphics. Before we begin talking about what OpenGL is and how it works, you should have at least a high-level understanding of real-time 3D graphics in general. Perhaps you picked up this book because you want to learn to use OpenGL, but you already have a good grasp of real-time 3D principles. If so, great: Skip directly to [Chapter 2](#), "Using OpenGL." If you bought this book because the pictures look cool and you want to learn how to do this on your PC...you should probably start here.

A Brief History of Computer Graphics

The first computers consisted of rows and rows of switches and lights. Technicians and engineers worked for hours, days, or even weeks to program these machines and read the results of their calculations. Patterns of illuminated bulbs conveyed useful information to the computer users, or some crude printout was provided. You might say that the first form of computer graphics was a panel of blinking lights. (This idea is supported by stories of early programmers writing programs that served no useful purpose other than creating patterns of blinking and chasing lights!)

Times have changed. From those first "thinking machines," as some called them, sprang fully programmable devices that printed on rolls of paper using a mechanism similar to a teletype machine. Data could be stored efficiently on magnetic tape, on disk, or even on rows of hole-punched paper or stacks of paper-punch cards. The "hobby" of computer graphics was born the day computers first started printing. Because each character in the alphabet had a fixed size and shape, creative programmers in the 1970s took delight in creating artistic patterns and images made up of nothing more than asterisks (*).

Enter the CRT

Paper as an output medium for computers is useful and persists today. Laser printers and color inkjet printers have replaced crude ASCII art with crisp presentation quality and near photographic reproductions of artwork. Paper, however, can be expensive to replace on a regular basis, and using it consistently is wasteful of our natural resources, especially because most of the time we don't really need hard-copy output of calculations or database queries.

The cathode ray tube (CRT) was a tremendously useful addition to the computer. The original computer monitors, CRTs were initially just video terminals that displayed ASCII text just like the first paper terminals—but CRTs were perfectly capable of drawing points and lines as well as alphabetic characters. Soon, other symbols and graphics began to supplement the character terminal. Programmers used computers and their monitors to create graphics that supplemented textual or tabular output. The first algorithms for creating lines and curves were developed and published; computer graphics became a science rather than a pastime.

The first computer graphics displayed on these terminals were *two-dimensional*, or *2D*. These flat lines, circles, and polygons were used to create graphics for a variety of purposes. Graphs and plots could display scientific or statistical data in a way that tables and figures could not. More adventurous programmers even created simple arcade games such as *Lunar Lander* and *Pong* using simple graphics consisting of little more than line drawings that were refreshed (redrawn) several times a second.

The term *real-time* was first applied to computer graphics that were animated. A broader use of the word in computer science simply means that the computer can process input as fast or faster than the input is being supplied. For example, talking on the phone is a real-time activity in which humans participate. You speak and the listener hears your communication immediately and responds, allowing you to hear immediately and respond again, and so on. In reality, there is some delay involved due to the electronics, but the delay is usually imperceptible to those having the conversation. In contrast, writing a letter is not a real-time activity.

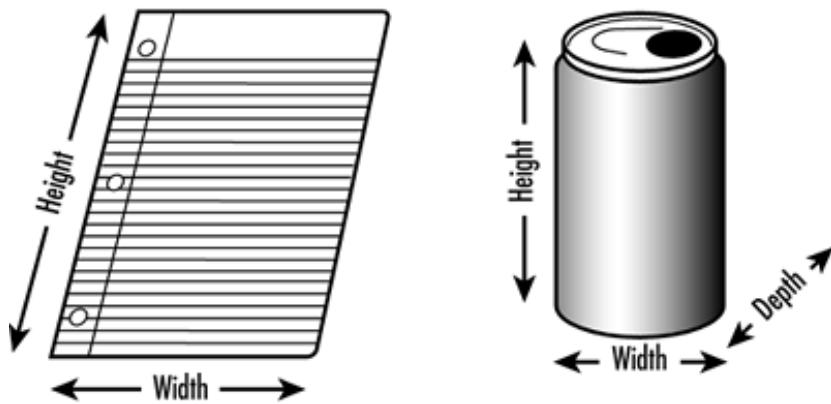
Applying the term *real-time* to computer graphics means that the computer is producing an animation or sequence of images directly in response to some input, such as joystick movement,

keyboard strokes, and so on. Real-time computer graphics can display a wave form being measured by electronic equipment, numerical readouts, or interactive games and visual simulations.

Going 3D

The term *three-dimensional*, or *3D*, means that an object being described or displayed has three dimensions of measurement: width, height, and depth. An example of a two-dimensional object is a piece of paper on your desk with a drawing or writing on it, having no perceptible depth. A three-dimensional object is the can of soda next to it. The soft drink can is round (width and height) and tall (depth). Depending on your perspective, you can alter which side of the can is the width or height, but the fact remains that the can has three dimensions. [Figure 1.1](#) shows how we might measure the dimensions of the can and piece of paper.

Figure 1.1. Measuring two- and three-dimensional objects.

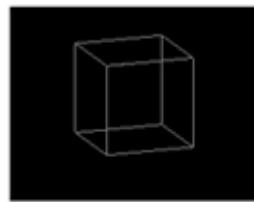


For centuries, artists have known how to make a painting appear to have real depth. A painting is inherently a two-dimensional object because it is nothing more than canvas with paint applied. Similarly, 3D computer graphics are actually two-dimensional images on a flat computer screen that provide an illusion of depth, or a third dimension.

2D + Perspective = 3D

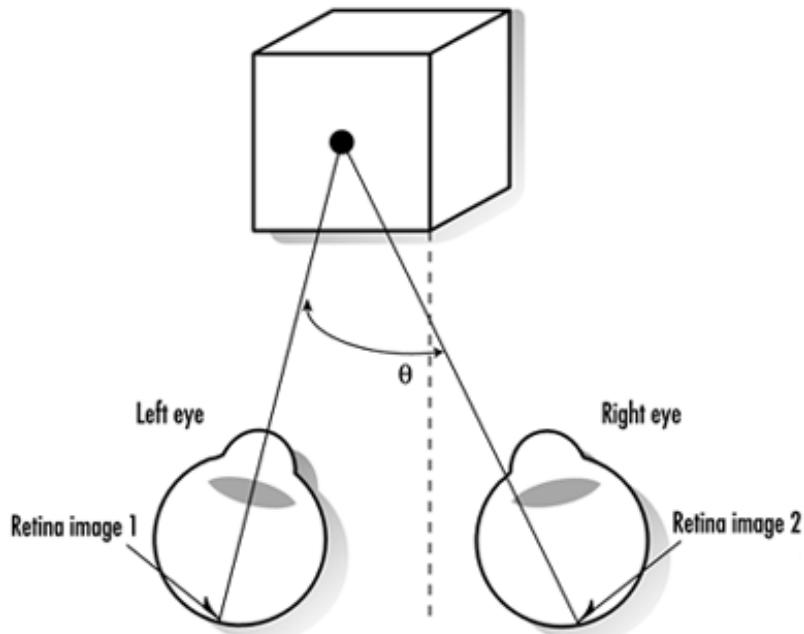
The first computer graphics no doubt appeared similar to [Figure 1.2](#), where you can see a simple three-dimensional cube drawn with 12 line segments. What makes the cube look three-dimensional is *perspective*, or the angle between the lines that lend the illusion of depth.

Figure 1.2. A simple wireframe 3D cube.



To truly see in 3D, you need to actually view an object with both eyes or supply each eye with separate and unique images of the object. Look at [Figure 1.3](#). Each eye receives a two-dimensional image that is much like a temporary photograph displayed on each retina (the back part of your eye). These two images are slightly different because they are received at two different angles. (Your eyes are spaced apart on purpose.) The brain then combines these slightly different images to produce a single, composite 3D picture in your head.

Figure 1.3. How you see three dimensions.

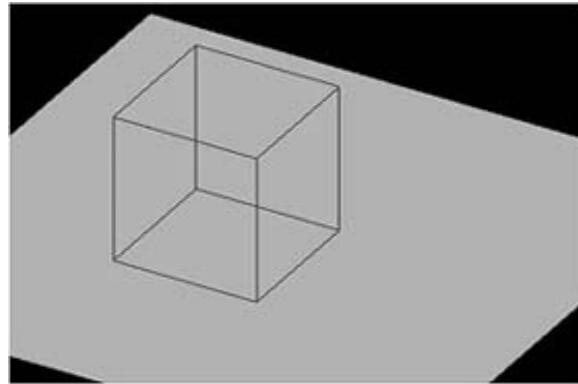


In [Figure 1.3](#), the angle between the images becomes smaller as the object goes farther away. You can amplify this 3D effect by increasing the angle between the two images. Viewmasters (those hand-held stereoscopic viewers you probably had as a kid) and 3D movies capitalize on this effect by placing each of your eyes on a separate lens or by providing color-filtered glasses that separate two superimposed images. These images are usually over-enhanced for dramatic or cinematic purposes. Of late this effect has become more popular on the PC as well. Shutter glasses that work with your graphics card and software will switch between one eye and the other, with a changing perspective displayed onscreen to each eye, thus giving a "true" stereo 3D experience. Unfortunately, many people complain that this effect gives them a headache or makes them dizzy!

A computer screen is one flat image on a flat surface, not two images from different perspectives falling on each eye. As it turns out, most of what is considered to be 3D computer graphics is actually an approximation of true 3D. This approximation is achieved in the same way that artists have rendered drawings with apparent depth for years, using the same tricks that nature provides for people with one eye.

You might have noticed at some time in your life that if you cover one eye, the world does not suddenly fall flat! What happens when you cover one eye? You might think you are still seeing in 3D, but try this experiment: Place a glass or some other object just out of arm's reach, off to your left side. (If it is close, this trick won't work.) Cover your right eye with your right hand and reach for the glass. (Maybe you should use an empty plastic one!) Notice that you have a more difficult time estimating how much farther you need to reach (if at all) before you touch the glass. Now, uncover your right eye and reach for the glass, and you can easily discern how far you need to lean to reach the glass. You now know why people with one eye often have difficulty with distance perception.

Perspective alone is enough to create the appearance of three dimensions. Note the cube shown previously in [Figure 1.2](#). Even without coloring or shading, the cube still has the appearance of a three-dimensional object. Stare at the cube for long enough, however, and the front and back of the cube switch places. Your brain is confused by the lack of any surface coloration in the drawing. [Figure 1.4](#) shows the output from the sample program BLOCK from this chapter's subdirectory on the CD-ROM. Run this program as we progress toward a more and more realistic appearing cube. We see here that the cube resting on a plane has an exaggerated perspective but still can produce the "popping" effect when you stare at it. By pressing the spacebar, you will progress toward a more and more believable image.

Figure 1.4. A line-drawn three-dimensional cube.

3D Artifacts

The reason the world doesn't become suddenly flat when you cover one eye is that many of a 3D world's effects are still present when viewed two-dimensionally. The effects are just enough to trigger your brain's ability to discern depth. The most obvious cue is that nearby objects appear larger than distant objects. This perspective effect is called *foreshortening*. This effect and color changes, textures, lighting, shading, and variations of color intensities (due to lighting) together add up to our perception of a three-dimensional image. In the next section, we take a survey of these tricks.

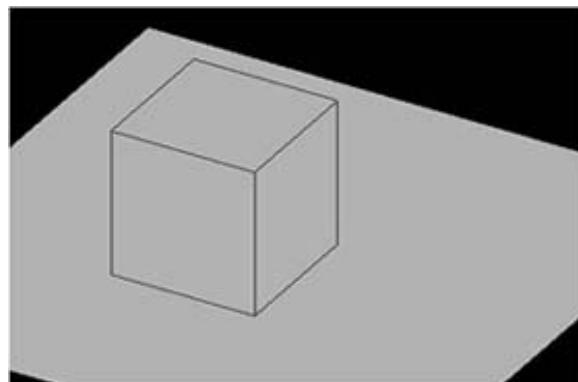
A Survey of 3D Effects

Now you have some idea that the illusion of 3D is created on a flat computer screen by means of a bag full of perspective and artistic tricks. Let's review some of these effects so we can refer to them later in the book, and you'll know what we are talking about.

The first term you should know is *render*. Rendering is the act of taking a geometric description of a three-dimensional object and turning it into an image of that object onscreen. All the following 3D effects are applied when rendering the objects or scene.

Perspective

Perspective refers to the angles between lines that lend the illusion of three dimensions. [Figure 1.4](#) shows a three-dimensional cube drawn with lines. This is a powerful illusion, but it can still cause perception problems as we mentioned earlier. (Just stare at this cube for a while, and it starts popping in and out.) In [Figure 1.5](#), on the other hand, the brain is given more clues as to the true orientation of the cube because of hidden line removal. You expect the front of an object to obscure the back of the object from view. For solid surfaces, we call this *hidden surface removal*.

Figure 1.5. A more convincing solid cube.

Color and Shading

If we stare at the cube in [Figure 1.5](#) long enough, we can convince ourselves that we are looking at a recessed image, and not the outward surfaces of a cube. To further our perception, we must move beyond line drawing and add color to create solid objects. [Figure 1.6](#) shows what happens when we naively add red to the color of the cube. It doesn't look like a cube anymore. By applying different colors to each side, as shown in [Figure 1.7](#), we regain our perception of a solid object.

Figure 1.6. Adding color alone can create further confusion.

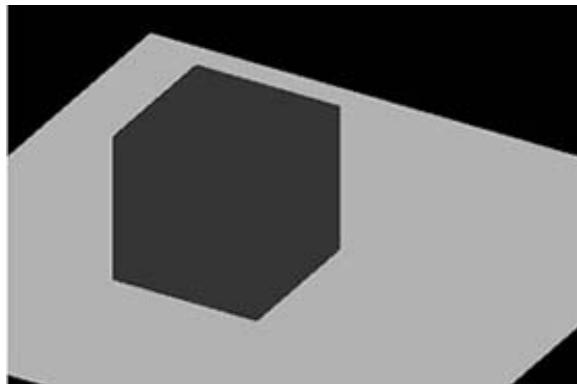
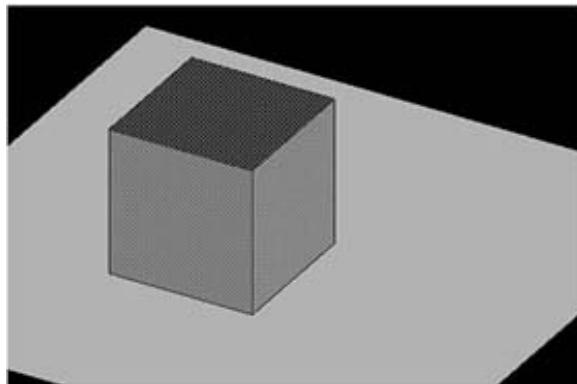


Figure 1.7. Adding different colors increases the illusion of three dimensions.



Light and Shadows

Making each side of the cube a different color helps your eye pick out the different sides of the object. By shading each side appropriately, we can give the cube the appearance of being one solid color (or material) but also show it is illuminated by a light at an angle, as shown in [Figure 1.8](#). [Figure 1.9](#) goes a step further by adding a shadow behind the cube. Now we are simulating the effects of light on one or more objects and their interactions. Our illusion at this point is very convincing.

Figure 1.8. Proper shading creates the illusion of illumination.

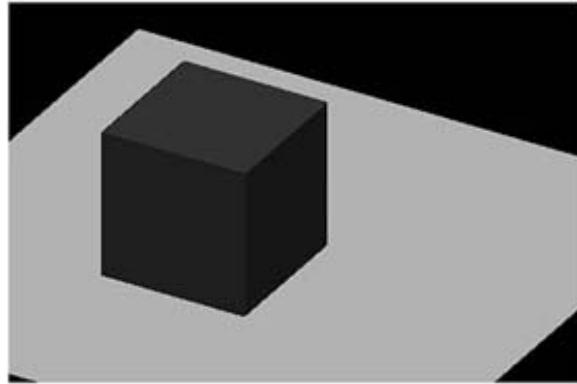
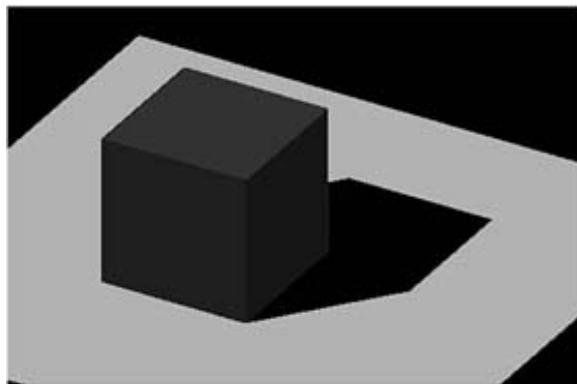


Figure 1.9. Adding a shadow to further increase realism.



Texture Mapping

Achieving a high level of realism with nothing but thousands or millions of tiny lit and shaded polygons is a matter of brute force and a lot of hard work. Unfortunately, the more geometry you throw at graphics hardware, the longer it takes to render. A clever technique allows you to use simpler geometry but achieve a higher degree of realism. This technique takes an image, such as a photograph of a real surface or detail, and then applies that image to the surface of a polygon.

Instead of plain-colored materials, you can have wood grains, cloth, bricks, and so on. This technique of applying an image to a polygon to supply additional detail is called *texture mapping*. The image you supply is called a *texture*, and the individual elements of the texture are called *texels*. Finally, the process of stretching or compressing the texels over the surface of an object is called *filtering*. [Figure 1.10](#) shows the now familiar cube example with textures applied to each polygon.

Figure 1.10. Texture mapping adds detail without adding additional geometry.



Fog

Most of us know what fog is. Fog is an atmospheric effect that adds haziness to objects in a scene, which is usually a relation of how far away the objects in the scene are from the viewer and how thick the fog is. Objects very far away (or nearby if the fog is thick) might even be totally obscured.

[Figure 1.11](#) shows the skyfly GLUT demo (included with the GLUT distribution on the CD-ROM) with fog enabled. Note how the fog lends substantially to the believability of the terrain.

Figure 1.11. Fog effects provide a convincing illusion for wide open spaces.

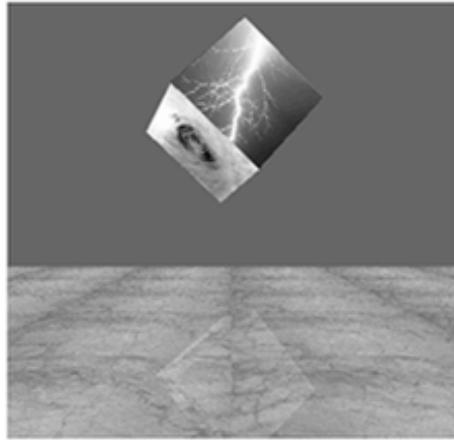


Blending and Transparency

Blending is the combination of colors or objects on the screen. This is similar to the effect you get with double-exposure photography, where two images are superimposed. You can use the blending effect for a variety of purposes. By varying the amount each object is blended with the scene, you can make objects look transparent such that you see the object and what is behind it (such as glass or a ghost image).

You can also use blending to achieve an illusion of reflection, as shown in [Figure 1.12](#). You see a textured cube rendered twice. First, the cube is rendered upside down below the floor level. The marble floor is then blended with the scene, allowing the cube to show through. Finally, the cube is drawn again right side up and floating over the floor. The result is the appearance of a reflection in a shiny marble surface.

Figure 1.12. Blending used to achieve a reflection effect.



Antialiasing

Aliasing is an effect that is visible onscreen due to the fact that an image consists of discrete pixels. In [Figure 1.13](#), you can see that the lines that make up the cube have jagged edges (sometimes called *jaggies*). By carefully blending the lines with the background color, you can eliminate the jagged edges and give the lines a smooth appearance, as shown in [Figure 1.14](#). This blending technique is called *antialiasing*. You can also apply antialiasing to polygon edges, making an object or scene look more realistic. This Holy Grail of real-time graphics is often referred to as photo-realistic rendering.

Figure 1.13. Cube with jagged lines drawn.

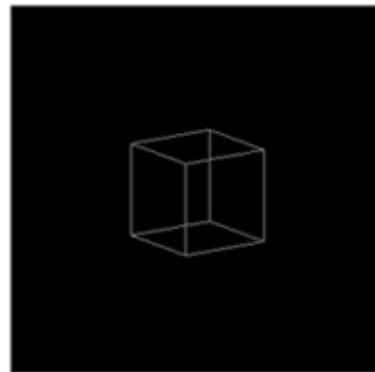
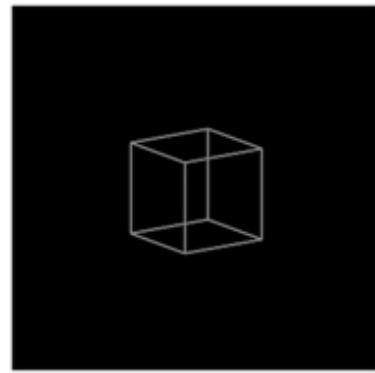


Figure 1.14. Cube with smoother antialiased lines.



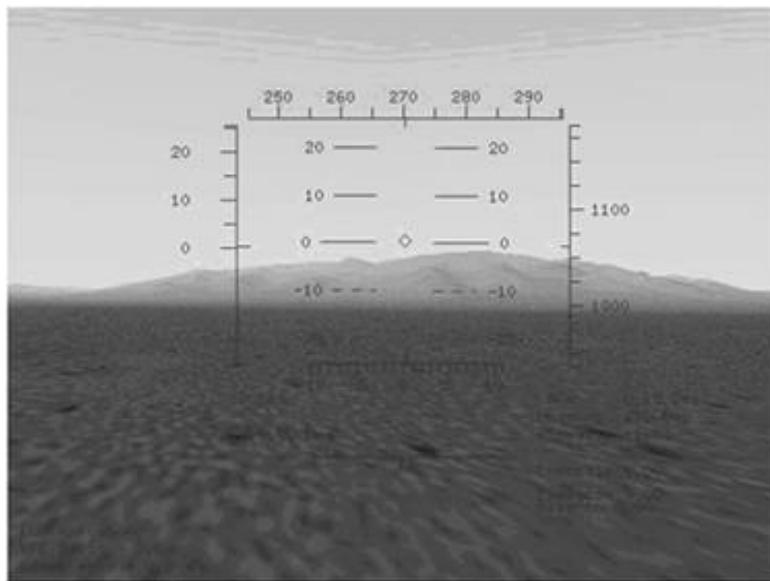
Common Uses for 3D Graphics

Three-dimensional graphics have many uses in modern computer applications. Applications for real-time 3D graphics range from interactive games and simulations to data visualization for scientific, medical, or business uses. Higher-end 3D graphics find their way into movies and technical and educational publications as well.

Real-Time 3D

As defined earlier, real-time 3D graphics are animated and interactive with the user. One of the earliest uses for real-time 3D graphics was in military flight simulators. Even today, flight simulators are a popular diversion for the home enthusiast. [Figure 1.15](#) shows a screenshot from a popular flight simulator that uses OpenGL for 3D rendering (www.flightgear.org).

Figure 1.15. A popular OpenGL-based flight simulator from Flight Gear.



The applications for 3D graphics on the PC are almost limitless. Perhaps the most common use today is for computer gaming. Hardly a title ships today that does not require a 3D graphics card in your PC to play. 3D has always been popular for scientific visualization and engineering applications, but the explosion of cheap 3D hardware has empowered these applications like never before. Business applications are also taking advantage of the new availability of hardware to incorporate more and more complex business graphics and database mining visualization techniques. Even the modern GUI is being effected, and is beginning to evolve to take advantage of 3D hardware capabilities. The new Macintosh OS X, for example, uses OpenGL to render all its windows and controls for a powerful and eye-popping visual interface.

[Figures 1.16](#) through [1.20](#) show some of the myriad applications of real-time 3D graphics on the modern PC. All these images were rendered using OpenGL.

Figure 1.16. 3D graphics used for computer-aided design (CAD).

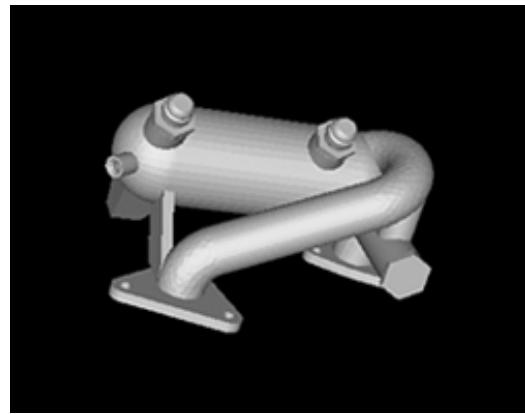


Figure 1.20. 3D graphics used for entertainment (Descent 3 from Outrage Entertainment, Inc.).



Figure 1.17. 3D graphics used for architectural or civil planning (image courtesy of Real 3D, Inc.).



Figure 1.18. 3D graphics used for medical imaging applications (VolView by Kitware).

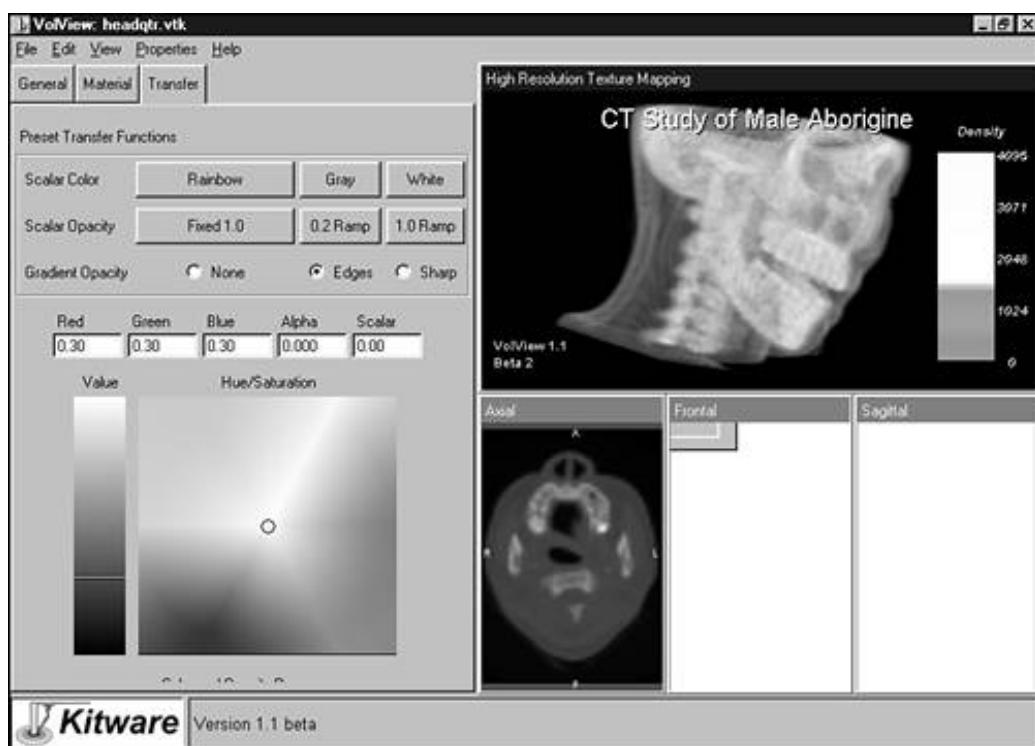
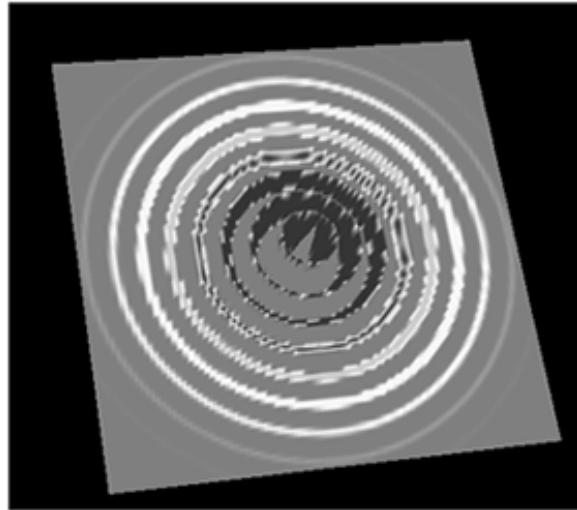


Figure 1.19. 3D graphics used for scientific visualization.



Non-Real-Time 3D

Some compromise is required for real-time 3D applications. Given more processing time, you can generate higher quality 3D graphics. Typically, you design models and scenes, and a ray tracer processes the definition to produce a high-quality 3D image. The typical process is that some modeling application uses real-time 3D graphics to interact with the artist to create the content. Then the frames are sent to another application (the ray tracer), which renders the image. Rendering a single frame for a movie such as *Toy Story* could take hours on a very fast computer, for example. The process of rendering and saving many thousands of frames generates an animated sequence for playback. Although the playback might appear real-time, the content is not interactive, so it is not considered real-time, but rather pre-rendered.

[Figure 1.21](#) shows an example from the CD-ROM. This spinning image shows crystal letters that spell *OpenGL*. The letters are transparent and fully antialiased and show a myriad of reflections and shadow effects. The file to play this animation on your computer is `opengl.avi` in the root directory of the CD-ROM that accompanies this book. This large file (more than 35MB!) took hours to render.

Figure 1.21. High-quality pre-rendered animation.



Basic 3D Programming Principles

Now, you have a pretty good idea of the basics of real-time 3D. We've covered some terminology and some sample applications on the PC. How do you actually create these images on your PC? Well, that's what the rest of this book is about! You still need a little more introduction to the basics, which we present here.

Immediate Mode and Retained Mode (Scene Graphs)

There are two different approaches to programming APIs for real-time 3D graphics. The first approach is called *retained mode*. In retained mode, you provide the API or toolkit with a description of your objects and the scene. The graphics package then creates the image onscreen.

The only additional manipulation you might make is to give commands to change the location and viewing orientation of the user (also called the *camera*) or other objects in the scene.

This type of approach is typical of ray tracers and many commercial flight simulators and image generators. Programmatically, the structure that is built is called a *scene graph*. The scene graph is a data structure (usually a DAG, or directed acyclic graph, for you computer science majors) that contains all the objects in your scene and their relationships to one another. Many high-level toolkits or "game engines" use this approach. The programmer doesn't need to understand the finer points of rendering, only that he has a model or database that will be handed over to the graphics library, which takes care of the rendering.

The second approach to 3D rendering is called *immediate mode*. Most retained mode APIs or scene graphs use an immediate mode API internally to actually perform the rendering. In immediate mode, you don't describe your models and environment from as high a level. Instead, you issue commands directly to the graphics processor that has an immediate effect on its state and the state of all subsequent commands.

With immediate mode, new commands have no effect on rendering commands that have already been executed. This gives you a great deal of low-level control. For example, you can render a series of textured unlit polygons to represent the sky. Then you issue a command to turn off texturing, followed by a command to turn on lighting. Thereafter, all geometry (probably drawn on the ground) that you render is affected by the light but is not textured with the sky image.

Coordinate Systems

Let's consider now how we describe objects in three dimensions. Before you can specify an object's location and size, you need a frame of reference to measure and locate against. When you draw lines or plot points on a simple flat computer screen, you specify a position in terms of a row and column. For example, a standard VGA screen has 640 pixels from the left to right and 480 pixels from top to bottom. To specify a point in the middle of the screen, you specify that a point should be plotted at (320,240)—that is, 320 pixels from the left of the screen and 240 pixels down from the top of the screen.

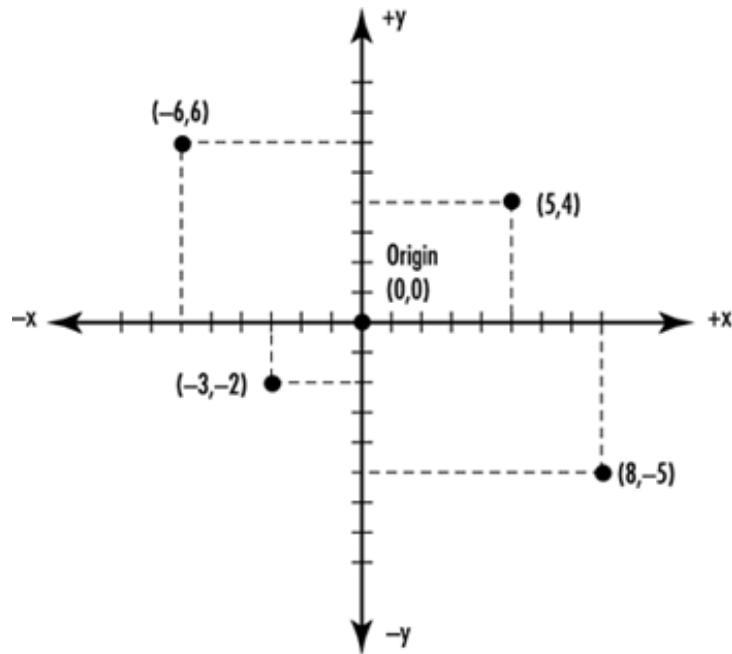
In OpenGL, or almost any 3D API, when you create a window to draw in, you must also specify the *coordinate system* you want to use and how to map the specified coordinates into physical screen pixels. Let's first see how this applies to two-dimensional drawing and then extend the principle to three dimensions.

2D Cartesian Coordinates

The most common coordinate system for two-dimensional plotting is the Cartesian coordinate system. Cartesian coordinates are specified by an x coordinate and a y coordinate. The x coordinate is a measure of position in the horizontal direction, and y is a measure of position in the vertical direction.

The *origin* of the Cartesian system is at $x=0$, $y=0$. Cartesian coordinates are written as coordinate pairs in parentheses, with the x coordinate first and the y coordinate second, separated by a comma. For example, the origin is written as (0,0). [Figure 1.22](#) depicts the Cartesian coordinate system in two dimensions. The x and y lines with tick marks are called the *axes* and can extend from negative to positive infinity. This figure represents the true Cartesian coordinate system pretty much as you used it in grade school. Today, differing window mapping modes can cause the coordinates you specify when drawing to be interpreted differently. Later in the book, you'll see how to map this true coordinate space to window coordinates in different ways.

Figure 1.22. The Cartesian plane.

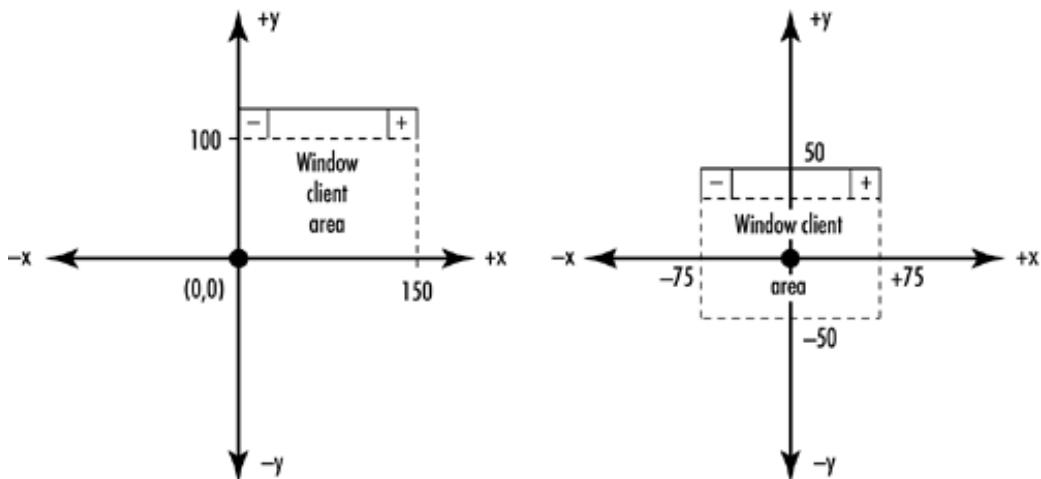


The x -axis and y -axis are perpendicular (intersecting at a right angle) and together define the xy plane. A *plane* is, most simply put, a flat surface. In any coordinate system, two axes (or two lines) that intersect at right angles define a plane. In a system with only two axes, there is naturally only one plane to draw on.

Coordinate Clipping

A window is measured physically in terms of pixels. Before you can start plotting points, lines, and shapes in a window, you must tell OpenGL how to translate specified coordinate pairs into screen coordinates. You do this by specifying the region of Cartesian space that occupies the window; this region is known as the clipping region. In two-dimensional space, the clipping region is the minimum and maximum x and y values that are inside the window. Another way of looking at this is specifying the origin's location in relation to the window. [Figure 1.23](#) shows two common clipping regions.

Figure 1.23. Two clipping regions.



In the first example, on the left of [Figure 1.23](#), x coordinates in the window range left to right from 0 to $+150$, and the y coordinates range bottom to top from 0 to $+100$. A point in the middle of the screen would be represented as $(75,50)$. The second example shows a clipping area with x coordinates ranging left to right from -75 to $+75$ and y coordinates ranging bottom to top from -50 to 50 .

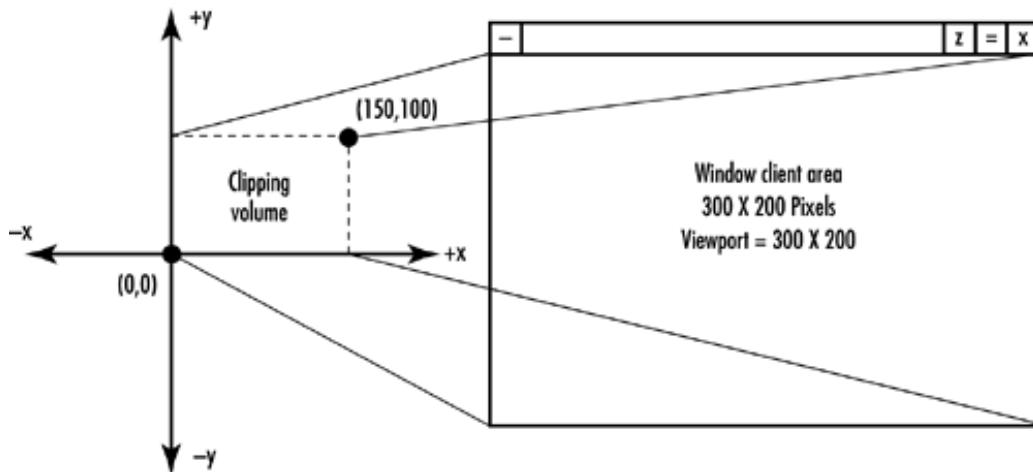
50 to +50. In this example, a point in the middle of the screen would be at the origin (0,0). It is also possible using OpenGL functions (or ordinary Windows functions for GDI drawing) to turn the coordinate system upside down or flip it right to left. In fact, the default mapping for Windows windows is for positive y to move down from the top to bottom of the window. Although useful when drawing text from top to bottom, this default mapping is not as convenient for drawing graphics.

Viewports: Mapping Drawing Coordinates to Window Coordinates

Rarely will your clipping area width and height exactly match the width and height of the window in pixels. The coordinate system must therefore be mapped from logical Cartesian coordinates to physical screen pixel coordinates. This mapping is specified by a setting known as the *viewport*. The viewport is the region within the window's client area that is used for drawing the clipping area. The viewport simply maps the clipping area to a region of the window. Usually, the viewport is defined as the entire window, but this is not strictly necessary; for instance, you might want to draw only in the lower half of the window.

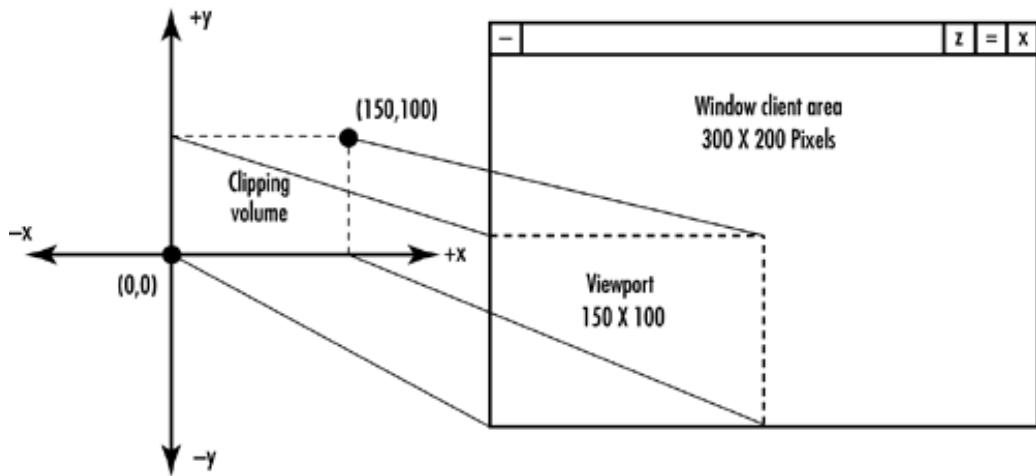
Figure 1.24 shows a large window measuring 300x200 pixels with the viewport defined as the entire client area. If the clipping area for this window were set to 0 to 150 along the x-axis and 0 to 100 along the y-axis, the logical coordinates would be mapped to a larger screen coordinate system in the viewing window. Each increment in the logical coordinate system would be matched by two increments in the physical coordinate system (pixels) of the window.

Figure 1.24. A viewport defined as twice the size of the clipping area.



In contrast, Figure 1.25 shows a viewport that matches the clipping area. The viewing window is still 300x200 pixels, however, and this causes the viewing area to occupy the lower-left side of the window.

Figure 1.25. A viewport defined as the same dimensions as the clipping area.



You can use viewports to shrink or enlarge the image inside the window and to display only a portion of the clipping area by setting the viewport to be larger than the window's client area.

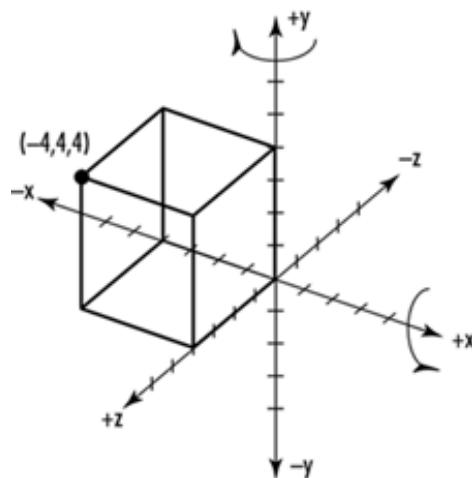
The Vertex—A Position in Space

In both 2D and 3D, when you draw an object, you actually compose it with several smaller shapes called *primitives*. Primitives are one- or two-dimensional entities or surfaces such as points, lines, and polygons (a flat, multisided shape) that are assembled in 3D space to create 3D objects. For example, a three-dimensional cube consists of six two-dimensional squares, each placed on a separate face. Each corner of the square (or of any primitive) is called a *vertex*. These vertices are then specified to occupy a particular coordinate in 3D space. A vertex is nothing more than a coordinate in 2D or 3D space. Creating solid 3D geometry is little more than a game of *connect-the-dots*! You'll learn about all the OpenGL primitives and how to use them in [Chapter 3, "Drawing in Space: Geometric Primitives](#) .

3D Cartesian Coordinates

Now, we extend our two-dimensional coordinate system into the third dimension and add a depth component. [Figure 1.26](#) shows the Cartesian coordinate system with a new axis, z . The z -axis is perpendicular to both the x - and y -axes. It represents a line drawn perpendicularly from the center of the screen heading toward the viewer. (We have rotated our view of the coordinate system from [Figure 1.22](#) to the left with respect to the y -axis and down and back with respect to the x -axis. If we hadn't, the z -axis would come straight out at you, and you wouldn't see it.) Now, we specify a position in three-dimensional space with three coordinates: x , y , and z . [Figure 1.26](#) shows the point $(-4, 4, 4)$ for clarification.

Figure 1.26. Cartesian coordinates in three dimensions.

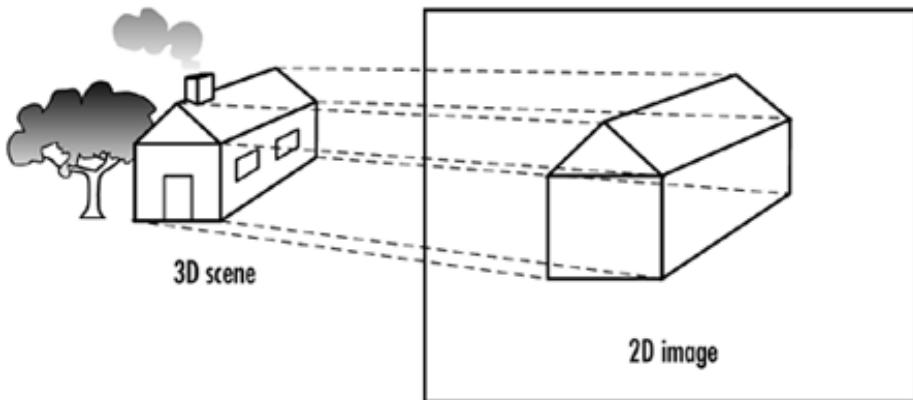


Projections: Getting 3D to 2D

You've seen how to specify a position in 3D space using Cartesian coordinates. No matter how we might convince your eye, however, pixels on a screen have only two dimensions. How does OpenGL translate these Cartesian coordinates into two-dimensional coordinates that can be plotted on a screen? The short answer is "Trigonometry and simple matrix manipulation." Simple? Well, not really; we could actually go on for many pages and lose most of our readers who didn't take or don't remember their linear algebra from college explaining this "simple" technique. You'll learn more about it in [Chapter 4](#), "Geometric Transformations: The Pipeline," and for a deeper discussion, you can check out the references in [Appendix A](#), "Further Reading." Fortunately, you don't need a deep understanding of the math to use OpenGL to create graphics. You might, however, discover that the deeper your understanding goes, the more powerful a tool OpenGL becomes!

The first concept you really need to understand is called *projection*. The 3D coordinates you use to create geometry are flattened or *projected* onto a 2D surface (the window background). It's like tracing the outlines of some object behind a piece of glass with a black marker. When the object is gone or you move the glass, you can still see the outline of the object with its angular edges. In [Figure 1.27](#), a house in the background is traced onto a flat piece of glass. By specifying the *viewing volume* that you want displayed in your window and how it should be transformed.

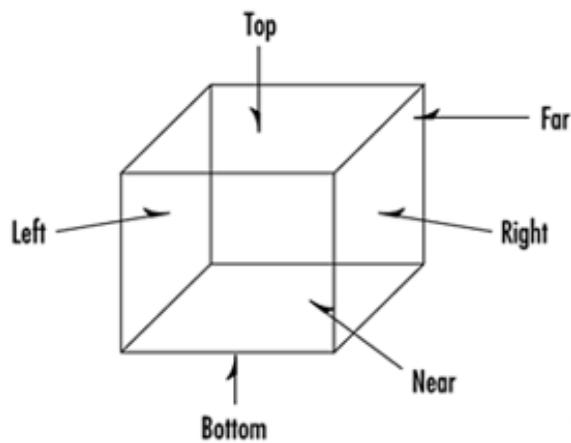
Figure 1.27. A 3D image projected onto a 2D surface.



Orthographic Projections

You are mostly concerned with two main types of projections in OpenGL. The first is called an *orthographic* or parallel projection. You use this projection by specifying a square or rectangular viewing volume. Anything outside this volume is not drawn. Furthermore, all objects that have the same dimensions appear the same size, regardless of whether they are far away or nearby. This type of projection (shown in [Figure 1.28](#)) is most often used in architectural design, computer-aided design (CAD), or 2D graphs. Frequently, you will use an orthographic projection to add text or 2D overlays on top of your 3D graphic scene.

Figure 1.28. The clipping volume for an orthographic projection.

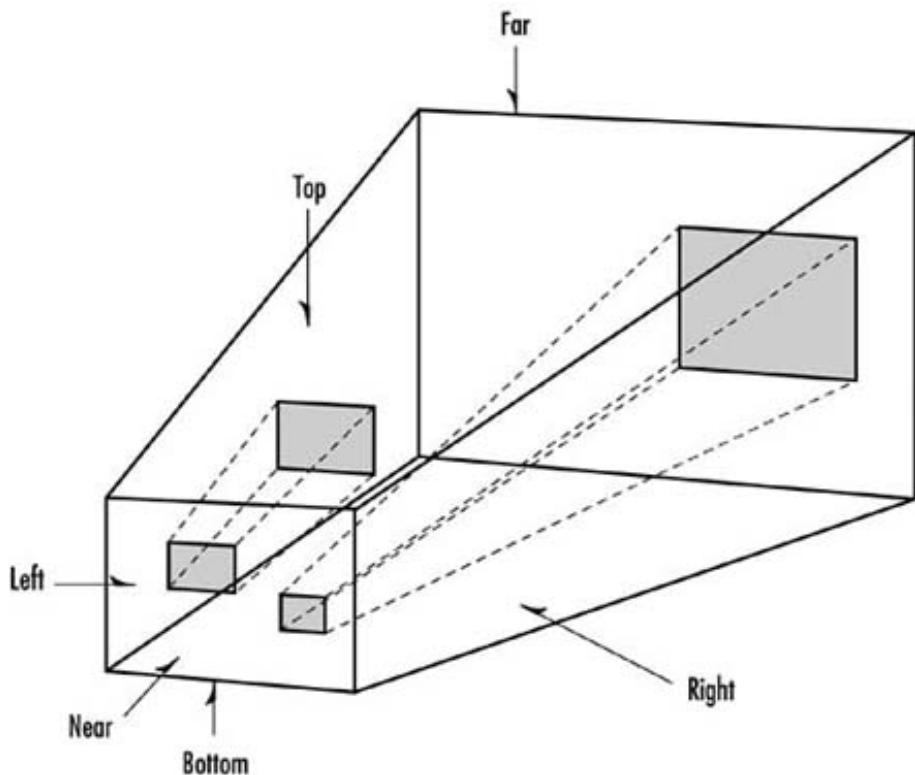


You specify the viewing volume in an orthographic projection by specifying the far, near, left, right, top, and bottom clipping planes. Objects and figures that you place within this viewing volume are then projected (taking into account their orientation) to a 2D image that appears on your screen.

Perspective Projections

The second and more common projection is the *perspective projection*. This projection adds the effect that distant objects appear smaller than nearby objects. The viewing volume (see [Figure 1.29](#)) is something like a pyramid with the top shaved off. The remaining shape is called the *frustum*. Objects nearer to the front of the viewing volume appear close to their original size, but objects near the back of the volume shrink as they are projected to the front of the volume. This type of projection gives the most realism for simulation and 3D animation.

Figure 1.29. The clipping volume (frustum) for a perspective projection.



Summary

In this chapter, we introduced the basics of 3D graphics. You saw why you actually need two images of an object from different angles to be able to perceive true three-dimensional space. You also saw the illusion of depth created in a 2D drawing by means of perspective, hidden line removal, coloring, shading, and other techniques. The Cartesian coordinate system was introduced for 2D and 3D drawing, and you learned about two methods used by OpenGL to project three-dimensional drawings onto a two-dimensional screen.

We purposely left out the details of how these effects are actually created by OpenGL. In the chapters that follow, you will find out how to employ these techniques and take maximum advantage of OpenGL's power. On the companion CD, you'll find one program for this chapter that demonstrates the 3D effects covered here. In the program BLOCK, pressing the spacebar advances you from a wireframe cube to a fully lit and textured block complete with shadow. You won't understand the code at this point, but it makes a powerful demonstration of what is to come. By the time you finish this book, you will be able to revisit this example and even be able to write it from scratch yourself.

Chapter 2. Using OpenGL

by *Richard S. Wright, Jr.*

WHAT YOU'LL LEARN IN THIS CHAPTER:

- [Where OpenGL came from and where it's going](#)
- [Which headers need to be included in your project](#)
- [How to use GLUT with OpenGL to create a window and draw in it](#)
- [How to set colors using RGB \(red, green, blue\) components](#)
- [How viewports and viewing volumes affect image dimensions](#)
- [How to perform a simple animation using double buffering](#)
- [How the OpenGL state machine works](#)
- [How to check for OpenGL errors](#)
- [How to make use of OpenGL extensions](#)

What Is OpenGL?

OpenGL is strictly defined as "a software interface to graphics hardware." In essence, it is a 3D graphics and modeling library that is highly portable and very fast. Using OpenGL, you can create elegant and beautiful 3D graphics with nearly the visual quality of a ray tracer. The greatest advantage to using OpenGL is that it is orders of magnitude faster than a ray tracer. Initially, it used algorithms carefully developed and optimized by Silicon Graphics, Inc. (SGI), an acknowledged world leader in computer graphics and animation. Over time OpenGL has evolved as other vendors have contributed their expertise and intellectual property to develop high-performance implementations of their own.

OpenGL is not a programming language like C or C++. It is more like the C runtime library, which provides some prepackaged functionality. There really is no such thing as an "OpenGL program," but rather a program the developer wrote that "happens" to use OpenGL as one of its Application Programming Interfaces (APIs). You might use the Windows API to access a file or the Internet, and you might use OpenGL to create real-time 3D graphics.

OpenGL is intended for use with computer hardware that is designed and optimized for the display and manipulation of 3D graphics. Software-only, "generic" implementations of OpenGL are also possible, and the Microsoft implementations fall into this category. With a software-only implementation, rendering may not be performed as quickly, and some advanced special effects may not be available. However, using a software implementation means that your program can potentially run on a wider variety of computer systems that may not have a 3D graphics card installed.

OpenGL is used for a variety of purposes, from CAD engineering and architectural applications to modeling programs used to create computer-generated monsters in blockbuster movies. The introduction of an industry-standard 3D API to mass-market operating systems such as Microsoft Windows and the Macintosh OS X has some exciting repercussions. With hardware acceleration and fast PC microprocessors becoming commonplace, 3D graphics are now typical components of consumer and business applications, not only of games and scientific applications.

Evolution of a Standard

The forerunner of OpenGL was IRIS GL from Silicon Graphics. Originally a 2D graphics library, it evolved into the 3D programming API for that company's high-end IRIS graphics workstations. These computers were more than just general-purpose computers; they had specialized hardware optimized for the display of sophisticated graphics. The hardware provided ultra-fast matrix transformations (a prerequisite for 3D graphics), hardware support for depth buffering, and other features.

Sometimes, however, the evolution of technology is hampered by the need to support legacy systems. IRIS GL had not been designed from the onset to have a vertex-style geometry processing interface, and it became apparent that to move forward SGI needed to make a clean break.

OpenGL is the result of SGI's efforts to evolve and improve IRIS GL's portability. The new graphics API would offer the power of GL but would be an "open" standard, with input from other graphics hardware vendors, and would allow for easier adaptability to other hardware platforms and operating systems. OpenGL would be designed from the ground up for 3D geometry processing.

The OpenGL ARB

An open standard is not really open if only one vendor controls it. SGI's business at the time was high-end computer graphics. Once you're at the top, you find that the opportunities for growth are somewhat limited. SGI realized that it would also be good for the company to do something good for the industry to help grow the market for high-end computer graphics hardware. A truly open standard embraced by a number of vendors would make it easier for programmers to create applications and content that is available for a wider variety of platforms. Software is what sells computers, and if SGI wanted to sell more computers, it needed more software that would run on its computers. Other vendors realized this, too, and the OpenGL Architecture Review Board (ARB) was born.

Although SGI controlled licensing of the OpenGL API, the founding members of the OpenGL ARB were SGI, Digital Equipment Corporation, IBM, Intel, and Microsoft. On July 1, 1992, Version 1.0 of the OpenGL specification was introduced. More recently, the ARB consists of many more members, many from the PC hardware community, and it meets four times a year to maintain and enhance the specification and to make plans to promote the OpenGL standard.

These meetings are open to the public, and nonmember companies can participate in discussions and even vote in straw polls. Permission to attend must be requested in advance, and meetings are kept small to improve productivity. Nonmember companies actually contribute significantly to the specification and do meaningful work on the conformance tests and other subcommittees.

Licensing and Conformance

An implementation of OpenGL is either a software library that creates three-dimensional images in response to the OpenGL function calls or a driver for a hardware device (usually a display card) that does the same. Hardware implementations are many times faster than software implementations and are now common even on inexpensive PCs.

A vendor who wants to create and market an OpenGL implementation must first license OpenGL from SGI. SGI provides the licensee with a sample implementation (entirely in software) and a device driver kit if the licensee is a PC hardware vendor. The vendor then uses this to create its own optimized implementation and can add value with its own extensions. Competition among

vendors typically is based on performance, image quality, and driver stability.

In addition, the vendor's implementation must pass the OpenGL conformance tests. These tests are designed to ensure that an implementation is complete (it contains all the necessary function calls) and produces 3D rendered output that is reasonably acceptable for a given set of functions.

Software developers do not need to license OpenGL or pay any fees to make use of OpenGL drivers. OpenGL is natively supported by the operating system, and licensed drivers are provided by the hardware vendors themselves. A free open source software implementation of OpenGL called *MESA* is included on the CD with this book. For legal reasons, MESA is not an "official" implementation of OpenGL, but the API is identical and we all know it is just OpenGL! Many Linux open source OpenGL hardware drivers are in fact based on the MESA source code.

The API Wars

Standards are good for everyone, except for vendors who think that they should be the only vendors customers can choose from because they know best what customers need. We have a special legal word for vendors who manage to achieve this status: *monopoly*. Most companies recognize that competition is good for everyone in the long run and will endorse, support, and even contribute to industry standards. An interesting diversion from this ideal occurred during OpenGL's youth on the Windows platform.

Enter DirectX

Few remember anymore what a "Windows Accelerator" is, but there was a time when you could buy a special graphics card that accelerated the 2D graphics commands used by Microsoft Windows. Although Graphics Device Interface (GDI) accelerated graphics cards were great for users of word processors or desktop publishing applications, they fell far short of adequate for PC game programmers. Early Windows-based games consisted primarily of puzzle or card games that did not need speedy animation. PC games that required fast animation, such as action video games, remained DOS-based programs that could control the entire screen and did not have to share system resources with other programs that might be running at the same time.

Microsoft made some attempts to win game programmers over to developing Windows native games, but early attempts to provide faster display access, such as the WinG API, still fell far short of the mark, and game programmers continued writing DOS programs that gave them direct access to graphics hardware. When Microsoft began shipping Windows 95, its first quasi-true 32-bit operating system for the consumer market, the company intended to put an end to DOS once and for all. The original Windows 95 Game Developers Kit included some new APIs for Windows aimed at game programmers. The most important of these was DirectDraw.

To remain competitive, graphics card vendors now needed a GDI driver *and* a DirectDraw driver, but the driver framework was meant to provide the same low-level (and fast) access to graphics hardware under Windows that developers were used to getting with DOS. This time, Microsoft was more successful, and Windows 95 native game titles that took advantage of the new "game" APIs began shipping. This group of APIs later came to be known as DirectX. DirectX now contains a whole family of APIs meant to empower multimedia developers on the Windows platform. It would be fair to compare DirectX on Windows to QuickTime on the Mac: Both offer a variety of multimedia services and APIs to the programmer.

The original intent for DirectX was direct low-level access to hardware features. This original purpose has been diluted over time as DirectX has come to include higher-level software functionality and additional layers over the low-level devices. What was originally the Windows Game Developers Kit has evolved over time to become the "DirectX Media APIs." Many of the DirectX APIs are independent of one another, and you can mix and match them at will. For example, you can use a third-party sound library with a Direct3D-rendered game or even use DirectSound with an OpenGL-rendered game.

OpenGL Comes to the PC

Around the same time that one group at Microsoft was focusing on establishing Windows as a viable gaming platform, OpenGL (which was a younger API at the time) began to gain some momentum on the PC as well and was being promoted by yet another group at Microsoft as the API of choice for scientific and engineering applications. Microsoft was even one of the founding members of the OpenGL ARB. Supporting OpenGL on the Windows platform would enable Microsoft to compete with UNIX-based workstations that had traditionally been the host of advanced visualization and simulation applications.

Originally, 3D graphics hardware for the PC was very expensive, so it was not frequently used for computer games. Only industries with deep pockets could afford such hardware, and as a result OpenGL first became popular in the fields of CAD, simulation, and scientific visualization. In these fields, performance was often a premium, so OpenGL was designed and evolved with performance as an important goal of the API. As 3D games became popular on the PC, OpenGL was applied to this domain as well and in some cases with great success.

By the time 3D graphics hardware for the PC became inexpensive enough to attract PC gamers, OpenGL was a mature and well-established 3D rendering API with a strong feature set. This timing coincided with Microsoft's attempts to promote its new Direct3D as a 3D rendering API for games. Ordinarily, a new, difficult-to-use, and relatively feature-weak API such as Direct3D would not have survived in the marketplace.

Many veteran 3D game programmers apply the unofficial term *API Wars* to this period of time when Microsoft and SGI were battling for the mind share of 3D game developers. Microsoft was a founding member of the OpenGL ARB and wanted OpenGL on Windows so that UNIX workstation software vendors could more easily port their scientific and engineering applications to Windows NT. Portability, however, was a two-edged sword; it also meant that developers who target Windows could later port their applications to other operating systems. PCs were well entrenched in the business workplace. Now it was time to go after the consumer market in a much bigger way.

OpenGL for Games?

When John Carmack, the author of one of the most popular 3D games of all time, rewrote his popular game *Quake* to use OpenGL over one weekend, his efforts set the gaming world abuzz. John demonstrated easily how 10 or so lines of OpenGL code required two to three pages of Direct3D code to accomplish the same task (rendering a few triangles). Many game programmers began to look carefully at OpenGL as a 3D rendering API suitable for games and not just "scientific" applications. John Carmack's company, ID Software, also had a policy of providing its games on several different platforms and operating systems. OpenGL simply made doing so much easier.

By this point, Microsoft had too much invested in its Direct3D API to back down from its own brainchild. The company was caught between a rock and a hard place. It could not back off promoting Direct3D for games because that would mean giving up the needed influence to keep game developers developing for Windows exclusively. Consumers like to play games, and keeping a hold on the consumer OS market meant keeping a hold on the number-one consumer application category. Likewise, Microsoft could not back off supporting OpenGL for the workstation market because that would mean giving up the needed influence to attract developers and applications away from competing operating systems.

Microsoft began to insist that OpenGL was for *precise and exacting* rendering needs and Direct3D was for *real-time* rendering. Official Microsoft literature described OpenGL as being something more like a ray tracer than the real-time rendering API it was designed to be. Why Microsoft wasn't thrown off the ARB for such a misinformation campaign remains under lock and key and a few dozen nondisclosure agreements. SGI took up the task of promoting OpenGL as an alternative to Direct3D, and most developers wanted to be able to choose which technology to use for their games.

Direct3D's Head Start

Before 3D hardware was firmly entrenched, game developers had to use software rendering to

create 3D games. It turned out that Microsoft's Direct3D software renderer was many times faster than its own OpenGL renderer. The reason for this difference, according to Microsoft, was that OpenGL is meant for CAD; unfortunately, game programmers don't know much about CAD, but CAD users don't usually like waiting all afternoon for their drawings to rotate either. Microsoft had assumed that OpenGL would be used only with expensive 3D graphics cards in the CAD industry and had not devoted the resources to creating a fast software implementation. Without hardware acceleration, OpenGL was really useless on Windows for anything other than simple static 3D graphics and visualizations. This had nothing to do with the difference between the OpenGL and Direct3D APIs, but only in how they had been implemented by Microsoft.

Silicon Graphics took up the challenge of demonstrating that the design of the OpenGL API was not the flaw, but rather the implementation. At the 1996 SigGraph conference in New Orleans, SGI demonstrated its own OpenGL implementation for Windows. By porting several Direct3D demos to OpenGL, the company easily showed OpenGL running the same animations at equivalent and better speeds.

In reality, however, both software implementations were too slow for really good games. Game developers could write their own optimized 3D code, take liberal shortcuts that would not impact their game, and get much better performance. What really launched both OpenGL and Direct3D as viable gaming APIs was the proliferation of cheap 3D accelerated hardware for the PC. What happened next is history.

Dirty Pool

Some game developers began to develop OpenGL-enabled titles for the 1997 Christmas season. Microsoft encouraged 3D hardware vendors to develop Direct3D drivers, and if they wanted to do OpenGL for Windows 98, they should use a driver kit that Microsoft provided. This kit used the Mini-Client Driver (MCD), which enabled hardware vendors to easily create OpenGL hardware drivers for Windows 98. In response to SGI's OpenGL implementation, an embarrassed Microsoft spent a lot of time tuning its own implementation, and the MCD allowed vendors to tap into this code for everything but the actual drawing commands, which were handled by the graphics card. However, Microsoft was still insisting that OpenGL was not suitable for games development, and the MCD was meant to provide a ready route to the blossoming PC CAD market. Nearly all major PC 3D vendors had MCD-based drivers to demonstrate privately at the 1997 Computer Game Developers Conference. Most were quiet about this fact because it was known that hardware vendors who strongly supported OpenGL for games had *difficulty* getting needed support for their Direct3D efforts. Because most games were being developed with Direct 3D, this would be market suicide.

In the summer of 1997, Microsoft announced that it would not be licensing the MCD code beyond the development stage and vendors would not be allowed to release their drivers for Windows 98. They could, however, ship MCD-based drivers for Windows NT, the workstation platform where OpenGL belonged. As a result, software vendors who devoted time to OpenGL versions of their games could not ship their titles with OpenGL support for Christmas, hardware vendors were left without shippable OpenGL drivers, and Direct3D conveniently got a year's head start on OpenGL as a hardware API standard for games. Meanwhile, Microsoft restated that OpenGL was for non-real-time NT-based workstation applications and Direct 3D was for Windows 98 consumer games.

Fortunately, this situation did not last too long. Silicon Graphics released its own OpenGL driver kit, based on its speedy software implementation for Windows. SGI's driver kit used a much more complex driver mechanism than the MCD, called the Installable Client Driver (ICD). Microsoft had discouraged consumer hardware vendors from starting with the ICD because the MCD would be so much easier for them to use (this was before Microsoft pulled the rug out from under them). SGI's kit, however, had an easier-to-use interface that made developing drivers similar to using the simpler MCD model, and it even improved driver performance.

As hardware drivers for OpenGL began to show up for Windows 98, game companies once again seriously began looking at using OpenGL for game development. Having won its head start, and now unable to further halt the advancement of OpenGL's use for consumer applications, Microsoft relented and again agreed to support OpenGL for Windows 98. This time, however, SGI had to drop all marketing efforts aimed at evangelizing OpenGL toward game developers. Instead, the

two companies would work together on a new *joint* API called Fahrenheit. Fahrenheit would incorporate the best of Direct3D and OpenGL for a new unified 3D API (available only on Windows and SGI hardware). At the time, SGI was losing the battle with NT for workstation sales, so it relented. Simultaneously, the company released a new SGI-branded NT workstation, with Microsoft's full endorsement. OpenGL had lost its greatest supporter for the consumer PC platform.

The Future of OpenGL

Many in the industry saw the Fahrenheit agreement as the beginning of the end for OpenGL. But a funny thing happened on the way to oblivion, and without SGI, OpenGL began to take on a life of its own. When OpenGL was again widely available on consumer hardware, developers didn't really need SGI or anyone else touting the virtues of OpenGL. OpenGL was easy to use and had been around for years. This meant there was an abundance of documentation (including the first edition of this book), sample programs, SigGraph papers, and so on. OpenGL began to flourish.

As more developers began to use OpenGL, it became clear who was really in charge of the industry: the developers. The more applications that shipped with OpenGL support, the more pressure mounted on hardware vendors to produce better OpenGL hardware and high-quality drivers. Consumers don't really care about API technology. They just want software that works, and they will buy whatever graphics card runs their favorite game the best. Developers care about time to market, portability, and code reuse. (Go ahead. Try to recompile that old Direct3D 4.0 program. I dare you!) Using OpenGL enabled many developers to meet customer demand better, and in the end it's the customers who pay the bills.

As time passed, Fahrenheit fell solely into Microsoft's hands and was eventually discontinued altogether. One can only speculate whether that was Microsoft's intent from the beginning. Direct3D has evolved further to include more and more OpenGL features, functionality, and ease of use. OpenGL's popularity has continued to grow as an alternative to Windows-specific rendering technology and is now widely supported across all major operating systems and hardware devices. Even cell phones with 3D graphics technology support a subset of OpenGL, called OpenGL ES. Today, all new 3D accelerated graphics cards for the PC ship with both OpenGL and Direct3D drivers. This is largely due to the fact that many developers continue to prefer OpenGL for new development. OpenGL today is widely recognized and accepted as an industry standard API for real-time 3D graphics.

Developers have continued to be attracted to OpenGL, and despite any political pressures, hardware vendors must satisfy the developers who make software that runs on their hardware. Ultimately, consumer dollars determine what standard survives, and developers who use OpenGL are turning out better games and applications, on more platforms, and in less time than their competitors. Only a few years ago, game developers were creating games with Microsoft's Direct 3D first because that was the only available API with a hardware driver model under consumer Windows—and then porting to OpenGL occasionally so that the same games ran under Windows NT (which didn't support Direct3D). Today, many game and software companies are creating OpenGL versions first and then porting to other platforms such as the Macintosh. It turns out that competitive advantage is more profitable than political alliances.

This momentum will carry OpenGL into the foreseeable future as the API of choice for a wide range of applications and hardware platforms. All this also makes OpenGL well positioned to take advantage of future 3D graphics innovations. Because of OpenGL's extension mechanism, vendors can expose new hardware features without waiting on either the ARB or Microsoft, and cutting-edge developers can exploit them as soon as updated drivers are available. With the addition of the OpenGL shading language (see [Part III](#) of this book), OpenGL has shown its continuing adaptability to meet the challenge of an evolving 3D graphics programming pipeline. Finally, OpenGL is a specification that has shown that it can be applied to a wide variety of programming paradigms. From C/C++ to Java and Visual Basic, even newer languages such as C# are now being used to create PC games using OpenGL. OpenGL is here to stay.

How Does OpenGL Work?

OpenGL is a procedural rather than a descriptive graphics API. Instead of describing the scene and how it should appear, the programmer actually prescribes the steps necessary to achieve a certain appearance or effect. These "steps" involve calls to the many OpenGL commands. These commands are used to draw graphics primitives such as points, lines, and polygons in three dimensions. In addition, OpenGL supports lighting and shading, texture mapping, blending, transparency, animation, and many other special effects and capabilities.

OpenGL does not include any functions for window management, user interaction, or file I/O. Each host environment (such as Microsoft Windows) has its own functions for this purpose and is responsible for implementing some means of handing over to OpenGL the drawing control of a window.

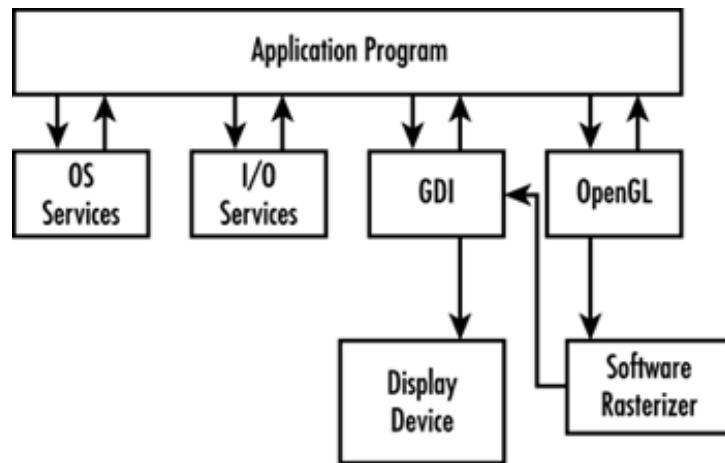
There is no "OpenGL file format" for models or virtual environments. Programmers construct these environments to suit their own high-level needs and then carefully program them using the lower-level OpenGL commands.

Generic Implementations

As mentioned previously, a generic implementation is a software implementation. Hardware implementations are created for a specific hardware device, such as a graphics card or image generator. A generic implementation can technically run just about anywhere as long as the system can display the generated graphics image.

Figure 2.1 shows the typical place that OpenGL and a generic implementation occupy when an application is running. The typical program calls many functions, some of which the programmer creates and some of which are provided by the operating system or the programming language's runtime library. Windows applications wanting to create output onscreen usually call a Windows API called the *Graphics Device Interface (GDI)*. The GDI contains methods that allow you to write text in a window, draw simple 2D lines, and so on.

Figure 2.1. OpenGL's place in a typical application program.



Usually, graphics card vendors supply a hardware driver that GDI interfaces with to create output on your monitor. A software implementation of OpenGL takes graphics requests from an application and constructs (rasterizes) a color image of the 3D graphics. It then supplies this image to the GDI for display on the monitor. On other operating systems, the process is reasonably equivalent, but you replace the GDI with that operating system's native display services.

OpenGL has a couple of common generic implementations. Microsoft has shipped its software implementation with every version of Windows NT since version 3.5 and Windows 95 (Service

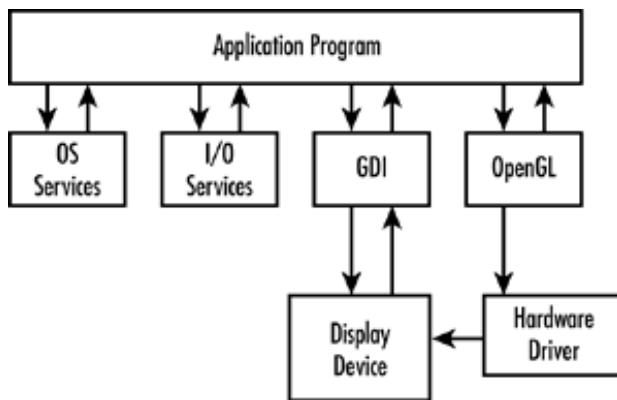
Release 2 and later). Windows 2000 and XP also contain support for OpenGL.

SGI released a software implementation of OpenGL for Windows that greatly outperformed Microsoft's implementation. This implementation is not officially supported but is still occasionally used by developers. MESA 3D is another "unofficial" OpenGL software implementation that is widely supported in the open source community. Mesa 3D is not an OpenGL license, so it is an "OpenGL work-alike" rather than an official implementation. In any respect other than legal, you can essentially consider it to be an OpenGL implementation nonetheless. The Mesa contributors even make a good attempt to pass the OpenGL conformance tests.

Hardware Implementations

A hardware implementation of OpenGL usually takes the form of a graphics card driver. [Figure 2.2](#) shows its relationship to the application similarly to the way [Figure 2.1](#) did for software implementations. Note that OpenGL API calls are passed to a hardware driver. This driver does not pass its output to the Windows GDI for display; the driver interfaces directly with the graphics display hardware.

Figure 2.2. Hardware-accelerated OpenGL's place in a typical application program.

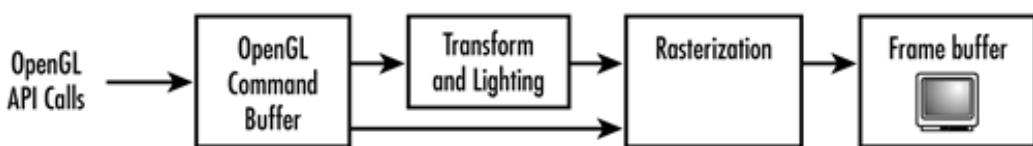


A hardware implementation is often referred to as an *accelerated implementation* because hardware-assisted 3D graphics usually far outperform software-only implementations. What isn't shown in [Figure 2.2](#) is that sometimes part of the OpenGL functionality is still implemented in software as part of the driver, and other features and functionality can be passed directly to the hardware. This idea brings us to our next topic: the OpenGL pipeline.

The Pipeline

The word *pipeline* is used to describe a process that can take two or more distinct stages or steps. [Figure 2.3](#) shows a simplified version of the OpenGL pipeline. As an application makes OpenGL API function calls, the commands are placed in a command buffer. This buffer eventually fills with commands, vertex data, texture data, and so on. When the buffer is flushed, either programmatically or by the driver's design, the commands and data are passed to the next stage in the pipeline.

Figure 2.3. A simplified version of the OpenGL pipeline.



Vertex data is usually transformed and lit initially. In subsequent chapters, you'll find out more about what this means. For now, you can consider "Transform and Lighting" to be a

mathematically intensive stage where points used to describe an object's geometry are recalculated for the given object's location and orientation. Lighting calculations are performed as well to indicate how brightly the colors should be at each vertex.

When this stage is complete, the data is fed to the rasterization portion of the pipeline. The rasterizer actually creates the color image from the geometric, color, and texture data. The image is then placed in the *frame buffer*. The frame buffer is the memory of the graphics display device, which means the image is displayed on your screen.

This diagram provides a simplistic view of the OpenGL pipeline, but it is sufficient for your current understanding of 3D graphics rendering. At a high level, this view is accurate, so we aren't compromising your understanding, but at a low level, many more boxes appear inside each box shown here. There are also some exceptions, such as the arrow in the figure indicating that some commands skip the Transform and Lighting (T&L) stage altogether (such as displaying raw image data on the screen).

Early OpenGL hardware accelerators were nothing more than fast rasterizers. They accelerated only the rasterization portion of the pipeline. The host system's CPU did transform and lighting in a software implementation of that portion of the pipeline. Higher-end (more expensive) accelerators had T&L on the graphics accelerator. This arrangement put more of the OpenGL pipeline in hardware and thus provided for higher performance.

Even most low-end consumer hardware today has the T&L stage in hardware. The net effect of this arrangement is that higher detailed models and more complex graphics are possible at real-time rendering rates on inexpensive consumer hardware. Games and applications developers can capitalize on this effect, yielding far more detailed and visually rich environments.

OpenGL: An API, Not a Language

For the most part, OpenGL is not a programming language; it is an Application Programming Interface (API). Whenever we say that a program is OpenGL-based or an OpenGL application, we mean that it was written in some programming language (such as C or C++) that makes calls to one or more of the OpenGL libraries. We are not saying that the program uses OpenGL exclusively to do drawing. It might combine the best features of two different graphics packages. Or it might use OpenGL for only a few specific tasks and environment-specific graphics (such as the Windows GDI) for others. The only exception to this rule of thumb is, of course, the *OpenGL Shading Language*, which will be covered later in this book.

As an API, the OpenGL library follows the C calling convention, and in this book, the sample programs are written in C. C++ programs can easily access C functions and APIs in the same manner as C, with only some minor considerations. Most C++ programmers can still program in C, and we don't want to exclude anyone or place any additional burden on the reader (such as having to get used to C++ syntax). Other programming languages—such as Visual Basic—that can call functions in C libraries can also make use of OpenGL, and OpenGL bindings are available for many other programming languages. Using OpenGL from these other languages is, however, outside the scope of this book and can be troublesome. To keep things simple and easily portable, we'll stick with C for our examples.

Libraries and Headers

Although OpenGL is a "standard" programming library, this library has many implementations. Microsoft Windows ships with support for OpenGL as a software renderer. This means that when a program written to use OpenGL makes OpenGL function calls, the Microsoft implementation performs the 3D rendering functions, and you see the results in your application window. The actual Microsoft software implementation is in the `opengl32.dll` dynamic link library, located in the Windows system directory. On most platforms, the OpenGL library is accompanied by the OpenGL utility library (GLU), which on Windows is in `glu32.dll`, also located in the system directory. The utility library is a set of utility functions that perform common (but sometimes complex) tasks, such as special matrix calculations, or provide support for common types of curves and surfaces.

The steps for setting up your compiler tools to link to the correct OpenGL libraries vary from tool to tool and from platform to platform. You can find some step-by-step instructions for Windows, Macintosh, and Linux in the platform-specific chapters in [Part II](#) of this book.

Prototypes for all OpenGL functions, types, and macros are contained (by convention) in the header file `gl.h`. Microsoft programming tools ship with this file, and so do most other programming environments for Windows or other platforms (at least those that natively support OpenGL). The utility library functions are prototyped in a different file, `glu.h`. These files are usually located in a special directory in your *include* path. For example, the following code shows the typical initial header inclusions for a typical Windows program that uses OpenGL:

```
#include<windows.h>
#include<gl/gl.h>
#include<gl/glu.h>
```

For the purposes of this book, we have created our own header file `OpenGL.h`, which has macros defined for the various platforms and operating systems to include the correct headers and libraries for use with OpenGL. All the sample programs include this source file.

API Specifics

OpenGL was designed by some clever people who had a lot of experience designing graphics programming APIs. They applied some standard rules to the way functions were named and variables were declared. The API is simple and clean and easy for vendors to extend. OpenGL tries to avoid as much *policy* as possible. Policy refers to assumptions that the designers make about how programmers will use the API. Examples of policies are assuming that you always specify vertex data as floating-point values, assuming that fog is always enabled before any rendering occurs, or assuming that all objects in a scene are affected by the same lighting parameters. To do so would eliminate many of the popular rendering techniques that have developed over time.

Data Types

To make it easier to port OpenGL code from one platform to another, OpenGL defines its own data types. These data types map to normal C data types that you can use instead, if you want. The various compilers and environments, however, have their own rules for the size and memory layout of various C variables. By using the OpenGL defined variable types, you can insulate your code from these types of changes.

[Table 2.1](#) lists the OpenGL data types, their corresponding C data types under the 32-bit Windows environments (Win32), and the appropriate suffix for literals. In this book, we use the suffixes for all literal values. You will see later that these suffixes are also used in many OpenGL function names.

Table 2.1. OpenGL Variable Types and Corresponding C Data Types

OpenGL Data Type	Internal Representation	Defined as C Type	C Literal Suffix
<code>GLbyte</code>	8-bit integer	<code>signed char</code>	<code>b</code>
<code>GLshort</code>	16-bit integer	<code>short</code>	<code>s</code>
<code>GLint, GLsizei</code>	32-bit integer	<code>long</code>	<code>l</code>
<code>GLfloat, GLclampf</code>	32-bit floating point	<code>float</code>	<code>f</code>

<code>GLdouble, GLclampd</code>	64-bit floating point	<code>double</code>	<code>d</code>
<code>GLubyte, GLboolean</code>	8-bit unsigned integer	<code>unsigned char</code>	<code>ub</code>
<code>GLushort</code>	16-bit unsigned integer	<code>unsigned short</code>	<code>us</code>
<code>GLuint, GLenum,</code> <code>GLbitfield</code>	32-bit unsigned integer	<code>unsigned long</code>	<code>ui</code>

All data types start with a `GL` to denote OpenGL. Most are followed by their corresponding C data types (`byte`, `short`, `int`, `float`, and so on). Some have a `u` first to denote an unsigned data type, such as `ubyte` to denote an unsigned byte. For some uses, a more descriptive name is given, such as `size` to denote a value of length or depth. For example, `GLsizei` is an OpenGL variable denoting a size parameter that is represented by an integer. The `clamp` designation is a hint that the value is expected to be "clamped" to the range 0.0–1.0. The `GLboolean` variables are used to indicate true and false conditions, `GLenum` for enumerated variables, and `GLbitfield` for variables that contain binary bit fields.

Pointers and arrays are not given any special consideration. An array of 10 `GLshort` variables is simply declared as

```
GLshort shorts[10];
```

and an array of 10 pointers to `GLdouble` variables is declared with

```
GLdouble *doubles[10];
```

Some other pointer object types are used for NURBS and quadrics. They require more explanation and are covered in later chapters.

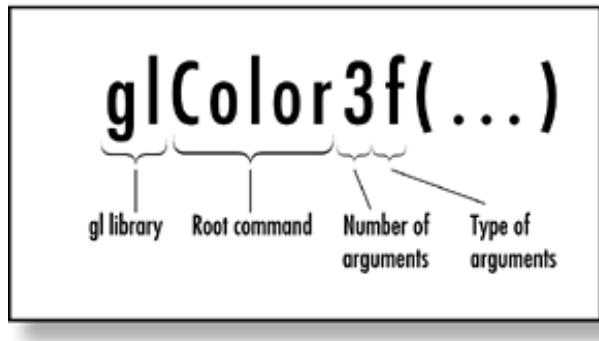
Function-Naming Conventions

Most OpenGL functions follow a naming convention that tells you which library the function is from and often how many and what types of arguments the function takes. All functions have a root that represents the function's corresponding OpenGL command. For example, `glColor3f` has the root `Color`. The `gl` prefix represents the `gl` library, and the `3f` suffix means the function takes three floating-point arguments. All OpenGL functions take the following format:

```
<Library prefix><Root command><Optional argument count><Optional argument type>
```

Figure 2.4 illustrates the parts of an OpenGL function. This sample function with the suffix `3f` takes three floating-point arguments. Other variations take three integers (`glColor3i`), three doubles (`glColor3d`), and so forth. This convention of adding the number and types of arguments (see Table 2.1) to the end of OpenGL functions makes it easy to remember the argument list without having to look it up. Some versions of `glColor` take four arguments to specify an alpha component (transparency), as well.

Figure 2.4. A dissected OpenGL function.



In the reference sections of this book, these "families" of functions are listed by their library prefix and root. All the variations of `glColor` (`glColor3f`, `glColor4f`, `glColor3i`, and so on) are listed under a single entry—`glColor`.

Many C/C++ compilers for Windows assume that any floating-point literal value is of type `double` unless explicitly told otherwise via the suffix mechanism. When using literals for floating-point arguments, if you don't specify that these arguments are of type `float` instead of `double`, the compiler issues a warning while compiling because it detects that you are passing a `double` to a function defined to accept only `floats`, resulting in a possible loss of precision. As OpenGL programs grow, these warnings quickly number in the hundreds and make it difficult to find any real syntax errors. You can turn off these warnings using the appropriate compiler options, but we advise against doing so. It's better to write clean, portable code the first time. So, clean up those warning messages by cleaning up the code (in this case, by explicitly using the `float` type)—not by disabling potentially useful warnings.

Additionally, you might be tempted to use the functions that accept double-precision floating-point arguments rather than go to all the bother of specifying your literals as floats. However, OpenGL uses floats internally, and using anything other than the single-precision floating-point functions adds a performance bottleneck because the values are converted to floats anyway before being processed by OpenGL—not to mention that every `double` takes up twice as much memory as a `float`. For a program with a lot of numbers "floating" around, these performance hits can add up pretty fast!

Platform Independence

OpenGL is a powerful and sophisticated API for creating 3D graphics, with more than 300 commands that cover everything from setting material colors and reflective properties to doing rotations and complex coordinate transformations. You might be surprised that OpenGL does not have a single function or command relating to window or screen management. In addition, there are no functions for keyboard input or mouse interaction. Consider, however, that one of the OpenGL designers' primary goals was platform independence. You create and open a window differently under the various platforms. Even if OpenGL did have a command for opening a window, would you use it, or would you use the operating system's own built-in API call?

Another platform issue is the handling of keyboard and mouse input events under the different operating systems and environments. If every environment handled these events the same, we would have only one environment to worry about and no need for an "open" API. This is not the case, however, and it probably won't happen within our brief lifetimes! So OpenGL's platform independence comes at the cost of having no OS and GUI functions.

Using GLUT

In the beginning, there was AUX, the OpenGL auxiliary library. The AUX library was created to facilitate the learning and writing of OpenGL programs without the programmer's being distracted by the minutiae of any particular environment, be it UNIX, Windows, or whatever. You wouldn't write "final" code when using AUX; it was more of a preliminary staging ground for testing your

ideas. A lack of basic GUI features limited the library's use for building useful applications.

Only a few years ago, most OpenGL samples circulating the Web (and OpenGL book samples!) were written using the AUX library. The Windows implementation of the AUX library was buggy and prone to cause frequent crashes. The lack of any GUI features whatsoever was another drawback in the modern GUI-oriented world.

AUX has since been replaced by the GLUT library for cross-platform programming examples and demonstrations. GLUT stands for *OpenGL utility toolkit* (not to be confused with the standard GLU—OpenGL utility library). Mark Kilgard, while at SGI, wrote GLUT as a more capable replacement for the AUX library and included some GUI features to at least make sample programs more usable under X Windows. This replacement includes using pop-up menus, managing other windows, and even providing joystick support. GLUT is not public domain, but it is free and free to redistribute. The latest GLUT distribution available at the time of printing is on the CD that accompanies this book.

For most of this book, we use GLUT as our program framework. This decision serves two purposes. The first is that it makes most of the book accessible to a wider audience than only Windows programmers. With a little effort, experienced Linux or Mac programmers should be able to set up GLUT for their programming environments and follow along most of the examples in this book. Platform-specific details are included in the chapters in Part II of this book.

The second point is that using GLUT eliminates the need to know and understand basic GUI programming on any specific platform. Although we explain the general concepts, we do not claim to write a book about GUI programming, but about OpenGL. Using GLUT for the basic coverage of the API, we make life a bit easier for Windows/Mac/Linux novices as well.

It's unlikely that all the functionality of a commercial application will be embodied entirely in the code used to draw in 3D, so you can't rely entirely on the GLUT library for everything. Nevertheless, the GLUT library excels in its role for learning and demonstration exercises. Even for

Animation with OpenGL and GLUT

So far, we've discussed the basics of using the GLUT library for creating a window and using OpenGL commands for the actual drawing. You will often want to move or rotate your images and scenes, creating an animated effect. Let's take the previous example, which draws a square, and make the square bounce off the sides of the window. You could create a loop that continually changes your object's coordinates before calling the `RenderScene` function. This would cause the square to appear to move around within the window.

The GLUT library enables you to register a callback function that makes it easier to set up a simple animated sequence. This function, `glutTimerFunc`, takes the name of a function to call and the amount of time to wait before calling the function:

```
void glutTimerFunc(unsigned int msecs, void (*func)(int value), int value);
```

This code sets up GLUT to wait `msecs` milliseconds before calling the function `func`. You can pass a user-defined value in the `value` parameter. The function called by the timer has the following prototype:

```
void TimerFunction(int value);
```

Unlike the Windows timer, this function is fired only once. To effect a continuous animation, you must reset the timer again in the timer function.

In our `GLRect` program, we can change the hard-coded values for the location of our rectangle to variables and then constantly modify those variables in the timer function. This causes the rectangle to appear to move across the window. Let's look at an example of this kind of animation.

In [Listing 2.3](#), we modify [Listing 2.2](#) to bounce around the square off the inside borders of the window. We need to keep track of the position and size of the rectangle as we go along and account for any changes in window size.

Listing 2.3. Animated Bouncing Square

```
#include <OpenGL.h>
// Initial square position and size
GLfloat x1 = 0.0f;
GLfloat y1 = 0.0f;
GLfloat rsize = 25;
// Step size in x and y directions
// (number of pixels to move each time)
GLfloat xstep = 1.0f;
GLfloat ystep = 1.0f;
// Keep track of windows changing width and height
GLfloat windowHeight;
GLfloat windowWidth;
///////////////////////////////
// Called to draw scene
void RenderScene(void)
{
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT);
    // Set current drawing color to red
    //          R      G      B
    glColor3f(1.0f, 0.0f, 0.0f);
    // Draw a filled rectangle with current color
    glRectf(x1, y1, x1 + rsize, y1 - rsize);
    // Flush drawing commands and swap
    glutSwapBuffers();
}
///////////////////////////////
// Called by GLUT library when idle (window not being
// resized or moved)
void TimerFunction(int value)
{
    // Reverse direction when you reach left or right edge
    if(x1 > windowWidth-rsize || x1 < -windowWidth)
        xstep = -xstep;
    // Reverse direction when you reach top or bottom edge
    if(y1 > windowHeight || y1 < -windowHeight + rsize)
        ystep = -ystep;
    // Actually move the square
    x1 += xstep;
    y1 += ystep;
    // Check bounds. This is in case the window is made
    // smaller while the rectangle is bouncing and the
    // rectangle suddenly finds itself outside the new
    // clipping volume
    if(x1 > (windowWidth-rsize + xstep))
        x1 = windowWidth-rsize-1;
    else if(x1 < -(windowWidth + xstep))
        x1 = -windowWidth -1;
    if(y1 > (windowHeight + ystep))
        y1 = windowHeight-1;
    else if(y1 < -(windowHeight - rsize + ystep))
        y1 = -windowHeight + rsize -1;
    // Redraw the scene with new coordinates
    glutPostRedisplay();
    glutTimerFunc(33,TimerFunction, 1);
}
```

```

///////////
// Setup the rendering state
void SetupRC(void)
{
    // Set clear color to blue
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
}
///////////
// Called by GLUT library when the window has changed size
void ChangeSize(GLsizei w, GLsizei h)
{
    GLfloat aspectRatio;
    // Prevent a divide by zero
    if(h == 0)
        h = 1;
    // Set Viewport to window dimensions
    glViewport(0, 0, w, h);
    // Reset coordinate system
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    // Establish clipping volume (left, right, bottom, top, near, far)
    aspectRatio = (GLfloat)w / (GLfloat)h;
    if (w <= h)
    {
        windowHeight = 100;
        windowHeight = 100 / aspectRatio;
        glOrtho (-100.0, 100.0, -windowHeight, windowHeight, 1.0, -1.0);
    }
    else
    {
        windowHeight = 100 * aspectRatio;
        windowHeight = 100;
        glOrtho (-windowWidth, windowHeight, -100.0, 100.0, 1.0, -1.0);
    }
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
///////////
// Main program entry point
void main(void)
{
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutCreateWindow("Bounce");
    glutDisplayFunc(RenderScene);
    glutReshapeFunc(ChangeSize);
    glutTimerFunc(33, TimerFunction, 1);
    SetupRC();
    glutMainLoop();
}

```

Double-Buffering

One of the most important features of any graphics packages is support for *double buffering*. This feature allows you to execute your drawing code while rendering to an offscreen buffer. Then a swap command places your drawing onscreen instantly.

Double buffering can serve two purposes. The first is that some complex drawings might take a long time to draw, and you might not want each step of the image composition to be visible. Using double buffering, you can compose an image and display it only after it is complete. The user never sees a partial image; only after the entire image is ready is it shown onscreen.

A second use for double buffering is animation. Each frame is drawn in the offscreen buffer and then swapped quickly to the screen when ready. The GLUT library supports double-buffered windows. In [Listing 2.3](#) note the following line:

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
```

We have changed `GLUT_SINGLE` to `GLUT_DOUBLE`. This change causes all the drawing code to render in an offscreen buffer.

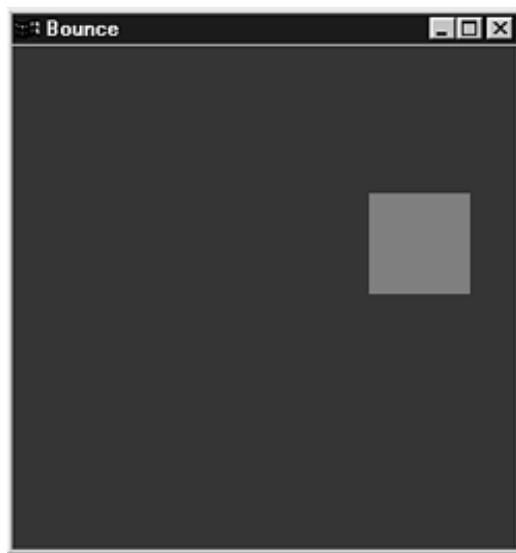
Next, we also changed the end of the `RenderScene` function:

```
. . .
    // Flush drawing commands and swap
    glutSwapBuffers();
}
```

No longer are we calling `glFlush`. This function is no longer needed because when we perform a buffer swap, we are implicitly performing a flush operation.

These changes cause a smoothly animated bouncing rectangle, shown in [Figure 2.11](#). The function `glutSwapBuffers` still performs the flush, even if you are running in single-buffered mode. Simply change `GLUT_DOUBLE` back to `GLUT_SINGLE` in the bounce sample to see the animation without double buffering. As you'll see, the rectangle constantly blinks and stutters, a very unpleasant and poor animation with single buffering.

Figure 2.11. Follow the bouncing square.



The GLUT library is a reasonably complete framework for creating sophisticated sample programs and perhaps even full-fledged commercial applications (assuming you do not need to use OS-specific or GUI features). It is not the purpose of this book to explore GLUT in all its glory and splendor, however. Here and in the reference section to come, we restrict ourselves to the small subset of GLUT needed to demonstrate the various features of OpenGL.

The OpenGL State Machine

Drawing 3D graphics is a complicated affair. In the chapters ahead, we will cover many OpenGL functions. For a given piece of geometry, many things can affect how it is drawn. Is a light shining on it? What are the properties of the light? What are the properties of the material? Which, if any, texture should be applied? The list could go on and on.

We call this collection of variables the *state* of the pipeline. A state machine is an abstract model of a collection of state variables, all of which can have various values, be turned on or off, and so on. It simply is not practical to specify all the state variables whenever we try to draw something in OpenGL. Instead, OpenGL employs a state model, or state machine, to keep track of all the OpenGL state variables. When a state value is set, it remains set until some other function changes it. Many states are simply on or off. Lighting, for example (see [Chapter 11](#)), is either turned on or off. Geometry drawn without lighting is drawn without any lighting calculations being applied to the colors set for the geometry. Any geometry drawn *after* lighting is turned back on is then drawn with the lighting calculations applied.

To turn these types of state variables on and off, you use the following OpenGL function:

```
void glEnable(GLenum capability);
```

You turn the variable back off with the corresponding function:

```
void glDisable(GLenum capability);
```

For the case of lighting, for instance, you can turn it on by using the following:

```
glEnable(GL_LIGHTING);
```

And you turn it back off with this function:

```
glDisable(GL_LIGHTING);
```

If you want to test a state variable to see whether it is enabled, OpenGL again has a convenient mechanism:

```
GLboolean glIsEnabled(GLenum capability);
```

Not all state variables, however, are simply on or off. Many of the OpenGL functions yet to come set up values that "stick" until changed. You can query what these values are at any time as well. A set of query functions allows you to query the values of booleans, integers, floats, and double variables. These four functions are prototyped thus:

```
void glGetBooleanv(GLenum pname, GLboolean *params);
void glGetDoublev(GLenum pname, GLdouble *params);
void glGetFloatv(GLenum pname, GLfloat *params);
void glGetIntegerv(GLenum pname, GLint *params);
```

Each function returns a single value or a whole array of values, storing the results at the address you supply. The various parameters are documented in the reference section (there are a lot of them!). Most may not make much sense to you right away, but as you progress through the book, you will begin to appreciate the power and simplicity of the OpenGL state machine.

Saving and Restoring States

OpenGL also has a convenient mechanism for saving a whole range of state values and restoring them later. The *stack* is a convenient data structure that allows values to be *pushed* on the stack

(saved) and *popped* off the stack later to retrieve them. Items are popped off in the opposite order in which they were pushed on the stack. We call this a Last In First Out (*LIFO*) data structure. It's an easy way to just say, "Hey, please save this" (push it on the stack), and then a little later say, "Give me what I just saved" (pop it off the stack). You'll see that the concept of the stack plays a very important role in matrix manipulation when you get to [Chapter 4](#).

A single OpenGL state value or a whole range of related state values can be pushed on the attribute stack with the following command:

```
void glPushAttrib(GLbitfield mask);
```

Values are correspondingly retrieved with this command:

```
void glPopAttrib(GLbitfield mask);
```

Note that the argument to these functions is a bitfield. This means that you use a bitwise mask, which allows you to perform a bitwise **OR** (in C using the `|` operator) of multiple state values with a single function call. For example, you could save the lighting and texturing state with a single call like this:

```
glPushAttrib(GL_TEXTURE_BIT | GL_LIGHTING_BIT);
```

A complete list of all the OpenGL state values that can be saved and restored with these functions is located in the reference sections.

OpenGL Errors

In any project, you want to write robust and well-behaved programs that respond politely to their users and have some amount of flexibility. Graphical programs that use OpenGL are no exception, and if you want your programs to run smoothly, you need to account for errors and unexpected circumstances. OpenGL provides a useful mechanism for you to perform an occasional sanity check in your code. This capability can be important because, from the code's standpoint, it's not really possible to tell whether the output was the Space Station Freedom or the Space Station Melted Crayon!

When Bad Things Happen to Good Code

Internally, OpenGL maintains a set of six error flags. Each flag represents a different type of error. Whenever one of these errors occurs, the corresponding flag is set. To see whether any of these flags are set, call `glGetError`:

```
Glenum glGetError(void);
```

The `glGetError` function returns one of the values listed in [Table 2.3](#). The GLU library defines three errors of its own, but these errors map exactly to two flags already present. If more than one of these flags is set, `glGetError` still returns only one distinct value. This value is then cleared when `glGetError` is called, and `glGetError` again will return either another error flag or `GL_NO_ERROR`. Usually, you want to call `glGetError` in a loop that continues checking for error flags until the return value is `GL_NO_ERROR`.

You can use another function in the GLU library, `gluErrorString`, to get a string describing the error flag:

```
const GLubyte* gluErrorString(GLenum errorCode);
```

This function takes as its only argument the error flag (returned from `glGetError`) and returns a

static string describing that error. For example, the error flag `GL_INVALID_ENUM` returns this string:

```
invalid enumerant
```

Table 2.3. OpenGL Error Codes

Error Code	Description
<code>GL_INVALID_ENUM</code>	The enum argument is out of range.
<code>GL_INVALID_VALUE</code>	The numeric argument is out of range.
<code>GL_INVALID_OPERATION</code>	The operation is illegal in its current state.
<code>GL_STACK_OVERFLOW</code>	The command would cause a stack overflow.
<code>GL_STACK_UNDERFLOW</code>	The command would cause a stack underflow.
<code>GL_OUT_OF_MEMORY</code>	Not enough memory is left to execute the command.
<code>GL_TABLE_TOO_LARGE</code>	The specified table is too large.
<code>GL_NO_ERROR</code>	No error has occurred.

You can take some peace of mind from the assurance that if an error is caused by an invalid call to OpenGL, the command or function call is ignored. The only exceptions to this are the functions (described in later chapters) that take pointers to memory (that may cause a program to crash if the pointer is invalid) and the out of memory condition. If you receive an out of memory error, all bets are off as to what you might see onscreen!

Identifying the Version

As mentioned previously, sometimes you want to take advantage of a known behavior in a particular implementation. If you know for a fact that you are running on a particular vendor's graphics card, you may rely on some known performance characteristics to enhance your program. You may also want to enforce some minimum version number for particular vendors' drivers. What you need is a way to query OpenGL for the vendor and version number of the rendering engine (the OpenGL driver). Both the GL library and GLU library can return version and vendor-specific information about themselves.

For the GL library, you can call `glGetString`:

```
const GLubyte *glGetString(GLenum name);
```

This function returns a static string describing the requested aspect of the GL library. The valid parameter values are listed under `glGetString` in the reference section, along with the aspect of the GL library they represent.

The GLU library has a corresponding function, `gluGetString`:

```
const GLubyte *gluGetString(GLenum name);
```

It returns a string describing the requested aspect of the GLU library. The valid parameters are listed under `gluGetString` in the reference section, along with the aspect of the GLU library they represent.

Getting a Clue with `glHint`

There is more than one way to skin a cat: so goes the old saying. The same is true with 3D graphics algorithms. Often a trade-off must be made for the sake of performance, or perhaps if visual fidelity is the most important issue, performance is less of a consideration. Often an OpenGL implementation may contain two ways of performing a given task—a fast way that compromises quality slightly and a slower way that improves visual quality. The function `glHint` allows you to specify certain preferences of quality or speed for different types of operations. The function is defined as follows:

```
void glHint(GLenum target, GLenum mode);
```

The `target` parameter allows you to specify types of behavior you want to modify. These values, listed under `glHint` in the reference section, include hints for fog and antialiasing accuracy. The `mode` parameter tells OpenGL what you care most about—faster render time and nicest output, for instance—or that you don't care (the only way to get back to the default behavior). Be warned, however, that all implementations are not required to honor calls into `glHint`; it's the only function in OpenGL whose behavior is intended to be entirely vendor-specific.

Using Extensions

With OpenGL being a "standard" API, you might think that hardware vendors are able to compete only on the basis of performance and perhaps visual quality. However, the field of 3D graphics is very competitive, and hardware vendors are constantly innovating, not just in the areas of performance and quality, but in graphics methodologies and special effects. OpenGL allows vendor innovation through its extension mechanism. This mechanism works in two ways. First, vendors can add new functions to the OpenGL API that developers can use. Second, new tokens or enumerants can be added that will be recognized by existing OpenGL functions such as `glEnable`.

Making use of new enumerants or tokens is simply a matter of adding a vendor-supplied header file to your project. Vendors must register their extensions with the OpenGL ARB, thus keeping one vendor from using a value used by someone else. Conveniently, there is a header file `glext.h` (supplied on the CD-ROM) that includes the most common extensions.

Checking for an Extension

Gone are the days when games would be recompiled for a specific graphics card. You have already seen that you can check for a string identifying the vendor and version of the OpenGL driver. You can also get a string that contains identifiers for all OpenGL extensions supported by the driver. One line of code returns a character array of extension names:

```
const char *szExtensions = glGetString(GL_EXTENSIONS);
```

This string contains the space-delimited names of all extensions supported by the driver. You can then search this string for the identifier of the extension you want to use. For example, you might do a quick search for a Windows-specific extension like this:

```
if (strstr(extensions, "WGL_EXT_swap_control" != NULL))
{
    wglSwapIntervalEXT =
        (PFNWGLSWAPINTERVALEXTPROC)wglGetProcAddress( "wglSwapIntervalEXT" );
    if(wglSwapIntervalEXT != NULL)
        wglSwapIntervalEXT(1);
}
```

If you use this method, you should also make sure that the character following the name of the extension is either a space or a NULL. What if, for example, this extension is superceded by the `WGL_EXT_swap_control2` extension? In this case, the C runtime function `strstr` would still find

the first string, but you may not be able to assume that the second extension behaves exactly like the first. A more robust toolkit function is included in the file `IsExtSupported.c` in the `\common` directory on the CD-ROM:

```
int gltIsExtSupported(const char *extension);
```

This function returns 1 if the named extension is supported or 0 if it is not. The `\common` directory contains a whole set of helper and utility functions for use with OpenGL, and many are used throughout this book. All the functions are prototyped in the file `gltools.h`.

This example also shows how to get a pointer to a new OpenGL function under Windows. The windows function `wglGetProcAddress` returns a pointer to an OpenGL function (extension) name. Getting a pointer to an extension varies from OS to OS and will be dealt with in more detail in [Part II](#) of this book. This Windows-specific extension and the typedef (`PFNWGLSWAPINTERVALEXTPROC`) for the function type is located in the `wglext.h` header file, also included on the CD-ROM. We also discuss this particular important extension in [Chapter 13](#), "Wiggle: OpenGL on Windows."

In the meantime, again the `gltools` library comes to the rescue with the following function:

```
void *gltGetExtensionPointer(const char *szExtensionName);
```

This function provides a platform-independent wrapper that returns a pointer to the named OpenGL extension. The implementation of this function is in the file `GetExtensionPointer.c`, which you will have to add to your projects to make use of this feature.

Who's Extension Is This?

Using OpenGL extensions, you can provide code paths in your code to improve rendering performance and visual quality or even add special effects that are supported only by a particular vendor's hardware. But who owns an extension? That is, which vendor created and supports a given extension? You can usually tell just by looking at the extension name. Each extension has a three-letter prefix that identifies the source of the extension. [Table 2.4](#) provides a sampling of extension identifiers.

Table 2.4. A Sampling of OpenGL Extension Prefixes

Prefix	Vendor
<code>SGI_</code>	Silicon Graphics
<code>ATI_</code>	ATI Technologies
<code>NV_</code>	NVidia
<code>IBM_</code>	IBM
<code>WGL_</code>	Microsoft
<code>EXT_</code>	Cross-Vendor
<code>ARB_</code>	ARB Approved

It is not uncommon for one vendor to support another vendor's extension. For example, some

NVidia extensions are widely popular and supported on ATI hardware. When this happens, the competing vendor must follow the original vendor's specification (details on how the extension is supposed to work). Frequently, everyone agrees that the extension is a good thing to have, and the extension has an **EXT**_ prefix to show that it is (supposed) to be vendor neutral and widely supported across implementations.

Finally, we also have ARB-approved extensions. The specification for these extensions has been reviewed (and argued about) by the OpenGL ARB. These extensions usually signal the final step before some new technique or function finds its way into the core OpenGL specification. We expound upon the idea of the core OpenGL API versus the OpenGL extensions in greater detail in [Part III](#) of this book.

Getting to OpenGL Beyond 1.1 on Windows

Most Windows programmers use the Microsoft development tools, meaning Visual C++ or the newer Visual C++ .NET development environments. The Microsoft software OpenGL implementation for Windows includes only functions and tokens for OpenGL as defined in the OpenGL specification version 1.1. Since that time, we have seen 1.2, 1.3, 1.4, 1.5, and 2.0. This means several OpenGL features are unavailable to users of the OpenGL header files included with Microsoft's development tools. Other OS vendors and tool makers have managed to keep more up to date, however, and the Macintosh OS X and Linux samples should compile with fewer problems.

Nevertheless, OpenGL can at times seem like a moving target, especially where cross-platform compatibility interacts with the later and new enhancements to OpenGL. The header file `glext.h`, included in the `\common` directory, contains constants and function prototypes for most OpenGL functionality past version 1.1 and is already a part of the standard development tools on some platforms. This header is included specifically for Windows developers, whereas Mac developers, for example, will use the `glext.h` headers included with the XCode or Project Workbench development environments.

Between the `glext.h` header file and the `gltGetExtensionPointer` function in `gltools`, it will not be difficult for you to build the samples in this book and run them with a wide variety of compilers and development environments.

Summary

We covered a lot of ground in this chapter. We introduced you to OpenGL, told you a little bit about its history, introduced the OpenGL utility toolkit (GLUT), and presented the fundamentals of writing a program that uses OpenGL. Using GLUT, we showed you the easiest possible way to create a window and draw in it using OpenGL commands. You learned to use the GLUT library to create windows that can be resized, as well as create a simple animation. You were also introduced to the process of using OpenGL to do drawing—composing and selecting colors, clearing the screen, drawing a rectangle, and setting the viewport and clipping volume to scale images to match the window size. We discussed the various OpenGL data types and the headers required to build programs that use OpenGL.

With a little coding finally under your belt, you were ready to dive into some other ideas that you need to be familiar with before you move forward. The OpenGL state machine underlies almost everything you do from here on out, and the extension mechanism will make sure you can access all the OpenGL features supported by your hardware driver, regardless of your development tool. You also learned how to check for OpenGL errors along the way to make sure you aren't making any illegal state changes or rendering commands. With this foundation, you can move forward to the chapters ahead.

Reference

glClearColor

Purpose: Sets the color and alpha values to use for clearing the color buffers.

Include File: `<gl.h>`

Syntax:

```
void glClearColor(GLclampf red, GLclampf green,
  GLclampf blue, GLclampf alpha);
```

Description: This function sets the fill values to be used when clearing the red, green, blue, and alpha buffers (jointly called the color buffer). The values specified are clamped to the range [0.0f, 1.0f].

Parameters:

red `GLclampf`: The red component of the fill value.

green `GLclampf`: The green component of the fill value.

blue `GLclampf`: The blue component of the fill value.

alpha `GLclampf`: The alpha component of the fill value.

Returns: None.

glDisable, glEnable

Purpose: Disables or enables an OpenGL state feature.

Include File: `<GL/gl.h>`

Syntax:

```
void glDisable(GLenum feature);
glEnable
```

Description: `glDisable` disables an OpenGL drawing feature, and `glEnable` enables an OpenGL drawing feature.

Parameters:

feature `GLenum`: The feature to disable or enable. A complete list of states is given in the OpenGL specification and grows regularly with new revisions from the ARB and new OpenGL extensions from hardware vendors. For illustration, [Table 2.5](#) lists a small sampling of states that are turned on and off.

**Table 2.5. Features Enabled/Disabled by `glEnable`/
`glDisable`**

Feature	Description
<code>GL_BLEND</code>	Color blending
<code>GL_CULL_FACE</code>	Polygon culling
<code>GL_DEPTH_TEST</code>	Depth test
<code>GL_DITHER</code>	Dithering
<code>GL_FOG</code>	OpenGL fog mode
<code>GL_LIGHTING</code>	OpenGL lighting
<code>GL_LIGHTx</code>	xth OpenGL light (minimum: 8)
<code>GL_POINT_SMOOTH</code>	Point antialiasing
<code>GL_LINE_SMOOTH</code>	Line antialiasing
<code>GL_LINE_STIPPLE</code>	Line stippling
<code>GL_POLYGON_SMOOTH</code>	Polygon antialiasing
<code>GL_SCISSOR_TEST</code>	Scissoring enabled
<code>GL_STENCIL_TEST</code>	Stencil test
<code>GL_TEXTURE_xD</code>	xD texturing (1, 2, or 3)
<code>GL_TEXTURE_CUBE_MAP</code>	Cube map texturing
<code>GL_TEXTURE_GEN_x</code>	Texgen for x (S, T, R, or Q)

Returns: None.

See Also: `glIsEnabled`, `glPopAttrib`, `glPushAttrib`

glFinish

Purpose: Forces all previous OpenGL commands to complete.

Syntax:

```
void glFinish(void);
```

Description: OpenGL commands are usually queued and executed in batches to optimize performance. The `glFinish` command forces all pending OpenGL commands to be executed. Unlike `glFlush`, this function does not return until all the rendering operations have been completed.

Returns: None.

See Also: `glFlush()`;

glFlush

Purpose: Flushes OpenGL command queues and buffers.

Syntax:

```
void glFlush(void);
```

Description: OpenGL commands are usually queued and executed in batches to optimize performance. This can vary among hardware, drivers, and OpenGL implementations. The `glFlush` command causes any waiting commands to be executed. This must be accomplished "in finite time." This is essentially the same as asynchronous execution of the graphics commands because `glFlush` returns immediately.

Returns: None.

See Also: `glFinish`

glGetXXXXV

Purpose: Retrieves a numeric state value or array of values.

Variations:

```
void glGetBooleanv(GLenum value, Glboolean *data);
void glGetIntegerv(GLenum value, int *data);
void glGetFloatv(GLenum value, float *data);
void glGetDoublev(GLenum value, float *data);
```

Description: Many OpenGL state variables are completely identified by symbolic constants. The values of these state variables can be retrieved with the `glGetXXXXV` commands.

The complete list of OpenGL state values is more than 28 pages long and can be found in [Table 6.6](#) of the OpenGL specification included on the CD-ROM.

Returns: Fills in the supplied buffer with the OpenGL state information.

glGetError

Purpose: Checks for OpenGL errors.

Syntax:

```
GLenum glGetError(void);
```

Description: This function returns one of the OpenGL error codes listed in [Table 2.3](#). Error codes are cleared when checked, and multiple error flags may be currently active. To retrieve all errors, call this function repeatedly until it return `GL_NO_ERROR`.

Returns: One of the OpenGL error codes listed in [Table 2.3](#).

glGetString

Purpose: Retrieves descriptive strings about the OpenGL implementation.

Syntax:

```
const GLubyte* glGetString(GLenum name);
```

Description: This function returns a character array describing some aspect of the current OpenGL implementation. The parameter `GL_VENDOR` returns the name of the vendor. `GL_RENDERER` is implementation dependent and may contain a brand name or the name of the vendor. `GL_VERSION` returns the version number followed by a space and any vendor-specific information. `GL_EXTENSIONS` returns a list of space-separated extension names supported by the implementation.

Returns: A constant byte array (character string) containing the requested string.

glHint

Purpose: Allows optional control of certain rendering behaviors.

Syntax:

```
void glHint(GLenum target, GLenum hint);
```

Description: Certain aspects of GL behavior may be controlled with hints. Valid hints are `GL_NICEST`, `GL_FASTEST`, and `GL_DONT_CARE`. Hints allow the programmers to specify whether they care more about rendering quality (`GL_NICEST`), performance (`GL_FASTEST`), or use the default for the implementation (`GL_DONT_CARE`).

- `GL_PERSPECTIVE_CORRECTION_HINT`— Desired quality of parameter interpolation
- `GL_POINT_SMOOTH_HINT`— Desired sampling quality of points
- `GL_LINE_SMOOTH_HINT`— Desired sampling quality of lines
- `GL_POLYGON_SMOOTH_HINT`— Desired sampling quality of polygons
- `GL_FOG_HINT`— Calculate fog by vertex (`GL_FASTEST`) or per pixel (`GL_NICEST`)
- `GL_GENERATE_MIPMAP_HINT`— Quality and performance of automatic mipmap-level generation
- `GL_TEXTURE_COMPRESSION_HINT`— Quality and performance of compressing texture images

Returns: None.

glIsEnabled

Purpose: Tests an OpenGL state variable to see whether it is enabled.

Syntax:

```
void glIsEnabled(GLenum feature);
```

Description: Many OpenGL state variables can be turned on and off with `glEnable` or `glDisable`. This function allows you to query a given state variable to see whether it is currently enabled. [Table 2.5](#) shows the list of states that can be queried.

Returns: None.

See Also: `glEnable`, `glDisable`

glOrtho

Purpose: Sets or modifies the clipping volume extents.

Syntax:

```
void glOrtho(GLdouble left, GLdouble right,
    GLdouble bottom, GLdouble top, GLdouble near,
    GLdouble far);
```

Description: This function describes a parallel clipping volume. This projection means that objects far from the viewer do not appear smaller (in contrast to a perspective projection). Think of the clipping volume in terms of 3D Cartesian coordinates, in which case `left` and `right` are the minimum and maximum x values, `top` and `bottom` are the minimum and maximum y values, and `near` and `far` are the minimum and maximum z values.

Parameters:

<code>left</code>	<code>GLdouble</code> : The leftmost coordinate of the clipping volume.
<code>right</code>	<code>GLdouble</code> : The rightmost coordinate of the clipping volume.
<code>bottom</code>	<code>GLdouble</code> : The bottommost coordinate of the clipping volume.
<code>top</code>	<code>GLdouble</code> : The topmost coordinate of the clipping volume.
<code>near</code>	<code>GLdouble</code> : The maximum distance from the origin to the viewer.
<code>far</code>	<code>GLdouble</code> : The maximum distance from the origin away from the viewer.

Returns: None.

See Also: `glViewport`

glPushAttrib/glPopAttrib

Purpose: Save and restore a set of related OpenGL state values.

Syntax:

```
void glPushAttrib(GLbitfield mask);
void glPopAttrib(GLbitfield mask);
```

Description: OpenGL allows for whole groups of state variables to be saved and retrieved. These functions push these groups onto an attribute stack and allow for them to be popped back off the stack. [Table 2.6](#) shows the complete list of attribute groups.

Returns: None.

Table 2.6. OpenGL Attribute Groups

Constant	Attributes
GL_ACCUM_BUFFER_BIT	Accumulation buffer settings
GL_COLOR_BUFFER_BIT	Color buffer settings
GL_CURRENT_BIT	Current color and coordinates
GL_DEPTH_BUFFER_BIT	Depth buffer settings
GL_ENABLE_BIT	All enabled flags
GL_EVAL_BIT	Evaluator settings
GL_FOG_BIT	Fog settings
GL_HINT_BIT	All OpenGL hints
GL_LIGHTING_BIT	Lighting Settings
GL_LINE_BIT	Line settings
GL_LIST_BIT	Display list settings
GL_MULTISAMPLE_BIT	Multisampling
GL_PIXEL_MODE_BIT	Pixel mode
GL_POINT_BIT	Point settings
GL_POLYGON_BIT	Polygon mode settings
GL_POLYGON_STIPPLE_BIT	Polygon stipple settings
GL_SCISSOR_BIT	Scissor test settings
GL_STENCIL_BUFFER_BIT	Stencil buffer settings
GL_TEXTURE_BIT	Texture settings
GL_TRANSFORM_BIT	Transformation settings
GL_VIEWPORT_BIT	Viewport settings
GL_ALL_ATTRIB_BITS	All OpenGL states

glRect

Purpose: Draws a flat rectangle.

Variations:

```
void glRectd(GLdouble x1, GLdouble y1, GLdouble x2
  ↪ , GLdouble y2);
void glRectf(GLfloat x1, GLfloat y1, GLfloat x2,
  ↪ GLfloat y2);
void glRecti(GLint x1, GLint y1, GLint x2, GLint y2);
void glRects(GLshort x1, GLshort y1, GLshort x1,
  ↪ GLshort y2);
void glRectdv(const GLdouble *v1, const GLdouble *v2);
void glRectfv(const GLfloat *v1, const GLfloat *v2);
void glRectiv(const GLint *v1, const GLint *v2);
void glRectsv(const GLshort *v1, const GLshort *v2);
```

Description: This function provides a simple method of specifying a rectangle as two corner points. The rectangle is drawn in the xy plane at $z = 0$.

Parameters:

x1, y1 Specifies the upper-left corner of the rectangle.
x2, y2 Specifies the lower-right corner of the rectangle.
**v1* An array of two values specifying the upper-left corner of the rectangle. Could also be described as *v1[2]*.
**v2* An array of two values specifying the lower-right corner of the rectangle. Could also be described as *v2[2]*.

Returns: None.

glViewport

Purpose: Sets the portion of a window that can be drawn in by OpenGL.

Syntax:

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

Description: This function sets the region within a window that is used for mapping the clipping volume coordinates to physical window coordinates.

Parameters:

x **GLint**: The number of pixels from the left side of the window to start the viewport.
y **GLint**: The number of pixels from the bottom of the window to start the viewport.
width **GLsizei**: The width in pixels of the viewport.
height **GLsizei**: The height in pixels of the viewport.

Returns: None.

See Also: [glOrtho](#)

gluErrorString**Purpose:** Returns a character string explanation for an OpenGL error code.**Syntax:**

```
const GLubyte* gluErrorString(GLenum errCode);
```

Description: This function returns an error string, given an error code returned by the `glGetError` function.**Parameters:**

`errCode` An OpenGL error code.

Returns: A constant pointer to an OpenGL error string.

See Also: `glGetError`

glutCreateWindow**Purpose:** Creates an OpenGL-enabled window.**Syntax:**

```
int glutCreateWindow(char *name);
```

Description: This function creates a top-level window in GLUT. This is considered the current window.**Parameters:**

`name` `char *`: The caption the window is to bear.

Returns: An integer uniquely identifying the window created.

See Also: `glutInitDisplayMode`

glutDisplayFunc**Purpose:** Sets the display callback function for the current window.**Syntax:**

```
void glutDisplayFunc(void (*func)(void));
```

Description: This function tells GLUT which function to call whenever the windows contents must be drawn. This can occur when the window is resized or uncovered or when GLUT is specifically asked to refresh with a call to the `glutPostRedisplay` function. Note that GLUT does not explicitly call `glFlush` or `glutSwapBuffers` for you after this function is called.

Parameters:

`func` `(*func)(void)`: The name of the function that does the rendering.

Returns: None.

See Also: `glFlush`, `glutSwapBuffers`, `glutReshapeFunc`

glutInitDisplayMode

Purpose: Initializes the display mode of the GLUT library OpenGL window.

Syntax:

```
void glutInitDisplayMode(unsigned int mode);
```

Description: This is the first function that must be called by a GLUT-based program to set up the OpenGL window. This function sets the characteristics of the window that OpenGL will use for drawing operations.

Parameters:

`mode` `unsigned int`: A mask or bitwise combination of masks from [Table 2.7](#). These mask values may be combined with a bitwise `OR`.

Returns: None.

See Also: `glutCreateWindow`

Table 2.7. Mask Values for Window Characteristics

Mask Value	Meaning
<code>GLUT_SINGLE</code>	Specifies a single-buffered window
<code>GLUT_DOUBLE</code>	Specifies a double-buffered window
<code>GLUT_RGBA</code> or <code>GLUT_RGB</code>	Specifies an RGBA-mode window
<code>GLUT_DEPTH</code>	Specifies a 32-bit depth buffer
<code>GLUT_LUMINANCE</code>	Specifies a luminance only color buffer
<code>GLUT_MULTISAMPLE</code>	Specifies a multisampled color buffer
<code>GLUT_STENCIL</code>	Specifies a stencil buffer
<code>GLUT_STEREO</code>	Specifies a stereo color buffer
<code>GLUT_ACCUM</code>	Specifies an accumulation buffer

GLUT_ALPHA	Specifies a destination alpha buffer
------------	--------------------------------------

glutKeyboardFunc

Purpose: Sets the keyboard callback function for the current window.

Syntax:

```
void glutKeyboardFunc(void (*func)(unsigned char
➥ key, int x, int y);
```

Description: This function establishes a callback function called by GLUT whenever one of the ASCII generating keys is pressed. Non-ASCII generating keys such as the Shift key are handled by the `glutSpecialFunc` callback. In addition to the ASCII value of the keystroke, the current x and y position of the mouse are returned.

Parameters:

`func` (`(*func)(unsigned char key, int x, int y)`): The name of the function to be called by GLUT when a keystroke occurs.

Returns: None.

glutMainLoop

Purpose: Starts the main GLUT processing loop.

Syntax:

```
void glutMainLoop(void);
```

Description: This function begins the main GLUT event-handling loop. The event loop is the place where all keyboard, mouse, timer, redraw, and other window messages are handled. This function does not return until program termination.

Parameters: None.

Returns: None.

glutMouseFunc

Purpose: Sets the mouse callback function for the current window.

Syntax:

```
void glutMouseFunc(void (*func)(int button, int
➥ state, int x, int y);
```

Description: This function establishes a callback function called by GLUT whenever a mouse event occurs. Three values are valid for the button parameter: `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON`, and `GLUT_RIGHT_BUTTON`. The state parameter is either `GLUT_UP` or `GLUT_DOWN`.

Parameters:

`func` `(*func)(int button, int state, int x, int y)`: The name of the function to be called by GLUT when a mouse event occurs.

Returns: None.

See Also: `glutSpecialFunc`, `glutKeyboardFunc`

glutReshapeFunc

Purpose: Sets the window reshape callback function for the current window.

Syntax:

```
void glutReshapeFunc(void (*func)(int width, int
➥ height));
```

Description: This function establishes a callback function called by GLUT whenever the window changes size or shape (including at least once when the window is created). The callback function receives the new width and height of the window.

Parameters:

`func` `(*func)(int x, int y)`: The name of the function to be called by GLUT when the window size changes.

Returns: None.

See Also: `glutDisplayFunc`

glutPostRedisplay

Purpose: Tells GLUT to refresh the current window.

Syntax:

```
void glutPostRedisplay(void);
```

Description: This function informs the GLUT library that the current window needs to be refreshed. Multiple calls to this function before the next refresh result in only one repainting of the window.

Parameters: None.

Returns: None.

See Also: `glutDisplayFunc`

glutSolidTeapot, glutWireTeapot**Purpose:** Draws a solid or wireframe teapot.**Syntax:**

```
void glutSolidTeapot(GLdouble size);
void glutWireTeapot(GLdouble size);
```

Description: This function draws a solid or wireframe teapot. This is the famous teapot seen so widely in computer graphics samples. In addition to surface normals for lighting, texture coordinates are also generated.**Parameters:**

size **GLdouble:** The approximate radius of the teapot. A sphere of this radius would totally enclose the model.

Returns: None.**glutSpecialFunc****Purpose:** Sets a special keyboard callback function for the current window for non-ASCII keystrokes.**Syntax:**

```
void glutSpecialFunc(void (*func)(int key, int x,
→ int y);
```

Description: This function establishes a callback function called by GLUT whenever one of the non-ASCII generating keys is pressed. Non-ASCII generating keys are keystrokes such as the Shift key, which can't be identified by an ASCII value. In addition, the current x and y position of the mouse are returned. Valid values for the key parameter are listed in [Table 2.8](#).**Parameters:**

func **(*func)(int key, int x, int y):** The name of the function to be called by GLUT when a non-ASCII keystroke occurs.

Returns: None.**See Also:** [glutKeyboardFunc](#), [glutMouseFunc](#)

**Table 2.8. Non-ASCII Key Values
Passed to the `glutSpecialFunc`
Callback**

Key Value	Keystroke

<code>GLUT_KEY_F1</code>	F1 key
<code>GLUT_KEY_F2</code>	F2 key
<code>GLUT_KEY_F3</code>	F3 key
<code>GLUT_KEY_F4</code>	F4 key
<code>GLUT_KEY_F5</code>	F5 key
<code>GLUT_KEY_F6</code>	F6 key
<code>GLUT_KEY_F7</code>	F7 key
<code>GLUT_KEY_F8</code>	F8 key
<code>GLUT_KEY_F9</code>	F9 key
<code>GLUT_KEY_F10</code>	F10 key
<code>GLUT_KEY_F11</code>	F11 key
<code>GLUT_KEY_F12</code>	F12 key
<code>GLUT_KEY_LEFT</code>	Left-arrow key
<code>GLUT_KEY_RIGHT</code>	Right-arrow key
<code>GLUT_KEY_UP</code>	Up-arrow key
<code>GLUT_KEY_DOWN</code>	Down-arrow key
<code>GLUT_KEY_PAGE_UP</code>	Page Up key
<code>GLUT_KEY_PAGE_DOWN</code>	Page Down key
<code>GLUT_KEY_HOME</code>	Home key
<code>GLUT_KEY_END</code>	End key
<code>GLUT_KEY_INSERT</code>	Insert key

glutSwapBuffers

Purpose: Performs a buffer swap in double-buffered mode.

Syntax:

```
void glutSwapBuffers(void);
```

Description: When the current GLUT window is operating in double-buffered mode, this function performs a flush of the OpenGL pipeline and does a buffer swap (places the hidden rendered image onscreen). If the current window is not in double-buffered mode, a flush of the pipeline is still performed.

Parameters: None.

Returns: None.

See Also: `glutDisplayFunc`

glutTimerFunc

Purpose: Registers a callback function to be called by GLUT after the timeout value expires.

Syntax:

```
void glutTimerFunc(unsigned int msecs, (*func)(int
➥ value), int value);
```

Description: This function registers a callback function that should be called after *msecs* milliseconds have elapsed. The callback function is passed the user-specified value in the *value* parameter.

Parameters:

msecs `unsigned int`: The number of milliseconds to wait before calling the specified function.

func `void (*func)(int value)`: The name of the function to be called when the timeout value expires.

value `int`: User-specified value passed to the callback function when it is executed.

Returns: None.

Chapter 3. Drawing in Space: Geometric Primitives and Buffers

by Richard S. Wright, Jr.

WHAT YOU'LL LEARN IN THIS CHAPTER:

How To	Functions You'll Use
Draw points, lines, and shapes	<code>glBegin/glEnd/glVertex</code>
Set shape outlines to wireframe or solid objects	<code>glPolygonMode</code>
Set point sizes for drawing	<code>glPointSize</code>
Set line drawing width	<code>glLineWidth</code>
Perform hidden surface removal	<code>glCullFace/glClear</code>
Set patterns for broken lines	<code>glLineStipple</code>
Set polygon fill patterns	<code>glPolygonStipple</code>
Use the OpenGL Scissor box	<code>glScissor</code>
Use the stencil buffer	<code>glStencilFunc/glStencilMask/glStencilOp</code>

If you've ever had a chemistry class (and probably even if you haven't), you know that all matter consists of atoms and that all atoms consist of only three things: protons, neutrons, and electrons. All the materials and substances you have ever come into contact with—from the petals of a rose to the sand on the beach—are just different arrangements of these three fundamental building blocks. Although this explanation is a little oversimplified for almost anyone beyond the third or fourth grade, it demonstrates a powerful principle: With just a few simple building blocks, you can create highly complex and beautiful structures.

The connection is fairly obvious. Objects and scenes that you create with OpenGL also consist of

smaller, simpler shapes, arranged and combined in various and unique ways. This chapter explores these building blocks of 3D objects, called *primitives*. All primitives in OpenGL are one- or two-dimensional objects, ranging from single points to lines and complex polygons. In this chapter, you learn everything you need to know to draw objects in three dimensions from these simpler shapes.

Drawing Points in 3D

When you first learned to draw any kind of graphics on any computer system, you probably started with pixels. A pixel is the smallest element on your computer monitor, and on color systems, that pixel can be any one of many available colors. This is computer graphics at its simplest: Draw a point somewhere on the screen, and make it a specific color. Then build on this simple concept, using your favorite computer language to produce lines, polygons, circles, and other shapes and graphics. Perhaps even a GUI...

With OpenGL, however, drawing on the computer screen is fundamentally different. You're not concerned with physical screen coordinates and pixels, but rather positional coordinates in your viewing volume. You let OpenGL worry about how to get your points, lines, and everything else projected from your established 3D space to the 2D image made by your computer screen.

This chapter and the next cover the most fundamental concepts of OpenGL or any 3D graphics toolkit. In the upcoming chapter, we provide substantial detail about how this transformation from 3D space to the 2D landscape of your computer monitor takes place, as well as how to transform (rotate, translate, and scale) your objects. For now, we take this ability for granted to focus on plotting and drawing in a 3D coordinate system. This approach might seem backward, but if you first know how to draw something and then worry about all the ways to manipulate your drawings, the material in [Chapter 4](#), "Geometric Transformations: The Pipeline," is more interesting and easier to learn. When you have a solid understanding of graphics primitives and coordinate transformations, you will be able to quickly master any 3D graphics language or API.

A 3D Point: The Vertex

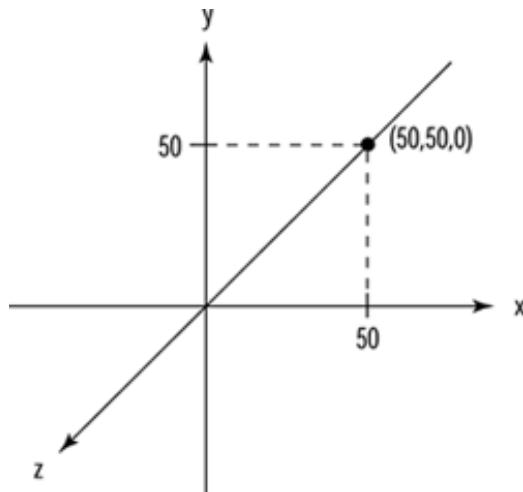
To specify a drawing point in this 3D "palette," we use the OpenGL function `glVertex`—without a doubt the most used function in all the OpenGL API. This is the "lowest common denominator" of all the OpenGL primitives: a single point in space. The `glVertex` function can take from one to four parameters of any numerical type, from bytes to doubles, subject to the naming conventions discussed in [Chapter 2](#), "Using OpenGL."

The following single line of code specifies a point in our coordinate system located 50 units along the x-axis, 50 units along the y-axis, and 0 units out the z-axis:

```
glVertex3f(50.0f, 50.0f, 0.0f);
```

[Figure 3.2](#) illustrates this point. Here, we chose to represent the coordinates as floating-point values, as we do for the remainder of the book. Also, the form of `glVertex` that we use takes three arguments for the x, y, and z coordinate values, respectively.

Figure 3.2. The point (50,50,0) as specified by `glVertex3f(50.0f, 50.0f, 0.0f)`.



Two other forms of `glVertex` take two and four arguments, respectively. We can represent the same point in [Figure 3.2](#) with this code:

```
glVertex2f(50.0f, 50.0f);
```

This form of `glVertex` takes only two arguments that specify the x and y values and assumes the z coordinate to be 0.0 always.

The form of `glVertex` taking four arguments, `glVertex4`, uses a fourth coordinate value w (set to 1.0 by default when not specified) for scaling purposes. You will learn more about this coordinate in [Chapter 4](#) when we spend more time exploring coordinate transformations.

Draw Something!

Now, we have a way of specifying a point in space to OpenGL. What can we make of it, and how do we tell OpenGL what to do with it? Is this vertex a point that should just be plotted? Is it the endpoint of a line or the corner of a cube? The geometric definition of a vertex is not just a point in space, but rather the point at which an intersection of two lines or curves occurs. This is the essence of primitives.

A primitive is simply the interpretation of a set or list of vertices into some shape drawn on the screen. There are 10 primitives in OpenGL, from a simple point drawn in space to a closed polygon of any number of sides. One way to draw primitives is to use the `glBegin` command to tell OpenGL to begin interpreting a list of vertices as a particular primitive. You then end the list of vertices for that primitive with the `glEnd` command. Kind of intuitive, don't you think?

Drawing Points

Let's begin with the first and simplest of primitives: points. Look at the following code:

```
glBegin(GL_POINTS);           // Select points as the primitive
    glVertex3f(0.0f, 0.0f, 0.0f); // Specify a point
    glVertex3f(50.0f, 50.0f, 50.0f); // Specify another point
glEnd();                     // Done drawing points
```

The argument to `glBegin`, `GL_POINTS`, tells OpenGL that the following vertices are to be interpreted and drawn as points. Two vertices are listed here, which translates to two specific points, both of which would be drawn.

This example brings up an important point about `glBegin` and `glEnd`: You can list multiple primitives between calls as long as they are for the same primitive type. In this way, with a single

`glBegin/glEnd` sequence, you can include as many primitives as you like. This next code segment is wasteful and will execute more slowly than the preceding code:

```
glBegin(GL_POINTS);           // Specify point drawing
    glVertex3f(0.0f, 0.0f, 0.0f);
glEnd();
glBegin(GL_POINTS);           // Specify another point
    glVertex3f(50.0f, 50.0f, 50.0f);
glEnd()
```

INDENTING YOUR CODE

In the foregoing examples, did you notice the indenting style used for the calls to `glVertex`? Most OpenGL programmers use this convention to make the code easier to read. It is not required, but it does make finding where primitives start and stop easier.

Our First Example

The code in [Listing 3.2](#) draws some points in our 3D environment. It uses some simple trigonometry to draw a series of points that form a corkscrew path up the z-axis. This code is from the POINTS program, which is on the CD in the subdirectory for this chapter. All the sample programs use the framework we established in [Chapter 2](#). Notice that in the `SetupRC` function, we are setting the current drawing color to green.

Listing 3.2. Rendering Code to Produce a Spring-Shaped Path of Points

```
// Define a constant for the value of PI
#define GL_PI 3.1415f
// This function does any needed initialization on the rendering
// context.
void SetupRC()
{
    //
    // Black background
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f );
    // Set drawing color to green
    glColor3f(0.0f, 1.0f, 0.0f);
}
// Called to draw scene
void RenderScene(void)
{
    GLfloat x,y,z,angle; // Storage for coordinates and angles
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT);
    // Save matrix state and do the rotation
    glPushMatrix();
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);
    // Call only once for all remaining points
    glBegin(GL_POINTS);
    z = -50.0f;
    for(angle = 0.0f; angle <= (2.0f*GL_PI)*3.0f; angle += 0.1f)
    {
        x = 50.0f*sin(angle);
        y = 50.0f*cos(angle);

        // Specify the point and move the Z value up a little
        glVertex3f(x, y, z);
        z += 0.5f;
    }
}
```

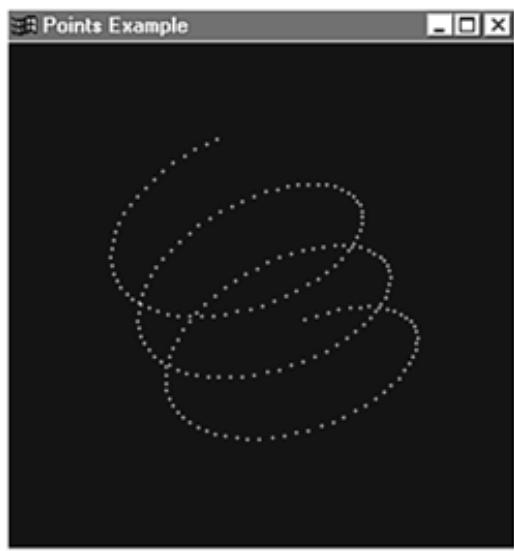
```

    }
// Done drawing points
glEnd();
// Restore transformations
glPopMatrix();
// Flush drawing commands
glFlush();
}

```

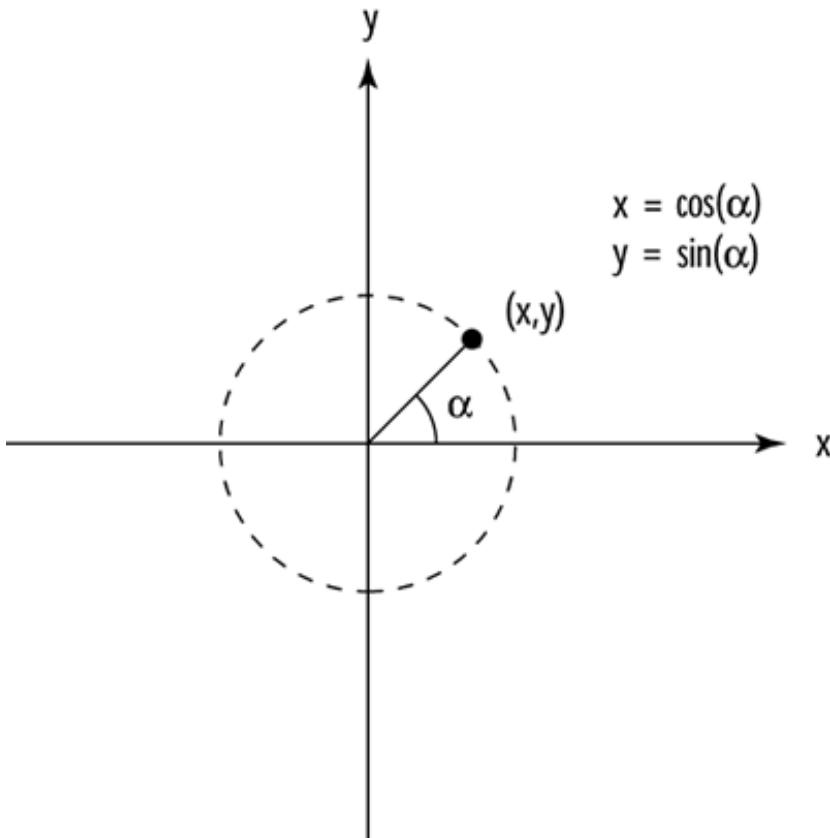
Only the code between calls to `glBegin` and `glEnd` is important for our purpose in this and the other examples for this chapter. This code calculates the x and y coordinates for an angle that spins between 0° and 360° three times. We express this programmatically in radians rather than degrees; if you don't know trigonometry, you can take our word for it. If you're interested, see the box "[The Trigonometry of Radians/Degrees](#)." Each time a point is drawn, the z value is increased slightly. When this program is run, all you see is a circle of points because you are initially looking directly down the z-axis. To see the effect, use the arrow keys to spin the drawing around the x- and y-axes. The effect is illustrated in [Figure 3.3](#).

Figure 3.3. Output from the POINTS sample program.



ONE THING AT A TIME

Again, don't get too distracted by the functions in this example that we haven't covered yet (`glPushMatrix`, `glPopMatrix`, and `glRotate`). These functions are used to rotate the image around so you can better see the positioning of the points as they are drawn in 3D space. We cover these functions in some detail in [Chapter 4](#). If we hadn't used these features now, you wouldn't be able to see the effects of your 3D drawings, and this and the following sample programs wouldn't be very interesting to look at. For the rest of the sample code in this chapter, we show only the code that includes the `glBegin` and `glEnd` statements.



THE TRIGONOMETRY OF RADIANS/DEGREES

The figure in this box shows a circle drawn in the xy plane. A line segment from the origin (0,0) to any point on the circle makes an angle (α) with the x-axis. For any given angle, the trigonometric functions sine and cosine return the x and y values of the point on the circle. By stepping a variable that represents the angle all the way around the origin, we can calculate all the points on the circle. Note that the C runtime functions `sin()` and `cos()` accept angle values measured in radians instead of degrees. There are 2π radians in a circle, where π is a nonrational number that is approximately 3.1415. (*Nonrational* means there are an infinite number of values past the decimal point.)

Setting the Point Size

When you draw a single point, the size of the point is one pixel by default. You can change this size with the function `glPointSize`:

```
void glPointSize(GLfloat size);
```

The `glPointSize` function takes a single parameter that specifies the approximate diameter in pixels of the point drawn. Not all point sizes are supported, however, and you should make sure the point size you specify is available. Use the following code to get the range of point sizes and the smallest interval between them:

```
GLfloat sizes[2];      // Store supported point size range
GLfloat step;          // Store supported point size increments
// Get supported point size range and step size
glGetFloatv(GL_POINT_SIZE_RANGE, sizes);
glGetFloatv(GL_POINT_SIZE_GRANULARITY, &step);
```

Here, the `sizes` array will contain two elements that contain the smallest and largest valid value for `glPointSize`. In addition, the variable `step` will hold the smallest step size allowable between the point sizes. The OpenGL specification requires only that one point size, 1.0, be supported. The Microsoft software implementation of OpenGL, for example, allows for point sizes from 0.5 to 10.0, with 0.125 the smallest step size. Specifying a size out of range is not interpreted as an error. Instead, the largest or smallest supported size is used, whichever is closest to the value specified.

Points, unlike other geometry, are not affected by the perspective division. That is, they do not become smaller when they are further from the viewpoint, and they do not become larger as they move closer. Points are also always square pixels, even if you use `glPointSize` to increase the size of the points. You just get bigger squares! To get round points, you must draw them antialiased (coming up in the next chapter).

OPENGL STATE VARIABLES

As we discussed in [Chapter 2](#), OpenGL maintains the state of many of its internal variables and settings. This collection of settings is called the *OpenGL State Machine*. You can query the State Machine to determine the state of any of its variables and settings. Any feature or capability you enable or disable with `glEnable`/`glDisable`, as well as numeric settings set with `glSet`, can be queried with the many variations of `glGet`.

Let's look at a sample that uses these new functions. The code in [Listing 3.3](#) produces the same spiral shape as our first example, but this time, the point sizes are gradually increased from the smallest valid size to the largest valid size. This example is from the program POINTSZ in the CD subdirectory for this chapter. The output from POINTSZ shown in [Figure 3.4](#) was run on Microsoft's software implementation of OpenGL. [Figure 3.5](#) shows the same program run on a hardware accelerator that supports much larger point sizes.

Listing 3.3. Code from POINTSZ That Produces a Spiral with Gradually Increasing Point Sizes

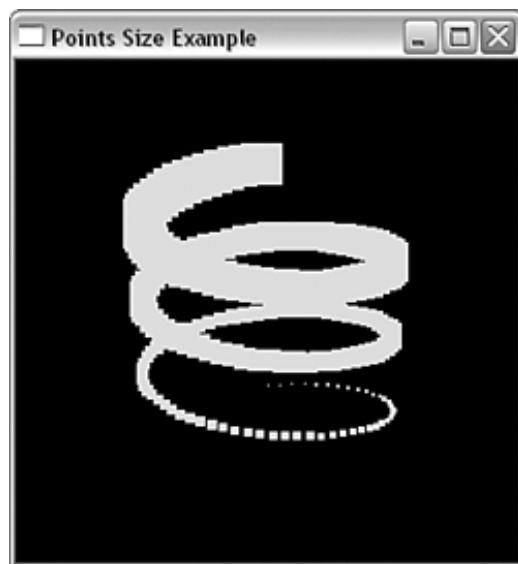
```
// Define a constant for the value of PI
#define GL_PI 3.1415f
// Called to draw scene
void RenderScene(void)
{
    GLfloat x,y,z,angle;    // Storage for coordinates and angles
    GLfloat sizes[2];        // Store supported point size range
    GLfloat step;            // Store supported point size increments
    GLfloat curSize;         // Store current point size
    ...
    ...
    // Get supported point size range and step size
    glGetFloatv(GL_POINT_SIZE_RANGE,sizes);
    glGetFloatv(GL_POINT_SIZE_GRANULARITY,&step);
    // Set the initial point size
    curSize = sizes[0];
    // Set beginning z coordinate
    z = -50.0f;
    // Loop around in a circle three times
    for(angle = 0.0f; angle <= (2.0f*GL_PI)*3.0f; angle += 0.1f)
    {
        // Calculate x and y values on the circle
        x = 50.0f*sin(angle);
        y = 50.0f*cos(angle);
        // Specify the point size before the primitive is specified
        glPointSize(curSize);
        glBegin(GL_POINTS);
        glVertex3f(x,y,z);
        glEnd();
        curSize += step;
    }
}
```

```
glPointSize(curSize);
// Draw the point
glBegin(GL_POINTS);
    glVertex3f(x, y, z);
glEnd();
// Bump up the z value and the point size
z += 0.5f;
curSize += step;
}
...
...
}
```

Figure 3.4. Output from the POINTSZ program.



Figure 3.5. Output from POINTSZ on hardware supporting much larger point sizes.



This example demonstrates a couple of important things. For starters, notice that `glPointSize` must be called outside the `glBegin/glEnd` statements. Not all OpenGL functions are valid between these function calls. Although `glPointSize` affects all points drawn after it, you don't begin drawing points until you call `glBegin(GL_POINTS)`. For a complete list of valid functions that you can call within a `glBegin/glEnd` sequence, see the reference section at the end of the chapter.

If you specify a point size larger than what is returned in the size variable, you also may notice (depending on your hardware) that OpenGL uses the largest available point size but does not keep growing. This is a general observation about OpenGL function parameters that have a valid range. Values outside the range are *clamped* to the range. Values too low are made the lowest valid value, and values too high are made the highest valid value.

The most obvious thing you probably noticed about the `POINTSZ` excerpt is that the larger point sizes are represented simply by larger cubes. This is the default behavior, but it typically is undesirable for many applications. Also, you might wonder why you can increase the point size by a value less than one. If a value of 1.0 represents one pixel, how do you draw less than a pixel or, say, 2.5 pixels?

The answer is that the point size specified in `glPointSize` isn't the exact point size in pixels, but the approximate diameter of a circle containing all the pixels that are used to draw the point. You can get OpenGL to draw the points as better points (that is, small filled circles) by enabling point smoothing. Together with line smoothing, point smoothing falls under the topic of *antialiasing*. Antialiasing is a technique used to smooth out jagged edges and round out corners; it is covered in more detail in [Chapter 6](#), "More on Colors and Materials."

Drawing Lines in 3D

The `GL_POINTS` primitive we have been using thus far is reasonably straightforward; for each vertex specified, it draws a point. The next logical step is to specify two vertices and draw a line between them. This is exactly what the next primitive, `GL_LINES`, does. The following short section of code draws a single line between two points (0,0,0) and (50,50,50):

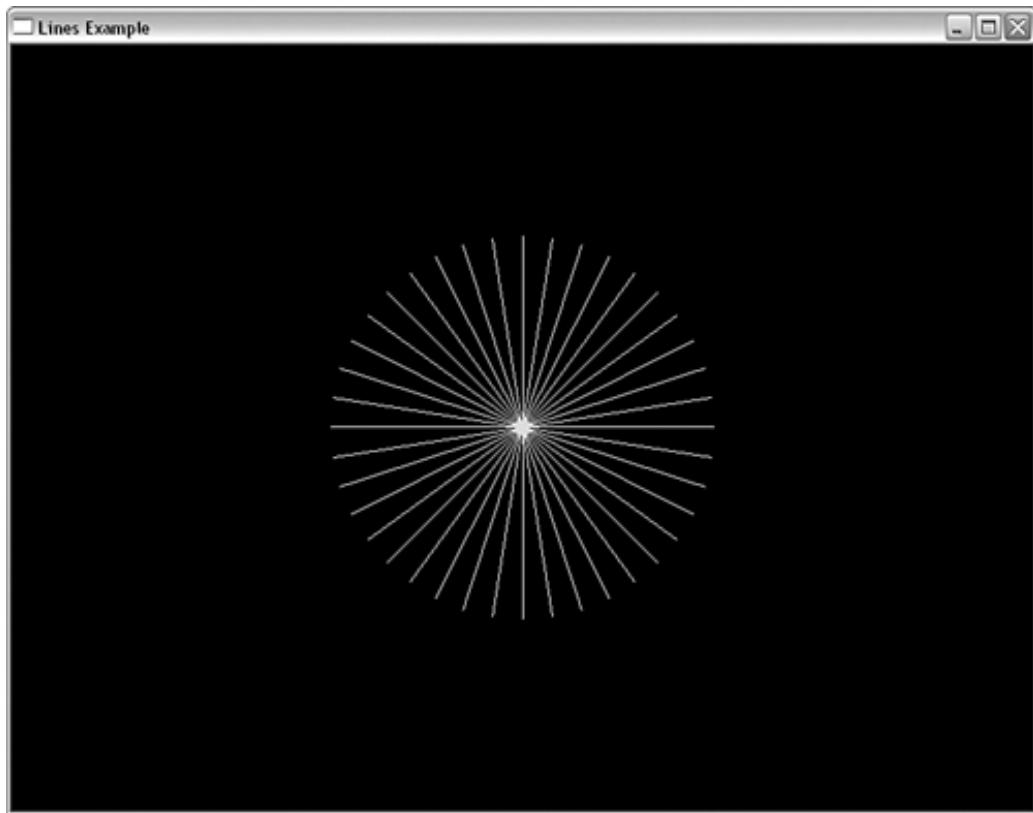
```
glBegin(GL_LINES);
    glVertex3f(0.0f, 0.0f, 0.0f);
    glVertex3f(50.0f, 50.0f, 50.0f);
glEnd();
```

Note here that two vertices specify a single primitive. For every two vertices specified, a single line is drawn. If you specify an odd number of vertices for `GL_LINES`, the last vertex is just ignored.

[Listing 3.4](#), from the `LINES` sample program on the CD, shows a more complex sample that draws a series of lines fanned around in a circle. Each point specified in this sample is paired with a point on the opposite side of a circle. The output from this program is shown in [Figure 3.6](#).

Listing 3.4. Code from the Sample Program LINES That Displays a Series of Lines Fanned in a Circle

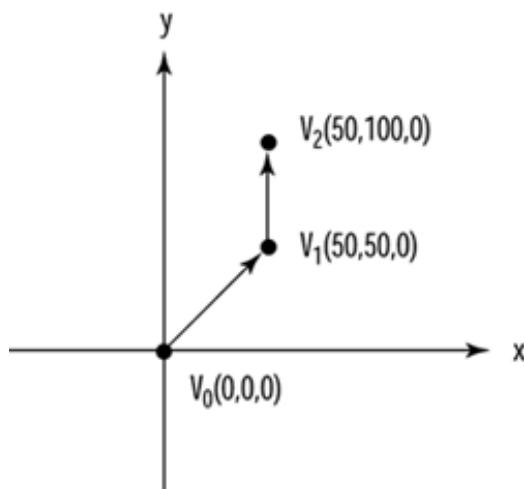
```
// Call only once for all remaining points
glBegin(GL_LINES);
// All lines lie in the xy plane.
z = 0.0f;
for(angle = 0.0f; angle <= GL_PI; angle += (GL_PI/20.0f))
{
    // Top half of the circle
    x = 50.0f*sin(angle);
    y = 50.0f*cos(angle);
    glVertex3f(x, y, z);           // First endpoint of line
    // Bottom half of the circle
    x = 50.0f*sin(angle + GL_PI);
    y = 50.0f*cos(angle + GL_PI);
    glVertex3f(x, y, z);           // Second endpoint of line
}
// Done drawing points
glEnd();
```

Figure 3.6. Output from the LINES sample program.

Line Strips and Loops

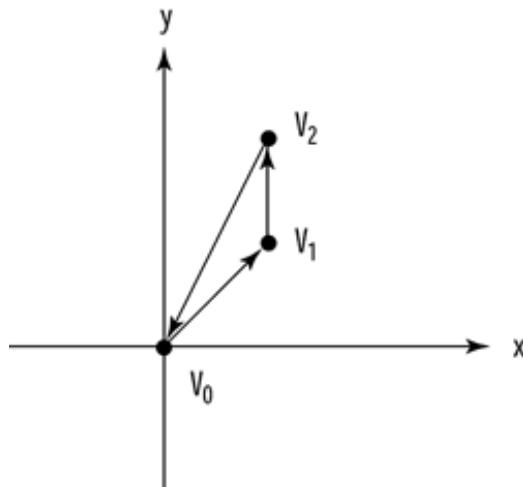
The next two OpenGL primitives build on `GL_LINES` by allowing you to specify a list of vertices through which a line is drawn. When you specify `GL_LINE_STRIP`, a line is drawn from one vertex to the next in a continuous segment. The following code draws two lines in the xy plane that are specified by three vertices. [Figure 3.7](#) shows an example.

```
glBegin(GL_LINE_STRIP);
    glVertex3f(0.0f, 0.0f, 0.0f);      // V0
    glVertex3f(50.0f, 50.0f, 0.0f);    // V1
    glVertex3f(50.0f, 100.0f, 0.0f);   // V2
glEnd();
```

Figure 3.7. An example of a `GL_LINE_STRIP` specified by three vertices.

The last line-based primitive is `GL_LINE_LOOP`. This primitive behaves just like `GL_LINE_STRIP`, but one final line is drawn between the last vertex specified and the first one specified. This is an easy way to draw a closed-line figure. Figure 3.8 shows a `GL_LINE_LOOP` drawn using the same vertices as for the `GL_LINE_STRIP` in Figure 3.7.

Figure 3.8. The same vertices from Figure 3.7 used by a `GL_LINE_LOOP` primitive.



Approximating Curves with Straight Lines

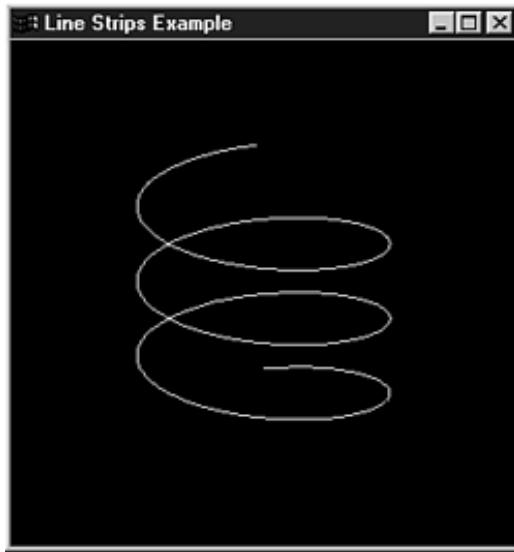
The `POINTS` sample program, shown earlier in Figure 3.3, showed you how to plot points along a spring-shaped path. You might have been tempted to push the points closer and closer together (by setting smaller values for the angle increment) to create a smooth spring-shaped curve instead of the broken points that only approximated the shape. This perfectly valid operation can move quite slowly for larger and more complex curves with thousands of points.

A better way of approximating a curve is to use `GL_LINE_STRIP` to play connect-the-dots. As the dots move closer together, a smoother curve materializes without your having to specify all those points. Listing 3.5 shows the code from Listing 3.2, with `GL_POINTS` replaced by `GL_LINE_STRIP`. The output from this new program, `LSTRIPS`, is shown in Figure 3.9. As you can see, the approximation of the curve is quite good. You will find this handy technique almost ubiquitous among OpenGL programs.

Listing 3.5. Code from the Sample Program LSTRIPS, Demonstrating Line Strips

```
// Call only once for all remaining points
glBegin(GL_LINE_STRIP);
z = -50.0f;
for(angle = 0.0f; angle <= (2.0f*GL_PI)*3.0f; angle += 0.1f)
{
    x = 50.0f*sin(angle);
    y = 50.0f*cos(angle);
    // Specify the point and move the z value up a little
    glVertex3f(x, y, z);
    z += 0.5f;
}
// Done drawing points
glEnd();
```

Figure 3.9. Output from the LSTRIPS program approximating a smooth curve.



Setting the Line Width

Just as you can set different point sizes, you can also specify various line widths when drawing lines by using the `glLineWidth` function:

```
void glLineWidth(GLfloat width);
```

The `glLineWidth` function takes a single parameter that specifies the approximate width, in pixels, of the line drawn. Just like point sizes, not all line widths are supported, and you should make sure the line width you want to specify is available. Use the following code to get the range of line widths and the smallest interval between them:

```
GLfloat sizes[2];      // Store supported line width range
GLfloat step;          // Store supported line width increments
// Get supported line width range and step size
glGetFloatv(GL_LINE_WIDTH_RANGE, sizes);
glGetFloatv(GL_LINE_WIDTH_GRANULARITY, &step);
```

Here, the `sizes` array will contain two elements that contain the smallest and largest valid value for `glLineWidth`. In addition, the variable `step` will hold the smallest step size allowable between the line widths. The OpenGL specification requires only that one line width, 1.0, be supported. The Microsoft implementation of OpenGL allows for line widths from 0.5 to 10.0, with 0.125 the smallest step size.

[Listing 3.6](#) shows code for a more substantial example of `glLineWidth`. It's from the program LINESW and draws 10 lines of varying widths. It starts at the bottom of the window at -90 on the y-axis and climbs the y-axis 20 units for each new line. Every time it draws a new line, it increases the line width by 1. [Figure 3.10](#) shows the output for this program.

Listing 3.6. Drawing Lines of Various Widths

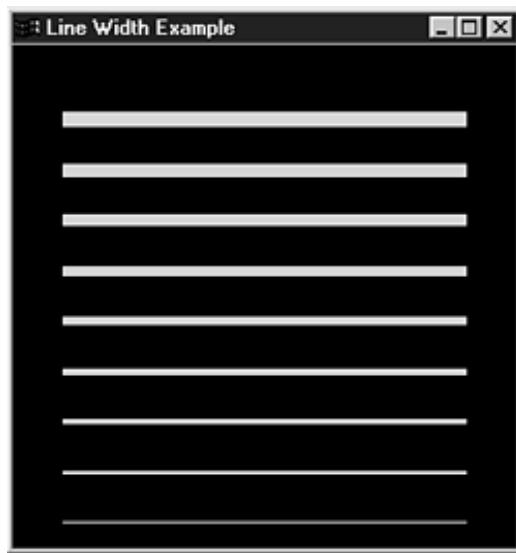
```
// Called to draw scene
void RenderScene(void)
{
    GLfloat y;                  // Storage for varying Y coordinate
    GLfloat fSizes[2];          // Line width range metrics
    GLfloat fCurrSize;          // Save current size
    ...
    ...
```

```

...
// Get line size metrics and save the smallest value
glGetFloatv(GL_LINE_WIDTH_RANGE, fSizes);
fCurrSize = fSizes[0];
// Step up y axis 20 units at a time
for(y = -90.0f; y < 90.0f; y += 20.0f)
{
    // Set the line width
    glLineWidth(fCurrSize);
    // Draw the line
    glBegin(GL_LINES);
        glVertex2f(-80.0f, y);
        glVertex2f(80.0f, y);
    glEnd();
    // Increase the line width
    fCurrSize += 1.0f;
}
...
...
}

```

Figure 3.10. Demonstration of `glLineWidth` from the LINESW program.



Notice that we used `glVertex2f` this time instead of `glVertex3f` to specify the coordinates for the lines. As mentioned, using this technique is only a convenience because we are drawing in the *xy* plane, with a *z* value of 0. To see that you are still drawing lines in three dimensions, simply use the arrow keys to spin your lines around. You easily see that all the lines lie on a single plane.

Line Stippling

In addition to changing line widths, you can create lines with a dotted or dashed pattern, called *stippling*. To use line stippling, you must first enable stippling with a call to

```
glEnable(GL_LINE_STIPPLE);
```

Then the function `glLineStipple` establishes the pattern that the lines use for drawing:

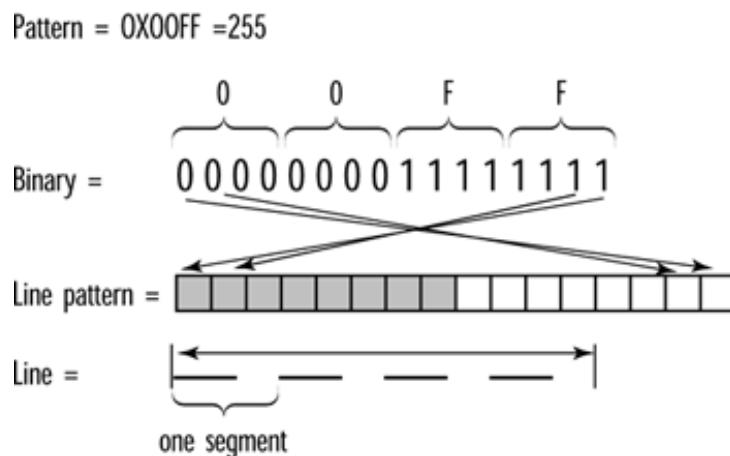
```
void glLineStipple(GLint factor, GLushort pattern);
```

REMINDER

Any feature or ability that is enabled by a call to `glEnable` can be disabled by a call to `glDisable`.

The `pattern` parameter is a 16-bit value that specifies a pattern to use when drawing the lines. Each bit represents a section of the line segment that is either on or off. By default, each bit corresponds to a single pixel, but the `factor` parameter serves as a multiplier to increase the width of the pattern. For example, setting `factor` to 5 causes each bit in the pattern to represent five pixels in a row that are either on or off. Furthermore, bit 0 (the least significant bit) of the pattern is used first to specify the line. [Figure 3.11](#) illustrates a sample bit pattern applied to a line segment.

Figure 3.11. A stipple pattern is used to construct a line segment.

**WHY ARE THESE PATTERNS BACKWARD?**

You might wonder why the bit pattern for stippling is used in reverse when drawing the line. Internally, it's much faster for OpenGL to shift this pattern to the left one place each time it needs to get the next mask value. OpenGL was designed for high-performance graphics and frequently employs similar tricks elsewhere.

[Listing 3.7](#) shows a sample of using a stippling pattern that is just a series of alternating on and off bits (0101010101010101). This code is taken from the LSTIPPLE program, which draws 10 lines from the bottom of the window up the y-axis to the top. Each line is stippled with the pattern 0x5555, but for each new line, the pattern multiplier is increased by 1. You can clearly see the effects of the widened stipple pattern in [Figure 3.12](#).

Listing 3.7. Code from LSTIPPLE That Demonstrates the Effect of `factor` on the Bit Pattern

```
// Called to draw scene
void RenderScene(void)
{
    GLfloat y;           // Storage for varying y coordinate
    GLint factor = 1;    // Stippling factor
    GLushort pattern = 0x5555; // Stipple pattern
    ...
    ...
    // Enable Stippling
```

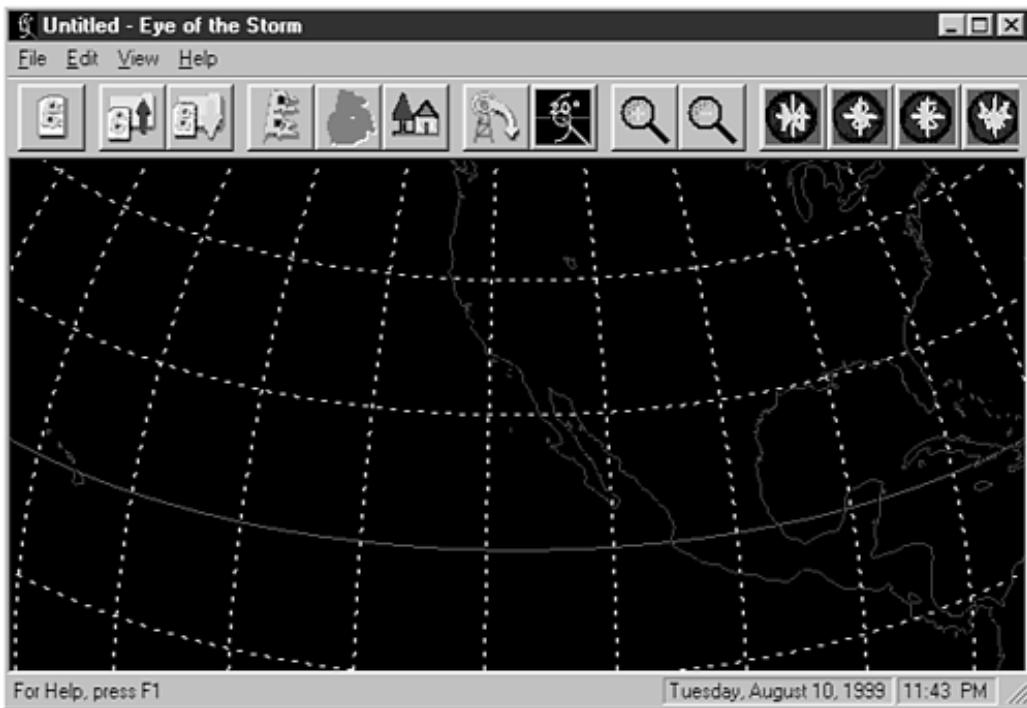
```
glEnable(GL_LINE_STIPPLE);
// Step up Y axis 20 units at a time
for(y = -90.0f; y < 90.0f; y += 20.0f)
{
    // Reset the repeat factor and pattern
    glLineStipple(factor,pattern);
    // Draw the line
    glBegin(GL_LINES);
        glVertex2f(-80.0f, y);
        glVertex2f(80.0f, y);
    glEnd();
    factor++;
}
...
...
}
```

Figure 3.12. Output from the LSTIPPLE program.



Just the ability to draw points and lines in 3D gives you a significant set of tools for creating your own 3D masterpiece. I wrote the commercial application shown in [Figure 3.13](#). Note that the OpenGL-rendered map is rendered entirely of solid and stippled line strips.

Figure 3.13. A 3D map rendered with solid and stippled lines.



Drawing Triangles in 3D

You've seen how to draw points and lines and even how to draw some enclosed polygons with `GL_LINE_LOOP`. With just these primitives, you could easily draw any shape possible in three dimensions. You could, for example, draw six squares and arrange them so they form the sides of a cube.

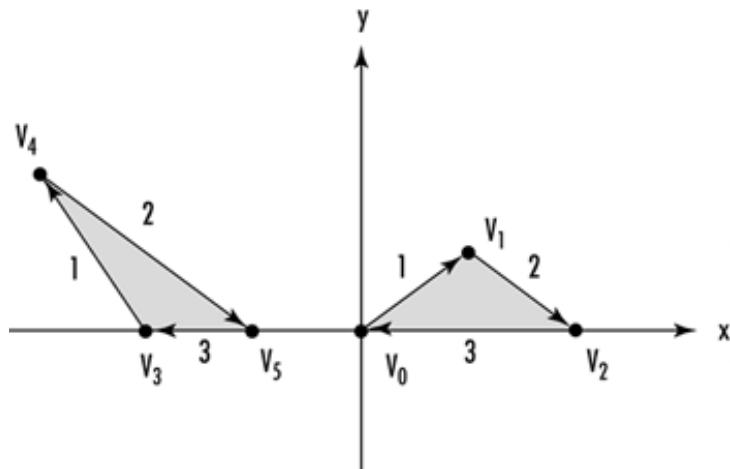
You might have noticed, however, that any shapes you create with these primitives are not filled with any color; after all, you are drawing only lines. In fact, all that arranging six squares produces is a wireframe cube, not a solid cube. To draw a solid surface, you need more than just points and lines; you need polygons. A polygon is a closed shape that may or may not be filled with the currently selected color, and it is the basis of all solid-object composition in OpenGL.

Triangles: Your First Polygon

The simplest polygon possible is the triangle, with only three sides. The `GL_TRIANGLES` primitive draws triangles by connecting three vertices together. The following code draws two triangles using three vertices each, as shown in [Figure 3.14](#):

```
glBegin(GL_TRIANGLES);
    glVertex2f(0.0f, 0.0f);           // V0
    glVertex2f(25.0f, 25.0f);        // V1
    glVertex2f(50.0f, 0.0f);         // V2
    glVertex2f(-50.0f, 0.0f);        // V3
    glVertex2f(-75.0f, 50.0f);       // V4
    glVertex2f(-25.0f, 0.0f);        // V5
glEnd();
```

Figure 3.14. Two triangles drawn using `GL_TRIANGLES`.



NOTE

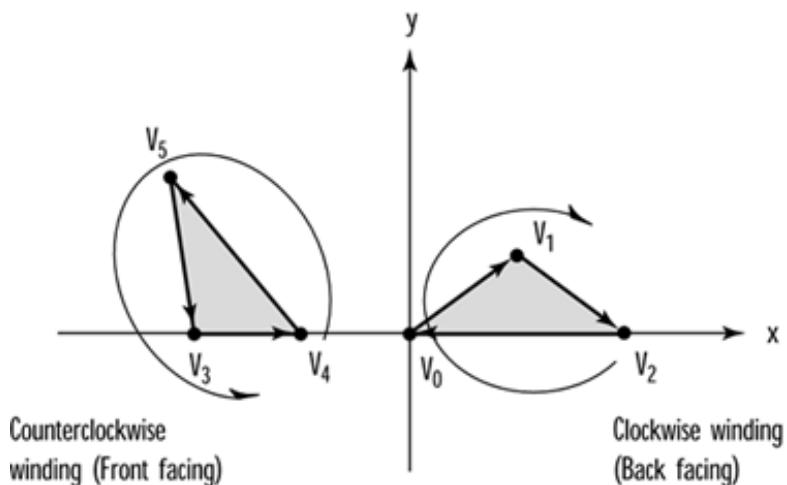
The triangles will be filled with the currently selected drawing color. If you don't specify a drawing color at some point, you can't be certain of the result.

Winding

An important characteristic of any polygonal primitive is illustrated in [Figure 3.14](#). Notice the arrows on the lines that connect the vertices. When the first triangle is drawn, the lines are drawn from V_0 to V_1 , then to V_2 , and finally back to V_0 to close the triangle. This path is in the order that the vertices are specified, and for this example, that order is clockwise from your point of view. The same directional characteristic is present for the second triangle as well.

The combination of order and direction in which the vertices are specified is called *winding*. The triangles in [Figure 3.14](#) are said to have *clockwise winding* because they are literally wound in the clockwise direction. If we reverse the positions of V_4 and V_5 on the triangle on the left, we get *counterclockwise winding*. [Figure 3.15](#) shows two triangles, each with opposite windings.

Figure 3.15. Two triangles with different windings.



OpenGL, by default, considers polygons that have counterclockwise winding to be front facing. This means that the triangle on the left in [Figure 3.15](#) shows the front of the triangle, and the one on the right shows the back side of the triangle.

Why is this issue important? As you will soon see, you will often want to give the front and back of a polygon different physical characteristics. You can hide the back of a polygon altogether or give it a different color and reflective property (see [Chapter 5](#), "Color, Materials, and Lighting: The Basics"). It's important to keep the winding of all polygons in a scene consistent, using front-facing polygons to draw the outside surface of any solid objects. In the upcoming section on solid objects, we demonstrate this principle using some models that are more complex.

If you need to reverse the default behavior of OpenGL, you can do so by calling the following function:

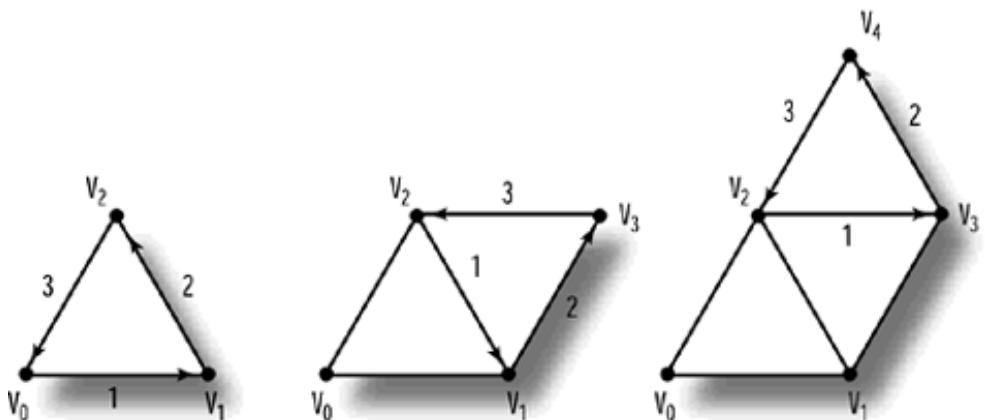
```
glFrontFace(GL_CW);
```

The `GL_CW` parameter tells OpenGL that clockwise-wound polygons are to be considered front facing. To change back to counterclockwise winding for the front face, use `GL_CCW`.

Triangle Strips

For many surfaces and shapes, you need to draw several connected triangles. You can save a lot of time by drawing a strip of connected triangles with the `GL_TRIANGLE_STRIP` primitive. [Figure 3.16](#) shows the progression of a strip of three triangles specified by a set of five vertices numbered V_0 through V_4 . Here, you see the vertices are not necessarily traversed in the same order they were specified. The reason for this is to preserve the winding (counterclockwise) of each triangle. The pattern is V_0, V_1, V_2 ; then V_2, V_1, V_3 ; then V_2, V_3, V_4 ; and so on.

Figure 3.16. The progression of a `GL_TRIANGLE_STRIP`.



For the rest of the discussion of polygonal primitives, we don't show any more code fragments to demonstrate the vertices and the `glBegin` statements. You should have the swing of things by now. Later, when we have a real sample program to work with, we resume the examples.

There are two advantages to using a strip of triangles instead of specifying each triangle separately. First, after specifying the first three vertices for the initial triangle, you need to specify only a single point for each additional triangle. This saves a lot of program or data storage space when you have many triangles to draw. The second advantage is mathematical performance and bandwidth savings. Fewer vertices mean a faster transfer from your computer's memory to your graphics card and fewer vertex transformations (see [Chapters 2 and 4](#)).

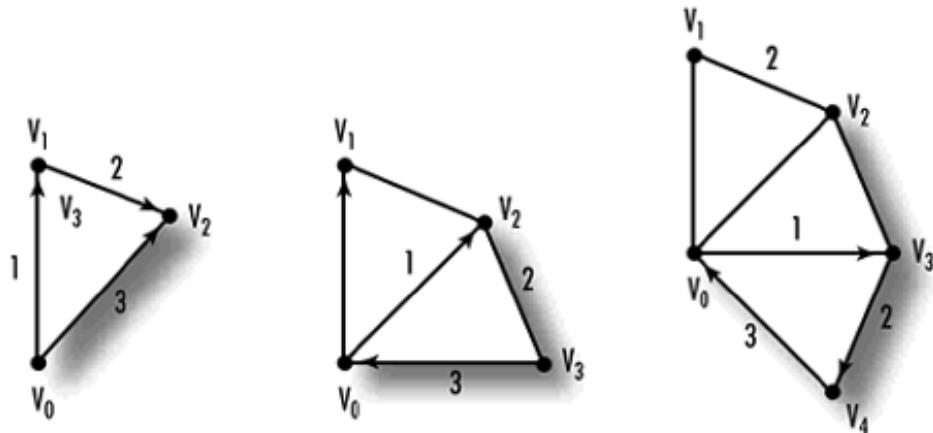
TIP

Another advantage to composing large flat surfaces out of several smaller triangles is that when lighting effects are applied to the scene, OpenGL can better reproduce the simulated effects. You'll learn more about lighting in [Chapter 5](#).

Triangle Fans

In addition to triangle strips, you can use `GL_TRIANGLE_FAN` to produce a group of connected triangles that fan around a central point. [Figure 3.17](#) shows a fan of three triangles produced by specifying four vertices. The first vertex, V_0 , forms the origin of the fan. After the first three vertices are used to draw the initial triangle, all subsequent vertices are used with the origin (V_0) and the vertex immediately preceding it (V_{n-1}) to form the next triangle.

Figure 3.17. The progression of `GL_TRIANGLE_FAN`.

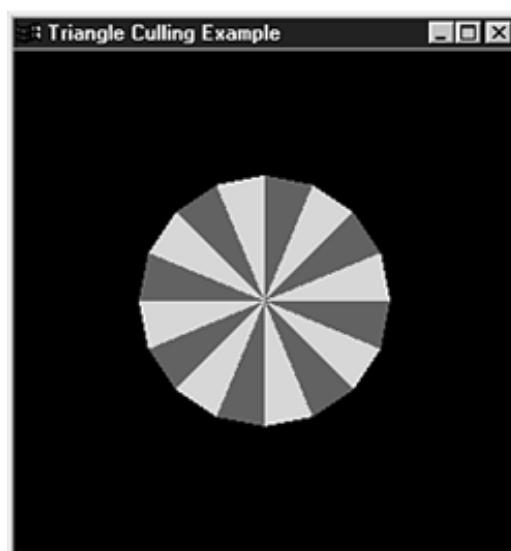


Building Solid Objects

Composing a solid object out of triangles (or any other polygon) involves more than assembling a series of vertices in a 3D coordinate space. Let's examine the sample program `TRIANGLE`, which uses two triangle fans to create a cone in our viewing volume. The first fan produces the cone shape, using the first vertex as the point of the cone and the remaining vertices as points along a circle further down the z-axis. The second fan forms a circle and lies entirely in the xy plane, making up the bottom surface of the cone.

The output from `TRIANGLE` is shown in [Figure 3.18](#). Here, you are looking directly down the z-axis and can see only a circle composed of a fan of triangles. The individual triangles are emphasized by coloring them alternately green and red.

Figure 3.18. Initial output from the `TRIANGLE` sample program.



The code for the `SetupRC` and `RenderScene` functions is shown in [Listing 3.8](#). (This listing contains some unfamiliar variables and specifiers that are explained shortly.) This program demonstrates several aspects of composing 3D objects. Right-click in the window, and you will notice an Effects menu; it will be used to enable and disable some 3D drawing features so we can explore some of the characteristics of 3D object creation. We describe these features as we progress.

Listing 3.8. `SetupRC` and `RenderScene` Code for the TRIANGLE Sample Program

```

// This function does any needed initialization on the rendering
// context.
void SetupRC()
{
    // Black background
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f );
    // Set drawing color to green
    glColor3f(0.0f, 1.0f, 0.0f);
    // Set color shading model to flat
    glShadeModel(GL_FLAT);
    // Clockwise-wound polygons are front facing; this is reversed
    // because we are using triangle fans
    glFrontFace(GL_CW);
}

// Called to draw scene
void RenderScene(void)
{
    GLfloat x,y,angle;           // Storage for coordinates and angles
    int iPivot = 1;              // Used to flag alternating colors
    // Clear the window and the depth buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // Turn culling on if flag is set
    if(bCull)
        glEnable(GL_CULL_FACE);
    else
        glDisable(GL_CULL_FACE);
    // Enable depth testing if flag is set
    if(bDepth)
        glEnable(GL_DEPTH_TEST);
    else
        glDisable(GL_DEPTH_TEST);
    // Draw the back side as a wireframe only, if flag is set
    if(bOutline)
        glPolygonMode(GL_BACK,GL_LINE);
    else
        glPolygonMode(GL_BACK,GL_FILL);

    // Save matrix state and do the rotation
    glPushMatrix();
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);
    // Begin a triangle fan
    glBegin(GL_TRIANGLE_FAN);
    // Pinnacle of cone is shared vertex for fan, moved up z-axis
    // to produce a cone instead of a circle
    glVertex3f(0.0f, 0.0f, 75.0f);
    // Loop around in a circle and specify even points along the circle
    // as the vertices of the triangle fan
    for(angle = 0.0f; angle < (2.0f*GL_PI); angle += (GL_PI/8.0f))
    {
        // Calculate x and y position of the next vertex
        x = 50.0f*sin(angle);
        y = 50.0f*cos(angle);
        glVertex3f(x, y, 0.0f);
    }
}

```

```

y = 50.0f*cos(angle);
// Alternate color between red and green
if((iPivot %2) == 0)
    glColor3f(0.0f, 1.0f, 0.0f);
else
    glColor3f(1.0f, 0.0f, 0.0f);
// Increment pivot to change color next time
iPivot++;
// Specify the next vertex for the triangle fan
glVertex2f(x, y);
}

// Done drawing fan for cone
glEnd();
// Begin a new triangle fan to cover the bottom
glBegin(GL_TRIANGLE_FAN);
// Center of fan is at the origin
glVertex2f(0.0f, 0.0f);
for(angle = 0.0f; angle < (2.0f*GL_PI); angle += (GL_PI/8.0f))
{
    // Calculate x and y position of the next vertex
    x = 50.0f*sin(angle);
    y = 50.0f*cos(angle);

    // Alternate color between red and green
    if((iPivot %2) == 0)
        glColor3f(0.0f, 1.0f, 0.0f);
    else
        glColor3f(1.0f, 0.0f, 0.0f);
    // Increment pivot to change color next time
    iPivot++;
    // Specify the next vertex for the triangle fan
    glVertex2f(x, y);
}
// Done drawing the fan that covers the bottom
glEnd();
// Restore transformations
glPopMatrix();
// Flush drawing commands
glFlush();
}

```

Setting Polygon Colors

Until now, we have set the current color only once and drawn only a single shape. Now, with multiple polygons, things get slightly more interesting. We want to use different colors so we can see our work more easily. Colors are actually specified per vertex, not per polygon. The shading model affects whether the polygon is solidly colored (using the current color selected when the last vertex was specified) or smoothly shaded between the colors specified for each vertex.

The line

```
glShadeModel(GL_FLAT);
```

tells OpenGL to fill the polygons with the solid color that was current when the polygon's last vertex was specified. This is why we can simply change the current color to red or green before specifying the next vertex in our triangle fan. On the other hand, the line

```
glShadeModel(GL_SMOOTH);
```

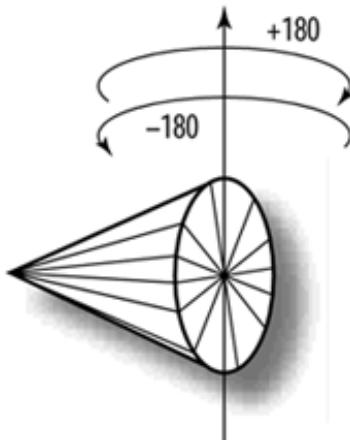
would tell OpenGL to shade the triangles smoothly from each vertex, attempting to interpolate the

colors between those specified for each vertex. You'll learn much more about color and shading in [Chapter 5](#).

Hidden Surface Removal

Hold down one of the arrow keys to spin the cone around, and don't select anything from the Effects menu yet. You'll notice something unsettling: The cone appears to be swinging back and forth plus and minus 180°, with the bottom of the cone always facing you, but not rotating a full 360°. [Figure 3.19](#) shows this effect more clearly.

Figure 3.19. The rotating cone appears to be wobbling back and forth.



This wobbling happens because the bottom of the cone is drawn after the sides of the cone are drawn. No matter how the cone is oriented, the bottom is drawn on top of it, producing the "wobbling" illusion. This effect is not limited to the various sides and parts of an object. If more than one object is drawn and one is in front of the other (from the viewer's perspective), the last object drawn still appears over the previously drawn object.

You can correct this peculiarity with a simple feature called *depth testing*. Depth testing is an effective technique for hidden surface removal, and OpenGL has functions that do this for you behind the scenes. The concept is simple: When a pixel is drawn, it is assigned a value (called the z value) that denotes its distance from the viewer's perspective. Later, when another pixel needs to be drawn to that screen location, the new pixel's z value is compared to that of the pixel that is already stored there. If the new pixel's z value is higher, it is closer to the viewer and thus in front of the previous pixel, so the previous pixel is obscured by the new pixel. If the new pixel's z value is lower, it must be behind the existing pixel and thus is not obscured. This maneuver is accomplished internally by a depth buffer with storage for a depth value for every pixel on the screen. Most all of the samples in this book use depth testing.

To enable depth testing, simply call

```
glEnable(GL_DEPTH_TEST);
```

Depth testing is enabled in [Listing 3.8](#) when the `bDepth` variable is set to `True`, and it is disabled if `bDepth` is `False`:

```
// Enable depth testing if flag is set
if(bDepth)
    glEnable(GL_DEPTH_TEST);
else
    glDisable(GL_DEPTH_TEST);
```

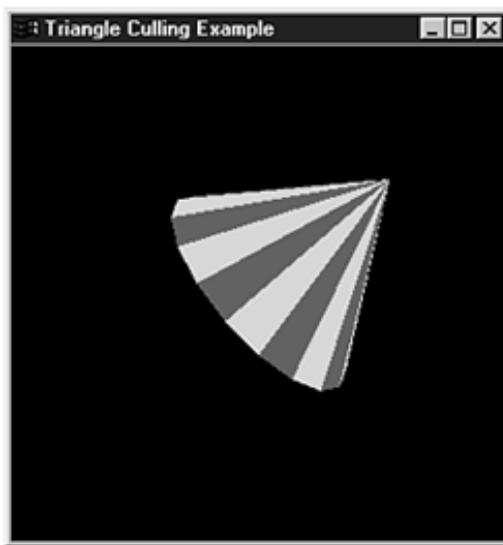
The `bDepth` variable is set when you select Depth Test from the Effects menu. In addition, the

depth buffer must be cleared each time the scene is rendered. The depth buffer is analogous to the color buffer in that it contains information about the distance of the pixels from the observer. This information is used to determine whether any pixels are hidden by pixels closer to the observer:

```
// Clear the window and the depth buffer
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

A right-click with the mouse opens a pop-up menu that allows you to toggle depth testing on and off. [Figure 3.20](#) shows the TRIANGLE program with depth testing enabled. It also shows the cone with the bottom correctly hidden behind the sides. You can see that depth testing is practically a prerequisite for creating 3D objects out of solid polygons.

Figure 3.20. The bottom of the cone is now correctly placed behind the sides for this orientation.



Culling: Hiding Surfaces for Performance

You can see that there are obvious visual advantages to not drawing a surface that is obstructed by another. Even so, you pay some performance overhead because every pixel drawn must be compared with the previous pixel's z value. Sometimes, however, you know that a surface will never be drawn anyway, so why specify it? *Culling* is the term used to describe the technique of eliminating geometry that we know will never be seen. By not sending this geometry to your OpenGL driver and hardware, you can make significant performance improvements. One culling technique is backface culling, which eliminates the backsides of a surface.

In our working example, the cone is a closed surface, and we never see the inside. OpenGL is actually (internally) drawing the back sides of the far side of the cone and then the front sides of the polygons facing us. Then, by a comparison of z buffer values, the far side of the cone is either overwritten or ignored. [Figures 3.21a](#) and [3.21b](#) show our cone at a particular orientation with depth testing turned on (a) and off (b). Notice that the green and red triangles that make up the cone sides change when depth testing is enabled. Without depth testing, the sides of the triangles at the far side of the cone show through.

Figure 3.21A. With depth testing.

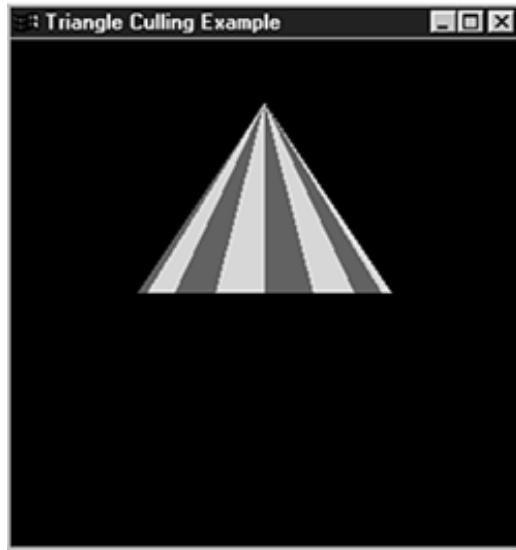
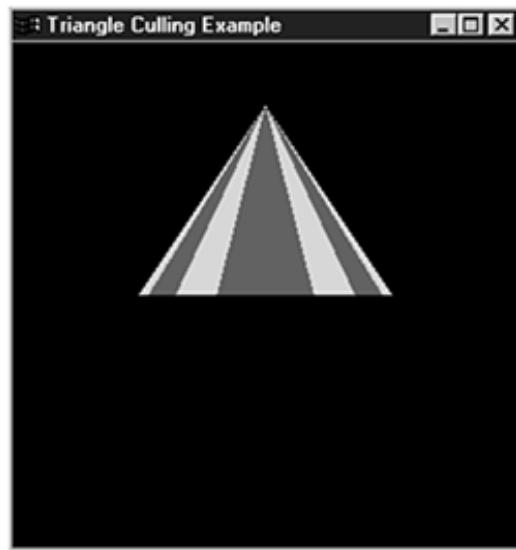


Figure 3.21B. Without depth testing.



Earlier in the chapter, we explained how OpenGL uses winding to determine the front and back sides of polygons and that it is important to keep the polygons that define the outside of our objects wound in a consistent direction. This consistency is what allows us to tell OpenGL to render only the front, only the back, or both sides of polygons. By eliminating the back sides of the polygons, we can drastically reduce the amount of necessary processing to render the image. Even though depth testing will eliminate the appearance of the inside of objects, internally OpenGL must take them into account unless we explicitly tell it not to.

Backface culling is enabled or disabled for our program by the following code from [Listing 3.8](#):

```

// Clockwise-wound polygons are front facing; this is reversed
// because we are using triangle fans
glFrontFace(GL_CW);

...
...

// Turn culling on if flag is set
if(bCull)
    glEnable(GL_CULL_FACE);
else
    glDisable(GL_CULL_FACE);

```

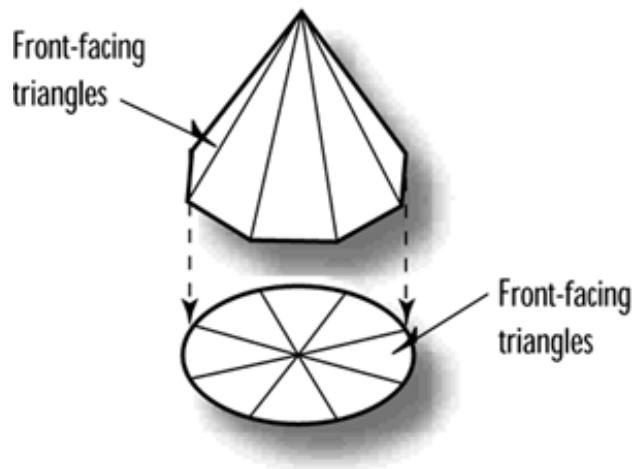
Note that we first changed the definition of front-facing polygons to assume clockwise winding (because our triangle fans are all wound clockwise).

Figure 3.22 demonstrates that the bottom of the cone is gone when culling is enabled. The reason is that we didn't follow our own rule about all the surface polygons having the same winding. The triangle fan that makes up the bottom of the cone is wound clockwise, like the fan that makes up the sides of the cone, but the front side of the cone's bottom section is then facing the inside (see **Figure 3.23**).

Figure 3.22. The bottom of the cone is culled because the front-facing triangles are inside.



Figure 3.23. How the cone was assembled from two triangle fans.



We could have corrected this problem by changing the winding rule, by calling

```
glFrontFace(GL_CCW);
```

just before we drew the second triangle fan. But in this example, we wanted to make it easy for you to see culling in action, as well as set up for our next demonstration of polygon tweaking.

WHY DO WE NEED BACKFACE CULLING?

You might wonder, "If backface culling is so desirable, why do we need the ability to turn it on and off?" Backface culling is useful when drawing closed objects or solids, but you won't always be rendering these types of geometry. Some flat objects (such as paper) can still be seen from both sides. If the cone we are drawing here were made of glass or plastic, you would actually be able to see the front and the back sides of the geometry. (See [Chapter 6](#) for a discussion of drawing transparent objects.)

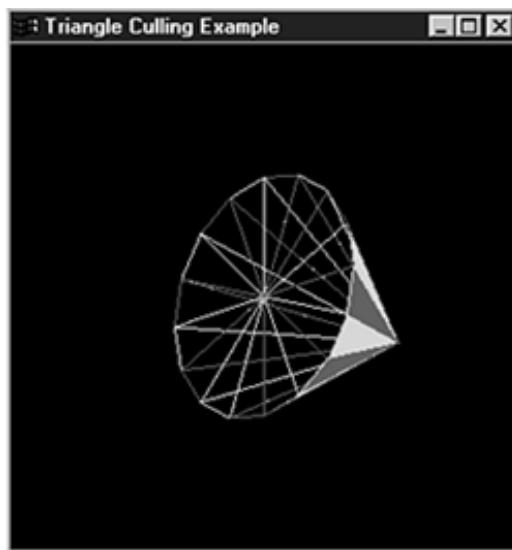
Polygon Modes

Polygons don't have to be filled with the current color. By default, polygons are drawn solid, but you can change this behavior by specifying that polygons are to be drawn as outlines or just points (only the vertices are plotted). The function `glPolygonMode` allows polygons to be rendered as filled solids, as outlines, or as points only. In addition, you can apply this rendering mode to both sides of the polygons or only to the front or back. The following code from [Listing 3.8](#) shows the polygon mode being set to outlines or solid, depending on the state of the Boolean variable `bOutline`:

```
// Draw back side as a polygon only, if flag is set
if(bOutline)
    glPolygonMode(GL_BACK,GL_LINE);
else
    glPolygonMode(GL_BACK,GL_FILL);
```

[Figure 3.24](#) shows the back sides of all polygons rendered as outlines. (We had to disable culling to produce this image; otherwise, the inside would be eliminated and you would get no outlines.) Notice that the bottom of the cone is now wireframe instead of solid, and you can see up inside the cone where the inside walls are also drawn as wireframe triangles.

Figure 3.24. Using `glPolygonMode` to render one side of the triangles as outlines.



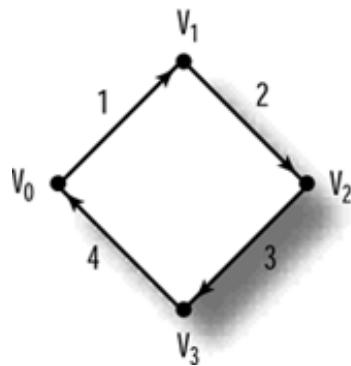
Other Primitives

Triangles are the preferred primitive for object composition because most OpenGL hardware specifically accelerates triangles, but they are not the only primitives available. Some hardware provides for acceleration of other shapes as well, and programmatically, using a general-purpose graphics primitive might be simpler. The remaining OpenGL primitives provide for rapid specification of a quadrilateral or quadrilateral strip, as well as a general-purpose polygon.

Four-Sided Polygons: Quads

If you add one more side to a triangle, you get a quadrilateral, or a four-sided figure. OpenGL's `GL_QUADS` primitive draws a four-sided polygon. In [Figure 3.25](#), a quad is drawn from four vertices. Note also that these quads have clockwise winding. One important rule to bear in mind when you use quads is that all four corners of the quadrilateral must lie in a plane (no bent quads!).

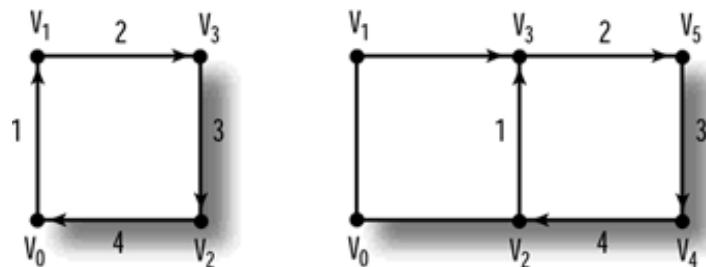
Figure 3.25. An example of `GL_QUADS`.



Quad Strips

As you can for triangle strips, you can specify a strip of connected quadrilaterals with the `GL_QUAD_STRIP` primitive. [Figure 3.26](#) shows the progression of a quad strip specified by six vertices. Note that these quad strips maintain a clockwise winding.

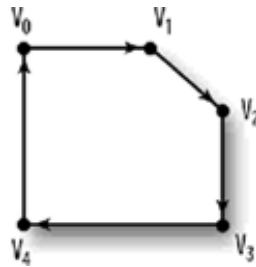
Figure 3.26. Progression of `GL_QUAD_STRIP`.



General Polygons

The final OpenGL primitive is the `GL_POLYGON`, which you can use to draw a polygon having any number of sides. [Figure 3.27](#) shows a polygon consisting of five vertices. Polygons, like quads, must have all vertices on the same plane. An easy way around this rule is to substitute `GL_TRIANGLE_FAN` for `GL_POLYGON`!

Figure 3.27. Progression of `GL_POLYGON`.



WHAT ABOUT RECTANGLES?

All 10 of the OpenGL primitives are used with `glBegin/glEnd` to draw general-purpose polygonal shapes. Although in [Chapter 2](#), we used the function `glRect` as an easy and convenient mechanism for specifying 2D rectangles, henceforth we will resort to using `GL_QUADS`.

Filling Polygons, or Stippling Revisited

There are two methods for applying a pattern to solid polygons. The customary method is texture mapping, in which an image is mapped to the surface of a polygon, and this is covered in [Chapter 8](#), "Texture Mapping: The Basics." Another way is to specify a stippling pattern, as we did for lines. A polygon stipple pattern is nothing more than a 32x32 monochrome bitmap that is used for the fill pattern.

To enable polygon stippling, call

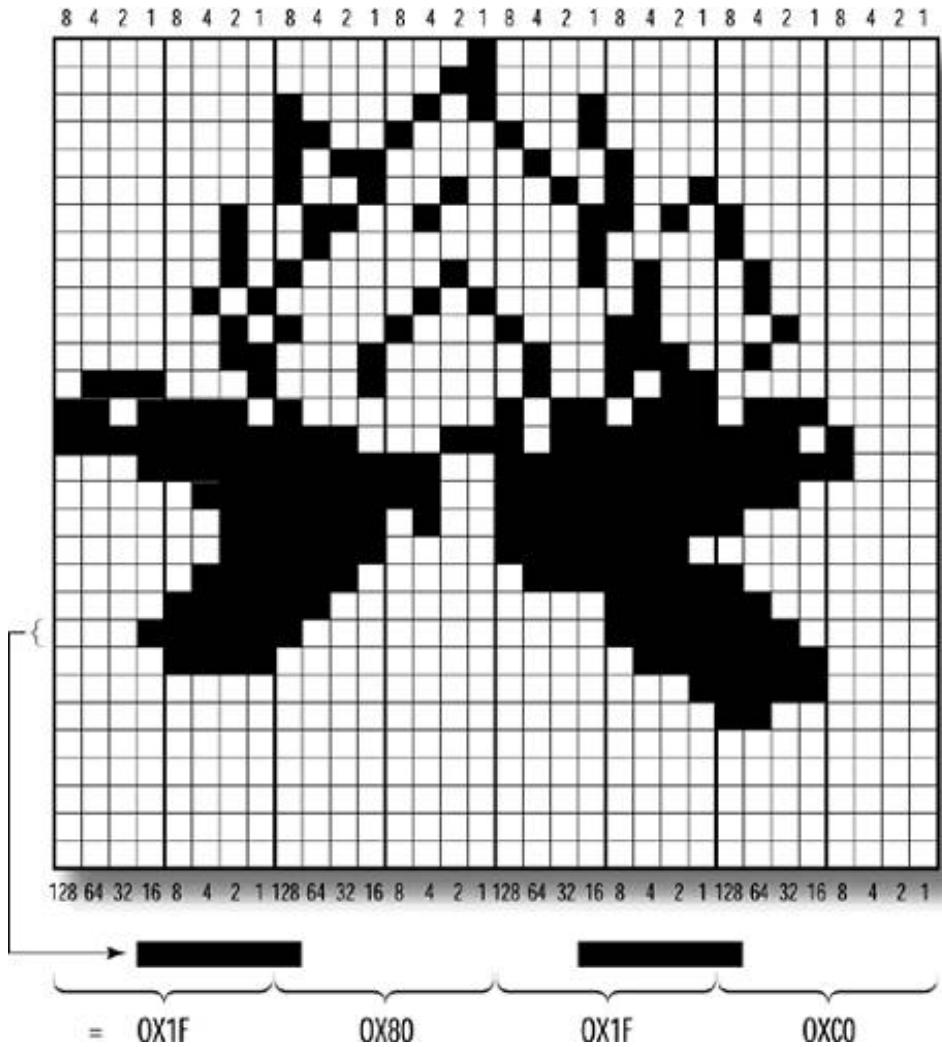
```
glEnable(GL_POLYGON_STIPPLE);
```

and then call

```
glPolygonStipple(pBitmap);
```

`pBitmap` is a pointer to a data area containing the stipple pattern. Hereafter, all polygons are filled using the pattern specified by `pBitmap` (`GLubyte *`). This pattern is similar to that used by line stippling, except the buffer is large enough to hold a 32-by-32-bit pattern. Also, the bits are read with the most significant bit (MSB) first, which is just the opposite of line stipple patterns. [Figure 3.28](#) shows a bit pattern for a campfire that we use for a stipple pattern.

Figure 3.28. Building a polygon stipple pattern.



PIXEL STORAGE

As you will learn in [Chapter 7](#), "Imaging with OpenGL," you can modify the way pixels for stipple patterns are interpreted by using the `glPixelStore` function. For now, however, we stick to the simple default polygon stippling.

To construct a mask to represent this pattern, we store one row at a time from the bottom up. Fortunately, unlike line stipple patterns, the data is, by default, interpreted just as it is stored, with the most significant bit read first. Each byte can then be read from left to right and stored in an array of `GLubyte` large enough to hold 32 rows of 4 bytes apiece.

[Listing 3.9](#) shows the code used to store this pattern. Each row of the array represents a row from [Figure 3.28](#). The first row in the array is the last row of the figure, and so on, up to the last row of the array and the first row of the figure.

Listing 3.9. Mask Definition for the Campfire in [Figure 3.28](#)

```
// Bitmap of campfire
GLubyte fire[] = { 0x00, 0x00, 0x00, 0x00,
                    0x00, 0x00, 0x00, 0x00,
```

```

0x00, 0x00, 0x00, 0xc0,
0x00, 0x00, 0x01, 0xf0,
0x00, 0x00, 0x07, 0xf0,
0x0f, 0x00, 0x1f, 0xe0,
0x1f, 0x80, 0x1f, 0xc0,
0x0f, 0xc0, 0x3f, 0x80,
0x07, 0xe0, 0x7e, 0x00,
0x03, 0xf0, 0xff, 0x80,
0x03, 0xf5, 0xff, 0xe0,
0x07, 0xfd, 0xff, 0xf8,
0x1f, 0xfc, 0xff, 0xe8,
0xff, 0xe3, 0xbf, 0x70,
0xde, 0x80, 0xb7, 0x00,
0x71, 0x10, 0x4a, 0x80,
0x03, 0x10, 0x4e, 0x40,
0x02, 0x88, 0x8c, 0x20,
0x05, 0x05, 0x04, 0x40,
0x02, 0x82, 0x14, 0x40,
0x02, 0x40, 0x10, 0x80,
0x02, 0x64, 0x1a, 0x80,
0x00, 0x92, 0x29, 0x00,
0x00, 0xb0, 0x48, 0x00,
0x00, 0xc8, 0x90, 0x00,
0x00, 0x85, 0x10, 0x00,
0x00, 0x03, 0x00, 0x00,
0x00, 0x00, 0x10, 0x00 };
```

To make use of this stipple pattern, we must first enable polygon stippling and then specify this pattern as the stipple pattern. The PSTIPPLE sample program does this and then draws an octagon using the stipple pattern. [Listing 3.10](#) shows the pertinent code, and [Figure 3.29](#) shows the output from PSTIPPLE.

Listing 3.10. Code from PSTIPPLE That Draws a Stippled Octagon

```

// This function does any needed initialization on the rendering
// context.
void SetupRC()
{
    // Black background
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f );
    // Set drawing color to red
    glColor3f(1.0f, 0.0f, 0.0f);
    // Enable polygon stippling
    glEnable(GL_POLYGON_STIPPLE);
    // Specify a specific stipple pattern
    glPolygonStipple(fire);
}

// Called to draw scene
void RenderScene(void)
{
    // Clear the window
    glClear(GL_COLOR_BUFFER_BIT);
    ...
    ...

    // Begin the stop sign shape,
    // use a standard polygon for simplicity
    glBegin(GL_POLYGON);
        glVertex2f(-20.0f, 50.0f);
        glVertex2f(20.0f, 50.0f);
        glVertex2f(50.0f, 20.0f);
```

```

glVertex2f(50.0f, -20.0f);
glVertex2f(20.0f, -50.0f);
glVertex2f(-20.0f, -50.0f);
glVertex2f(-50.0f, -20.0f);
glVertex2f(-50.0f, 20.0f);

glEnd();
...
...
// Flush drawing commands
glFlush();
}

```

Figure 3.29. Output from the PSTIPPLE program.



[Figure 3.30](#) shows the octagon rotated somewhat. Notice that the stipple pattern is still used, but the pattern is not rotated with the polygon. The stipple pattern is used only for simple polygon filling onscreen. If you need to map an image to a polygon so that it mimics the polygon's surface, you must use texture mapping (see [Chapter 8](#)).

Figure 3.30. PSTIPPLE output with the polygon rotated, showing that the stipple pattern is not rotated.

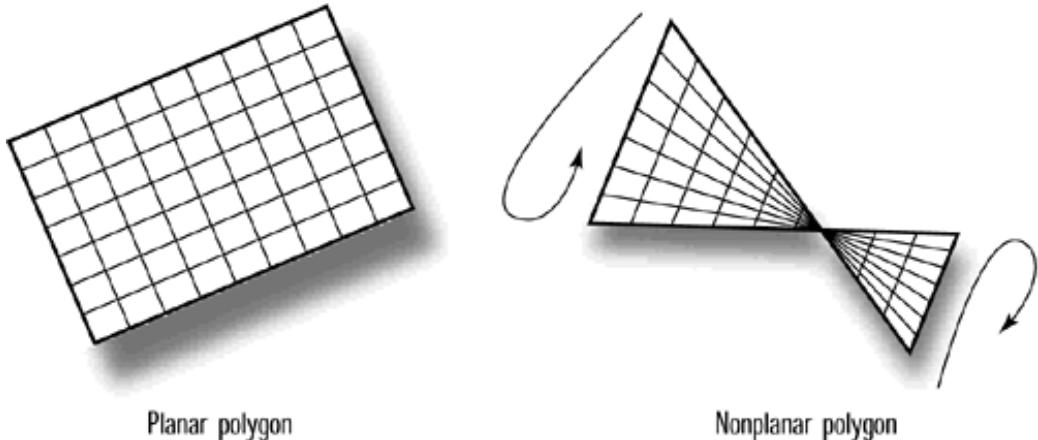


Polygon Construction Rules

When you are using many polygons to construct a complex surface, you need to remember two important rules.

The first rule is that all polygons must be planar. That is, all the vertices of the polygon must lie in a single plane, as illustrated in [Figure 3.31](#). The polygon cannot twist or bend in space.

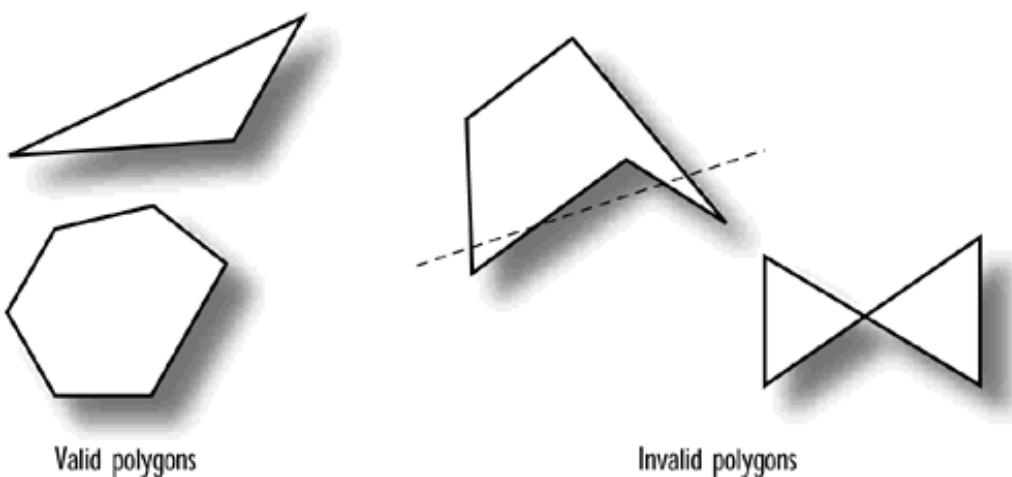
Figure 3.31. Planar versus nonplanar polygons.



Here is yet another good reason to use triangles. No triangle can ever be twisted so that all three points do not line up in a plane because mathematically it only takes exactly three points to define a plane. (If you can plot an invalid triangle, aside from winding it in the wrong direction, the Nobel Prize committee might be looking for you!)

The second rule of polygon construction is that the polygon's edges must not intersect, and the polygon must be convex. A polygon intersects itself if any two of its lines cross. *Convex* means that the polygon cannot have any indentations. A more rigorous test of a convex polygon is to draw some lines through it. If any given line enters and leaves the polygon more than once, the polygon is not convex. [Figure 3.32](#) gives examples of good and bad polygons.

Figure 3.32. Some valid and invalid primitive polygons.



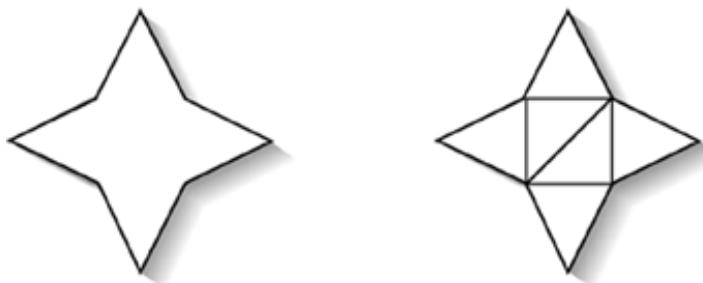
WHY THE LIMITATIONS ON POLYGONS?

You might wonder why OpenGL places the restrictions on polygon construction. Handling polygons can become quite complex, and OpenGL's restrictions allow it to use very fast algorithms for rendering these polygons. We predict that you'll not find these restrictions burdensome and that you'll be able to build any shapes or objects you need using the existing primitives. [Chapter 10](#), "Curves and Surfaces," discusses some techniques for breaking a complex shape into smaller triangles.

Subdivision and Edges

Even though OpenGL can draw only convex polygons, there's still a way to create a nonconvex polygon: by arranging two or more convex polygons together. For example, let's take a four-point star, as shown in [Figure 3.33](#). This shape is obviously not convex and thus violates OpenGL's rules for simple polygon construction. However, the star on the right is composed of six separate triangles, which are legal polygons.

Figure 3.33. A nonconvex four-point star made up of six triangles.



When the polygons are filled, you won't be able to see any edges and the figure will seem to be a single shape onscreen. However, if you use `glPolygonMode` to switch to an outline drawing, it is distracting to see all those little triangles making up some larger surface area.

OpenGL provides a special flag called an *edge flag* to address those distracting edges. By setting and clearing the edge flag as you specify a list of vertices, you inform OpenGL which line segments are considered border lines (lines that go around the border of your shape) and which ones are not (internal lines that shouldn't be visible). The `glEdgeFlag` function takes a single parameter that sets the edge flag to `True` or `False`. When the function is set to `True`, any vertices that follow mark the beginning of a boundary line segment. [Listing 3.11](#) shows an example of this from the STAR sample program on the CD.

Listing 3.11. Sample Usage of `glEdgeFlag` from the STAR Program

```
// Begin the triangles
glBegin(GL_TRIANGLES);
    glEdgeFlag(bEdgeFlag);
    glVertex2f(-20.0f, 0.0f);
    glEdgeFlag(TRUE);
    glVertex2f(20.0f, 0.0f);
    glVertex2f(0.0f, 40.0f);
    glVertex2f(-20.0f, 0.0f);
    glVertex2f(-60.0f, -20.0f);
    glEdgeFlag(bEdgeFlag);
    glVertex2f(-20.0f, -40.0f);
    glEdgeFlag(TRUE);
    glVertex2f(-20.0f, -40.0f);
```

```

glVertex2f(0.0f, -80.0f);
glEdgeFlag(bEdgeFlag);
glVertex2f(20.0f, -40.0f);
glEdgeFlag(TRUE);
glVertex2f(20.0f, -40.0f);
glVertex2f(60.0f, -20.0f);
glEdgeFlag(bEdgeFlag);
glVertex2f(20.0f, 0.0f);
glEdgeFlag(TRUE);
// Center square as two triangles
glEdgeFlag(bEdgeFlag);
glVertex2f(-20.0f, 0.0f);
glVertex2f(-20.0f,-40.0f);
glVertex2f(20.0f, 0.0f);
glVertex2f(-20.0f,-40.0f);
glVertex2f(20.0f, -40.0f);
glVertex2f(20.0f, 0.0f);
glEdgeFlag(TRUE);
// Done drawing Triangles
glEnd();

```

The Boolean variable `bEdgeFlag` is toggled on and off by a menu option to make the edges appear and disappear. If this flag is `True`, all edges are considered boundary edges and appear when the polygon mode is set to `GL_LINES`. In [Figures 3.34a](#) and [3.34b](#), you can see the output from STAR, showing the wireframe star with and without edges.

Figure 3.34A. STAR program with edges enabled.

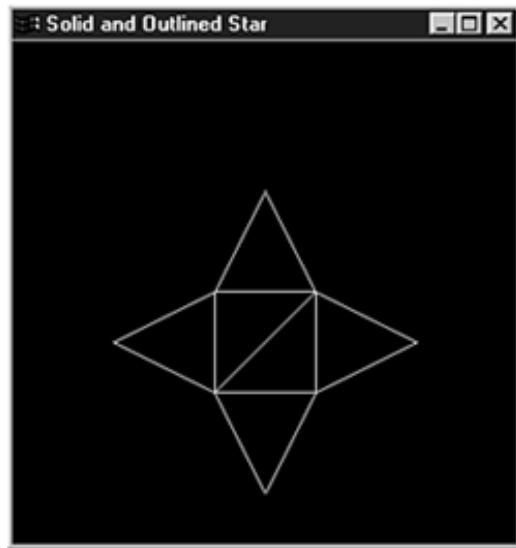
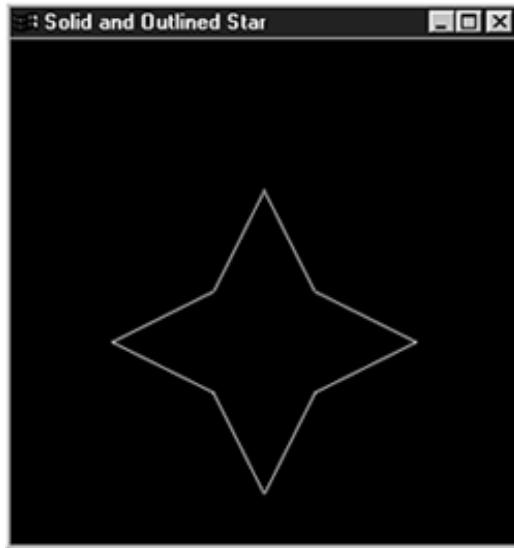


Figure 3.34B. STAR program without edges enabled.



Other Buffer Tricks

You learned from [Chapter 2](#) that OpenGL does not render (draw) these primitives directly on the screen. Instead, rendering is done in a buffer, which is later swapped to the screen. We refer to these two buffers as the front (the screen) and back color buffers. By default, OpenGL commands are rendered into the back buffer, and when you call `glutSwapBuffers` (or your operating system-specific buffer swap function), the front and back buffers are swapped so that you can see the rendering results. You can, however, render directly into the front buffer if you want. This capability can be useful for displaying a series of drawing commands so that you can see some object or shape actually being drawn. There are two ways to do this; both are discussed in the following section.

Using Buffer Targets

The first way to render directly into the front buffer is to just tell OpenGL that you want drawing to be done there. You do this by calling the following function:

```
void glDrawBuffer(GLenum mode);
```

Specifying `GL_FRONT` causes OpenGL to render to the front buffer, and `GL_BACK` moves rendering back to the back buffer. OpenGL implementations can support more than just a single front and back buffer for rendering, such as left and right buffers for stereo rendering, and auxiliary buffers. These other buffers are documented further in the reference section at the end of this chapter.

The second way to render to the front buffer is to simply not request double-buffered rendering when OpenGL is initialized. OpenGL is initialized differently on each OS platform, but with GLUT, we initialize our display mode for RGB color and double-buffered rendering with the following line of code:

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
```

To get single-buffered rendering, you simply omit the bit flag `GLUT_DOUBLE`, as shown here:

```
glutInitDisplayMode(GLUT_RGB);
```

When you do single-buffered rendering, it is important to call either `glFlush` or `glFinish` whenever you want to see the results actually drawn to screen. A buffer swap implicitly performs a flush of the pipeline and waits for rendering to complete before the swap actually occurs. We'll discuss the mechanics of this process in more detail in [Chapter 11](#), "It's All About the Pipeline:

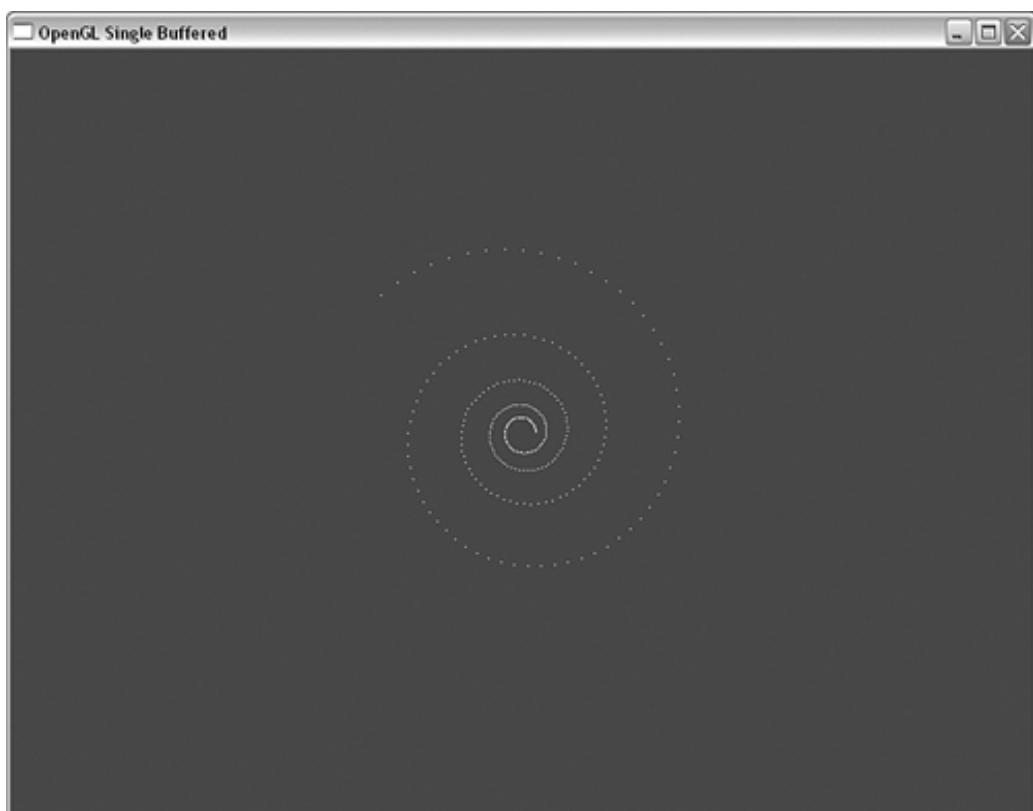
Faster Geometry Throughput."

[Listing 3.12](#) shows the drawing code for the sample program SINGLE. This example uses a single rendering buffer to draw a series of points spiraling out from the center of the window. The `RenderScene()` function is called repeatedly and uses static variables to cycle through a simple animation. The output of the SINGLE sample program is shown in [Figure 3.35](#).

Listing 3.12. Drawing Code for the SINGLE Sample

```
///////////////////////////////
// Called to draw scene
void RenderScene(void)
{
    static GLdouble dRadius = 0.1;
    static GLdouble dAngle = 0.0;
    // Clear blue window
    glClearColor(0.0f, 0.0f, 1.0f, 0.0f);
    if(dAngle == 0.0)
        glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_POINTS);
        glVertex2d(dRadius * cos(dAngle), dRadius * sin(dAngle));
    glEnd();
    dRadius *= 1.01;
    dAngle += 0.1;
    if(dAngle > 30.0)
    {
        dRadius = 0.1;
        dAngle = 0.0;
    }
    glFlush();
}
```

Figure 3.35. Output from the single-buffered rendering example.



Manipulating the Depth Buffer

The color buffers are not the only buffers that OpenGL renders into. In the preceding chapter, we mentioned other buffer targets, including the depth buffer. However, the depth buffer is filled with depth values instead of color values. Requesting a depth buffer with GLUT is as simple as adding the `GLUT_DEPTH` bit flag when initializing the display mode:

```
glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
```

You've already seen that enabling the use of the depth buffer for depth testing is as easy as calling the following:

```
glEnable(GL_DEPTH_TEST);
```

Even when depth testing is not enabled, if a depth buffer is created, OpenGL will write corresponding depth values for all color fragments that go into the color buffer.

Sometimes, though, you may want to temporarily turn off writing values to the depth buffer as well as depth testing. You can do this with the function `glDepthMask`:

```
void glDepthMask(GLboolean mask);
```

Setting the mask to `GL_FALSE` disables writes to the depth buffer but does not disable depth testing from being performed using any values that have already been written to the depth buffer. Calling this function with `GL_TRUE` re-enables writing to the depth buffer, which is the default state. Masking color writes is also possible but a bit more involved, and will be discussed in [Chapter 6](#).

Cutting It Out with Scissors

One way to improve rendering performance is to update only the portion of the screen that has changed. You may also need to restrict OpenGL rendering to a smaller rectangular region inside the window. OpenGL allows you to specify a scissor rectangle within your window where rendering can take place. By default, the scissor rectangle is the size of the window, and no scissor test takes place. You turn on the scissor test with the ubiquitous `glEnable` function:

```
glEnable(GL_SCISSOR_TEST);
```

You can, of course, turn off the scissor test again with the corresponding `glDisable` function call. The rectangle within the window where rendering is performed, called the *scissor box*, is specified in window coordinates (pixels) with the following function:

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

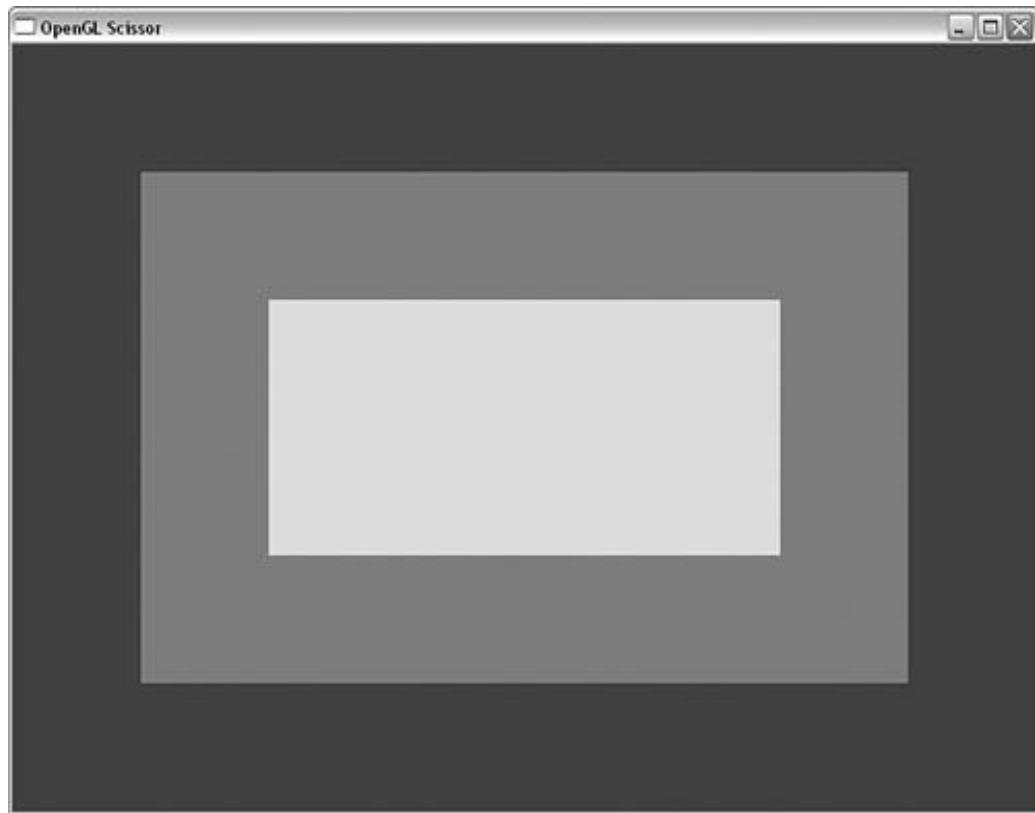
The `x` and `y` parameters specify the lower-left corner of the scissor box, with `width` and `height` being the corresponding dimensions of the scissor box. [Listing 3.13](#) shows the rendering code for the sample program SCISSOR. This program clears the color buffer three times, each time with a smaller scissor box specified before the clear. The result is a set of overlapping colored rectangles, as shown in [Figure 3.36](#).

Listing 3.13. Using the Scissor Box to Render a Series of Rectangles

```
void RenderScene(void)
{
    // Clear blue window
```

```
glClearColor(0.0f, 0.0f, 1.0f, 0.0f);
glClear(GL_COLOR_BUFFER_BIT);
// Now set scissor to smaller red sub region
glClearColor(1.0f, 0.0f, 0.0f, 0.0f);
glScissor(100, 100, 600, 400);
glEnable(GL_SCISSOR_TEST);
glClear(GL_COLOR_BUFFER_BIT);
// Finally, an even smaller green rectangle
glClearColor(0.0f, 1.0f, 0.0f, 0.0f);
glScissor(200, 200, 400, 200);
glClear(GL_COLOR_BUFFER_BIT);
// Turn scissor back off for next render
glDisable(GL_SCISSOR_TEST);
glutSwapBuffers();
}
```

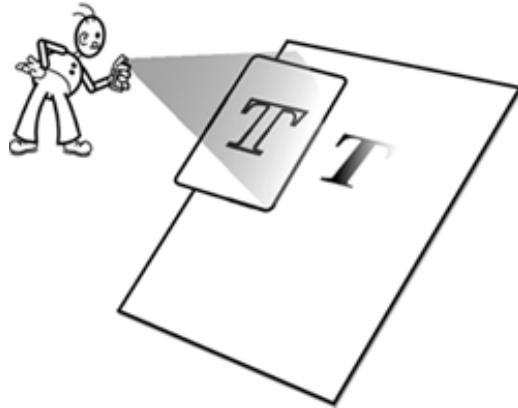
Figure 3.36. Shrinking scissor boxes.



Using the Stencil Buffer

Using the OpenGL scissor box is a great way to restrict rendering to a rectangle within the window. Frequently, however, we want to mask out an irregularly shaped area using a stencil pattern. In the real world, a stencil is a flat piece of cardboard or other material that has a pattern cut out of it. Painters use the stencil to apply paint to a surface using the pattern in the stencil. [Figure 3.37](#) shows how this process works.

Figure 3.37. Using a stencil to paint a surface in the real world.



In the OpenGL world, we have the *stencil buffer* instead. The stencil buffer provides a similar capability but is far more powerful because we can create the stencil pattern ourselves with rendering commands. To use OpenGL stenciling, we must first request a stencil buffer using the platform-specific OpenGL setup procedures. When using GLUT, we request one when we initialize the display mode. For example, the following line of code sets up a double-buffered RGB color buffer with stencil:

```
glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_STENCIL);
```

The stencil operation is relatively fast on modern hardware-accelerated OpenGL implementations, but it can also be turned on and off with `glEnable/glDisable`. For example, we turn on the stencil test with the following line of code:

```
glEnable(GL_STENCIL_TEST);
```

With the stencil test enabled, drawing occurs only at locations that pass the stencil test. You set up the stencil test that you want to use with this function:

```
void glStencilFunc(GLenum func, GLint ref, GLuint mask);
```

The stencil function that you want to use, `func`, can be any one of these values: `GL_NEVER`, `GL_ALWAYS`, `GL_LESS`, `GL_EQUAL`, `GL_GREATER`, `GL_NOTEQUAL`, `GL_LESS`, `GL_EQUAL`, `GL_GREATER`, and `GL_NOTEQUAL`. These values tell OpenGL how to compare the value already stored in the stencil buffer with the value you specify in `ref`. These values correspond to never or always passing, passing if the reference value is less than, less than or equal, greater than or equal, greater than, and not equal to the value already stored in the stencil buffer, respectively. In addition, you can specify a mask value that is bit-wise `ANDed` with both the reference value and the value from the stencil buffer before the comparison takes place.

STENCIL BITS

You need to realize that the stencil buffer may be of limited precision. Stencil buffers are typically only between 1 and 8 bits deep. Each OpenGL implementation may have its own limits on the available bit depth of the stencil buffer, and each operating system or environment has its own methods of querying and setting this value. In GLUT, you just get the most stencil bits available, but for finer-grained control, you need to refer to the operating system-specific chapters later in the book. Values passed to `ref` and `mask` that exceed the available bit depth of the stencil buffer are simply truncated, and only the maximum number of least significant bits is used.

Creating the Stencil Pattern

You now know how the stencil test is performed, but how are values put into the stencil buffer to begin with? First, we must make sure that the stencil buffer is cleared before we start any drawing operations. We do this in the same way that we clear the color and depth buffers with `glClear`—using the bit mask `GL_STENCIL_BUFFER_BIT`. For example, the following line of code clears the color, depth, and stencil buffers simultaneously:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
```

The value used in the clear operation is set previously with a call to

```
glClearStencil(GLint s);
```

When the stencil test is enabled, rendering commands are tested against the value in the stencil buffer using the `glStencilFunc` parameters we just discussed. Fragments (color values placed in the color buffer) are either written or discarded based on the outcome of that stencil test. The stencil buffer itself is also modified during this test, and what goes into the stencil buffer depends on how you've called the `glStencilOp` function:

```
void glStencilOp(GLenum fail, GLenum zfail, GLenum zpass);
```

These values tell OpenGL how to change the value of the stencil buffer if the stencil test fails (`fail`), and even if the stencil test passes, you can modify the stencil buffer if the depth test fails (`zfail`) or passes (`zpass`). The valid values for these arguments are `GL_KEEP`, `GL_ZERO`, `GL_REPLACE`, `GL_INCR`, `GL_DECR`, `GL_INVERT`, `GL_INCR_WRAP`, and `GL_DECR_WRAP`. These values correspond to keeping the current value, setting it to zero, replacing with the reference value (from `glStencilFunc`), incrementing or decrementing the value, inverting it, and incrementing/decrementing with wrap, respectively. Both `GL_INCR` and `GL_DECR` increment and decrement the stencil value but are clamped to the minimum and maximum value that can be represented in the stencil buffer for a given bit depth. `GL_INCR_WRAP` and likewise `GL_DECR_WRAP` simply wrap the values around when they exceed the upper and lower limits of a given bit representation.

In the sample program STENCIL, we create a spiral line pattern in the stencil buffer, but not in the color buffer. The bouncing rectangle from [Chapter 2](#) comes back for a visit, but this time, the stencil test prevents drawing of the red rectangle anywhere the stencil buffer contains a 0x1 value. [Listing 3.14](#) shows the relevant drawing code.

Listing 3.14. Rendering Code for the STENCIL Sample

```
void RenderScene(void)
{
    GLdouble dRadius = 0.1; // Initial radius of spiral
    GLdouble dAngle; // Looping variable
    // Clear blue window
    glClearColor(0.0f, 0.0f, 1.0f, 0.0f);
    // Use 0 for clear stencil, enable stencil test
    glClearStencil(0.0f);
    glEnable(GL_STENCIL_TEST);
    // Clear color and stencil buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
    // All drawing commands fail the stencil test, and are not
    // drawn, but increment the value in the stencil buffer.
    glStencilFunc(GL_NEVER, 0x0, 0x0);
    glStencilOp(GL_INCR, GL_INCR, GL_INCR);
    // Spiral pattern will create stencil pattern
    // Draw the spiral pattern with white lines. We
    // make the lines white to demonstrate that the
    // stencil function prevents them from being drawn
    glColor3f(1.0f, 1.0f, 1.0f);
```

```

glBegin(GL_LINE_STRIP);
    for(dAngle = 0; dAngle < 400.0; dAngle += 0.1)
    {
        glVertex2d(dRadius * cos(dAngle), dRadius * sin(dAngle));
        dRadius *= 1.002;
    }
glEnd();
// Now, allow drawing, except where the stencil pattern is 0x1
// and do not make any further changes to the stencil buffer
glStencilFunc(GL_NOTEQUAL, 0x1, 0x1);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
// Now draw red bouncing square
// (x and y) are modified by a timer function
	glColor3f(1.0f, 0.0f, 0.0f);
	glRectf(x, y, x + rsize, y - rsize);
// All done, do the buffer swap
	glutSwapBuffers();
}

```

The following two lines cause all fragments to fail the stencil test. The values of *ref* and *mask* are irrelevant in this case and are not used.

```

glStencilFunc(GL_NEVER, 0x0, 0x0);
glStencilOp(GL_INCR, GL_INCR, GL_INCR);

```

The arguments to `glStencilOp`, however, cause the value in the stencil buffer to be written (incremented actually), regardless of whether anything is seen on the screen. Following these lines, a white spiral line is drawn, and even though the color of the line is white so you can see it against the blue background, it is not drawn in the color buffer because it always fails the stencil test (`GL_NEVER`). You are essentially rendering only to the stencil buffer!

Next, we change the stencil operation with these lines:

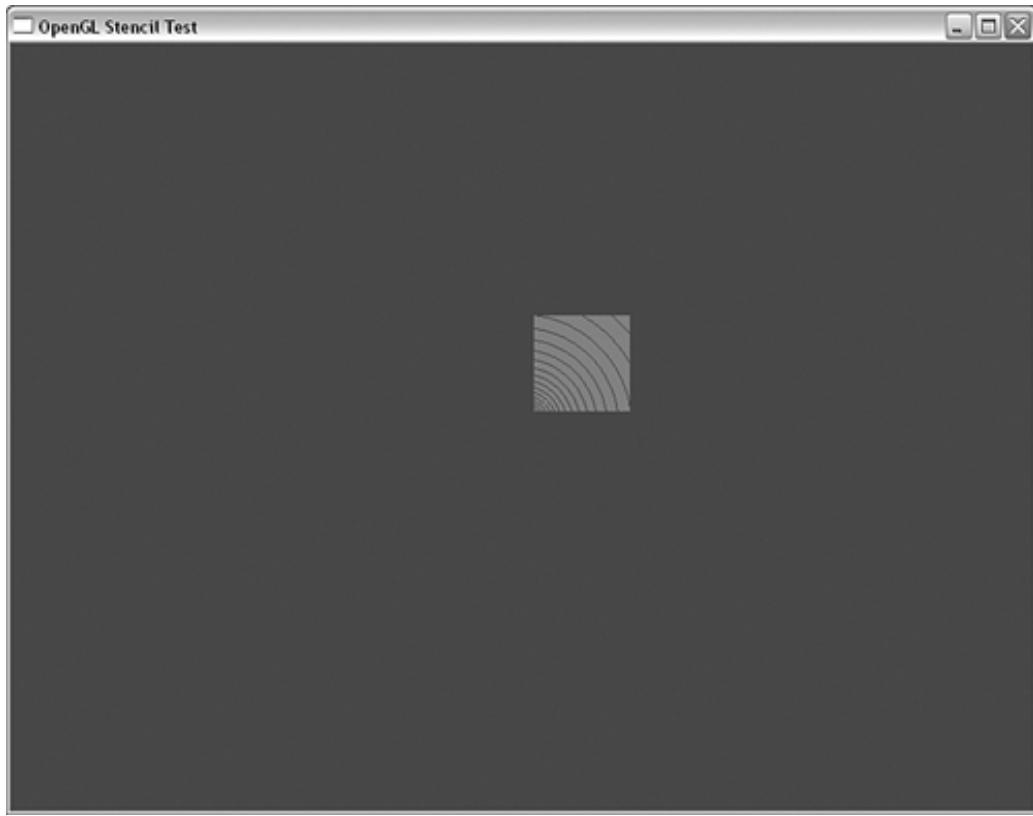
```

glStencilFunc(GL_NOTEQUAL, 0x1, 0x1);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);

```

Now, drawing will occur anywhere the stencil buffer is not equal (`GL_NOTEQUAL`) to 0x1, which is anywhere onscreen that the spiral line is not drawn. The subsequent call to `glStencilOp` is optional for this example, but it tells OpenGL to leave the stencil buffer alone for all future drawing operations. Although this sample is best seen in action, [Figure 3.38](#) shows an image of what the bounding red square looks like as it is "stenciled out."

Figure 3.38. The bouncing red square with masking stencil pattern.



Just like the depth buffer, you can also mask out writes to the stencil buffer by using the function `glStencilMask`:

```
void glStencilMask(GLboolean mask);
```

Setting the mask to `false` does not disable stencil test operations but does prevent any operation from writing values into the stencil buffer.

Summary

We covered a lot of ground in this chapter. At this point, you can create your 3D space for rendering, and you know how to draw everything from points and lines to complex polygons. We also showed you how to assemble these two-dimensional primitives as the surface of three-dimensional objects.

You also learned about some of the other buffers that OpenGL renders into besides the color buffer. As we move forward throughout the book, we will use the depth and stencil buffers for many other techniques and special effects. In [Chapter 6](#), you will learn about yet another OpenGL buffer, the Accumulation buffer. You'll see later that all these buffers working together can create some outstanding and very realistic 3D graphics.

We encourage you to experiment with what you have learned in this chapter. Use your imagination and create some of your own 3D objects before moving on to the rest of the book. You'll then have some personal samples to work with and enhance as you learn and explore new techniques throughout the book.

Reference

glBegin

Purpose: Denotes the beginning of a group of vertices that define one or more primitives.

Include File: `<gl.h>`

Syntax:

```
void glBegin(GLenum mode);
```

Description: This function is used in conjunction with `glEnd` to delimit the vertices of an OpenGL primitive. You can include multiple vertices sets within a single `glBegin`/`glEnd` pair, as long as they are for the same primitive type. You can also make other settings with additional OpenGL commands that affect the vertices following them. You can call only these OpenGL functions within a `glBegin/glEnd` sequence: `glVertex`, `glColor`, `glNormal`, `glEvalCoord`, `glCallList`, `glCallLists`, `glTexCoord`, `glEdgeFlag`, and `glMaterial`. Note that display lists (`glCallList(s)`) may only contain the other functions listed here.

Parameters:

`mode` **GLenum:** This value specifies the primitive to be constructed. It can be any of the values in [Table 3.1](#).

Table 3.1. OpenGL Primitives Supported by `glBegin`

Mode	Primitive Type
<code>GL_POINTS</code>	The specified vertices are used to create a single point each.
<code>GL_LINES</code>	The specified vertices are used to create line segments. Every two vertices specify a single and separate line segment. If the number of vertices is odd, the last one is ignored.
<code>GL_LINE_STRIP</code>	The specified vertices are used to create a line strip. After the first vertex, each subsequent vertex specifies the next point to which the line is extended.
<code>GL_LINE_LOOP</code>	This mode behaves like <code>GL_LINE_STRIP</code> , except a final line segment is drawn between the last and the first vertex specified. This is typically used to draw closed regions that might violate the rules regarding <code>GL_POLYGON</code> usage.
<code>GL_TRIANGLES</code>	The specified vertices are used to construct triangles. Every three vertices specify a new triangle. If the number of vertices is not evenly divisible by three, the extra vertices are ignored.
<code>GL_TRIANGLE_STRIP</code>	The specified vertices are used to create a strip of triangles. After the first three vertices are specified, each of any subsequent vertices is used with the two preceding ones to construct the next triangle. Each triplet of vertices (after the initial set) is automatically rearranged to ensure consistent winding of the triangles.

<code>GL_TRIANGLE_FAN</code>	The specified vertices are used to construct a triangle fan. The first vertex serves as an origin, and each vertex after the third is combined with the foregoing one and the origin. Any number of triangles may be fanned in this manner.
<code>GL_QUADS</code>	Each set of four vertices is used to construct a quadrilateral (a four-sided polygon). If the number of vertices is not evenly divisible by four, the remaining ones are ignored.
<code>GL_QUAD_STRIP</code>	The specified vertices are used to construct a strip of quadrilaterals. One quadrilateral is defined for each pair of vertices after the first pair. Unlike the vertex ordering for <code>GL_QUADS</code> , each pair of vertices is used in the reverse order specified to ensure consistent winding.
<code>GL_POLYGON</code>	The specified vertices are used to construct a convex polygon. The polygon edges must not intersect. The last vertex is automatically connected to the first vertex to ensure the polygon is closed.

Returns: None.

See Also: `glEnd`, `glVertex`

`glClearDepth`

Purpose: Specifies a depth value to be used for depth buffer clears.

Include File: `<gl.h>`

Syntax:

```
void glClearDepth(GLclampd depth);
```

Description: This function sets the depth value that is used when clearing the depth buffer with `glClear(GL_DEPTH_BUFFER_BIT)`.

Parameters:

depth `GLclampd`: The clear value for the depth buffer.

Returns: None.

See Also: `glClear`, `glDepthFunc`, `glDepthMask`, `glDepthRange`

`glClearStencil`

Purpose: Specifies a stencil value to be used for stencil buffer clears.

Include File: `<gl.h>`

Syntax:

```
void glClearStencil(GLint value);
```

Description: This function sets the stencil value that is used when clearing the stencil buffer with `glClear(GL_STENCIL_BUFFER_BIT)`.

Parameters:

`value` `GLint`: The clear value for the stencil buffer.

Returns: None.

See Also: `glStencilFunc`, `glStencilOp`

glCullFace

Purpose: Specifies whether the front or back of polygons should be eliminated from drawing.

Include File: `<gl.h>`

Syntax:

```
void glCullFace(GLenum mode);
```

Description: This function eliminates all drawing operations on either the front or back of a polygon. This eliminates unnecessary rendering computations when the back side of polygons are never visible, regardless of rotation or translation of the objects. Culling is enabled or disabled by calling `glEnable` or `glDisable` with the `GL_CULL_FACE` parameter. The front and back of the polygon are defined by use of `glFrontFace` and by the order in which the vertices are specified (clockwise or counterclockwise winding).

Parameters:

`mode` `GLenum`: Specifies which face of polygons should be culled. May be either `GL_FRONT` or `GL_BACK`.

Returns: None.

See Also: `glFrontFace`, `glLightModel`

glDepthFunc

Purpose: Specifies the depth-comparison function used against the depth buffer to decide whether color fragments should be rendered.

Include File: `<gl.h>`

Syntax:

```
void glDepthFunc(GLenum func);
```

Description: Depth testing is the primary means of hidden surface removal in OpenGL. When a color value is written to the color buffer, a corresponding depth value is written to the depth buffer. When depth testing is enabled by calling `glEnable(GL_DEPTH_TEST)`, color values are not written to the color buffer unless the corresponding depth value passes the depth test with the depth value already present. This function allows you to tweak the function used for depth buffer comparisons. The default function is `GL_LESS`.

Parameters:

func `GLenum`: Specifies which depth-comparison function to use. Valid values are listed in [Table 3.2](#).

Table 3.2. Depth Function Enumerants

Depth Function	Meaning
<code>GL_NEVER</code>	Fragments never pass the depth test.
<code>GL_LESS</code>	Fragments pass only if the incoming z value is less than the z value already present in the z buffer. This is the default value.
<code>GL_EQUAL</code>	Fragments pass only if the incoming z value is less than or equal to the z value already present in the z buffer.
<code>GL_GREATER</code>	Fragments pass only if the incoming z value is equal to the z value already present in the z buffer.
<code>GL_NOTEQUAL</code>	Fragments pass only if the incoming z value is not equal to the z value already present in the z buffer.
<code>GL_GEQUAL</code>	Fragments pass only if the incoming z value is greater than or equal to the z value already present in the z buffer.
<code>GL_ALWAYS</code>	Fragments always pass regardless of any z value.

Returns: None.

See Also: `glClearDepth`, `glDepthMask`, `glDepthRange`

glDepthMask

Purpose: Selectively allows or disallows changes to the depth buffer.

Include File: `<gl.h>`

Syntax:

```
void glDepthMask(GLBoolean flag);
```

Description: If a depth buffer is created for an OpenGL rendering context, OpenGL will calculate and store depth values. Even when depth testing is disabled, depth values are still calculated and stored in the depth buffer by default. This function allows you to selectively enable and disable writing to the depth buffer.

Parameters:

flag **GLboolean**: When this parameter is set to **GL_TRUE**, depth buffer writes are enabled (default). Setting this parameter to **GL_FALSE** disables changes to the depth buffer.

Returns: None.

See Also: `glClearDepth`, `glDepthFunc`, `glDepthRange`

glDepthRange

Purpose: Allows a mapping of z values to window coordinates from normalized device coordinates.

Include File: `<gl.h>`

Syntax:

```
void glDepthRange(GLclampd zNear, GLclampd zFar);
```

Description: Normally, depth buffer values are stored internally in the range from -1.0 to 1.0 . This function allows a specific linear mapping of depth values to a normalized range of window z coordinates. The default range for window z values is 0 to 1 corresponding to the near and far clipping planes.

Parameters:

zNear **GLclampd**: A clamped value that represents the nearest possible window z value.

zFar **GLclampd**: A clamped value that represents the largest possible window z value.

Returns: None.

See Also: `glClearDepth`, `glDepthMask`, `glDepthFunc`

glDrawBuffer

Purpose: Redirects OpenGL rendering to a specific color buffer.

Include File: `<gl.h>`

Syntax:

```
void glDrawBuffer(GLenum mode);
```

Description: By default, OpenGL renders to the back color buffer for double-buffered rendering contexts and to the front for single-buffered rendering contexts. This function allows you to direct OpenGL rendering to any available color buffer. Note that many implementations do not support left and right (stereo) or auxiliary color buffers. In the case of stereo contexts, the modes that omit references to the left and right channels will render to both channels. For example, specifying `GL_FRONT` for a stereo context will actually render to both the left and right front buffers, and `GL_FRONT_AND_BACK` will render to up to four buffers simultaneously.

Parameters:

`mode` `GLenum`: A constant flag that specifies which color buffer should be the render target. Valid values for this parameter are listed in [Table 3.3](#).

Table 3.3. Color Buffer Destinations

Constant	Description
<code>GL_NONE</code>	Do not write anything to any color buffer.
<code>GL_FRONT_LEFT</code>	Write only to the front-left color buffer.
<code>GL_FRONT_RIGHT</code>	Write only to the front-right color buffer.
<code>GL_BACK_LEFT</code>	Write only to the back-left color buffer.
<code>GL_BACK_RIGHT</code>	Write only to the back-right color buffer.
<code>GL_FRONT</code>	Write only to the front color buffer. This is the default value for single-buffered rendering contexts.
<code>GL_BACK</code>	Write only to the back color buffer. This is the default value for double-buffered rendering contexts.
<code>GL_LEFT</code>	Write only to the left color buffer.
<code>GL_RIGHT</code>	Write only to the right color buffer.
<code>GL_FRONT_AND_BACK</code>	Write to both the front and back color buffers.
<code>GL_AUX<i>i</i></code>	Write only to the auxiliary buffer <i>i</i> , with <i>i</i> being a value between 0 and <code>GL_AUX_BUFFERS - 1</code> .

Returns: None.

See Also: `glClear`, `glColorMask`

glEdgeFlag

Purpose: Flags polygon edges as either boundary or nonboundary edges. You can use this to determine whether interior surface lines are visible.

Include File: `<gl.h>`

Variations:

```
void glEdgeFlag(GLboolean flag);
void glEdgeFlagv(const GLboolean *flag);
```

Description: When two or more polygons are joined to form a larger region, the edges on the outside define the boundary of the newly formed region. This function flags inside edges as nonboundary. This is used only when the polygon mode is set to either `GL_LINE` or `GL_POINT`.

Parameters:

`flag` `GLboolean`: Sets the edge flag to this value, `True` or `False`.

`*flag` `const GLboolean *`: A pointer to a value that is used for the edge flag.

Returns: None.

See Also: `glBegin`, `glPolygonMode`

glEnd

Purpose: Terminates a list of vertices that specify a primitive initiated by `glBegin`.

Include File: `<gl.h>`

Syntax:

```
void glEnd();
```

Description: This function is used in conjunction with `glBegin` to delimit the vertices of an OpenGL primitive. You can include multiple vertices sets within a single `glBegin`/`glEnd` pair, as long as they are for the same primitive type. You can also make other settings with additional OpenGL commands that affect the vertices following them. You can call only these OpenGL functions within a `glBegin/glEnd` sequence: `glVertex`, `glColor`, `glIndex`, `glNormal`, `glEvalCoord`, `glCallList`, `glCallLists`, `glTexCoord`, `glEdgeFlag`, and `glMaterial`.

Returns: None.

See Also: `glBegin`

glFrontFace

Purpose: Defines which side of a polygon is the front or back.

Include File: `<gl.h>`

Syntax:

```
void glFrontFace(GLenum mode);
```

Description: When a scene is made up of objects that are closed (you cannot see the inside), color or lighting calculations on the inside of the object are unnecessary. The `glCullFace` function turns off such calculations for either the front or back of polygons. The `glFrontFace` function determines which side of the polygons is considered the front. If the vertices of a polygon as viewed from the front are specified so that they travel clockwise around the polygon, the polygon is said to have clockwise winding. If the vertices travel counterclockwise, the polygon is said to have counterclockwise winding. This function allows you to specify either the clockwise or counterclockwise wound face to be the front of the polygon.

Parameters:

`mode` `GLenum`: Specifies the orientation of front-facing polygons: clockwise (`GL_CW`) or counterclockwise (`GL_CCW`).

Returns: None.

See Also: `glCullFace`, `glLightModel`, `glPolygonMode`, `glMaterial`

glGetPolygonStipple

Purpose: Returns the current polygon stipple pattern.

Include File: `<gl.h>`

Syntax:

```
void glGetPolygonStipple(GLubyte *mask);
```

Description: This function copies a 32-by-32-bit pattern that represents the polygon stipple pattern into a user-specified buffer. The pattern is copied to the memory location pointed to by `mask`. The packing of the pixels is affected by the last call to `glPixelStore`.

Parameters:

`*mask` `GLubyte`: A pointer to the place where the polygon stipple pattern is to be copied.

Returns: None.

See Also: `glPolygonStipple`, `glLineStipple`, `glPixelStore`

glLineStipple

Purpose: Specifies a line stipple pattern for line-based primitives `GL_LINES`, `GL_LINE_STRIP`, and `GL_LINE_LOOP`.

Include File: `<gl.h>`

Syntax:

```
void glLineStipple(GLint factor, GLushort pattern);
```

Description: This function uses the bit pattern to draw stippled (dotted and dashed) lines. The bit pattern begins with bit 0 (the rightmost bit), so the actual drawing pattern is the reverse of what is specified. The *factor* parameter is used to widen the number of pixels drawn or not drawn along the line specified by each bit in *pattern*. By default, each bit in the pattern specifies one pixel. To use line stippling, you must first enable stippling by calling

```
glEnable(GL_LINE_STIPPLE);
```

Line stippling is disabled by default. If you are drawing multiple line segments, the pattern is reset for each new segment. That is, if a line segment is drawn such that it terminates halfway through the pattern, the next specified line segment is unaffected.

Parameters:

factor **GLint**: Specifies a multiplier that determines how many pixels will be affected by each bit in the *pattern* parameter. Thus, the pattern width is multiplied by this value. The default value is 1, and the maximum value is clamped to 255.

pattern **GLushort**: Sets the 16-bit stippling pattern. The least significant bit (bit 0) is used first for the stippling pattern. The default pattern is all 1s.

Returns: None.

See Also: [glPolygonStipple](#)

glLineWidth

Purpose: Sets the width of lines drawn with **GL_LINES**, **GL_LINE_STRIP**, or **GL_LINE_LOOP**.

Include File: `<gl.h>`

Syntax:

```
void glLineWidth(GLfloat width);
```

Description: This function sets the width in pixels of lines drawn with any of the line-based primitives.

You can get the current line width setting by calling

```
GLfloat fSize;
...
glGetFloatv(GL_LINE_WIDTH, &fSize);
```

The current line width setting will be returned in *fSize*. In addition, you can find the minimum and maximum supported line widths by calling

```
GLfloat fSizes[2];
...
glGetFloatv(GL_LINE_WIDTH_RANGE, fSizes);
```

In this instance, the minimum supported line width will be returned in `fSizes[0]`, and the maximum supported width will be stored in `fSizes[1]`. Finally, you can find the smallest supported increment between line widths by calling

```
GLfloat fStepSize;
...
glGetFloatv(GL_LINE_WIDTH_GRANULARITY, &fStepSize);
```

For any implementation of OpenGL, the only line width guaranteed to be supported is 1.0. For the Microsoft Windows generic implementation, the supported line widths range from 0.5 to 10.0, with a granularity of 0.125.

Parameters:

`width` `GLfloat`: Sets the width of lines that are drawn with the line primitives. The default value is 1.0.

Returns: None.

See Also: `glPointSize`

glPointSize

Purpose: Sets the point size of points drawn with `GL_POINTS`.

Include File: `<gl.h>`

Syntax:

```
void glPointSize(GLfloat size);
```

Description: This function sets the diameter in pixels of points drawn with the `GL_POINTS` primitive. You can get the current pixel size setting by calling

```
GLfloat fSize;
...
glGetFloatv(GL_POINT_SIZE, &fSize);
```

The current pixel size setting will be returned in `fSize`. In addition, you can find the minimum and maximum supported pixel sizes by calling

```
GLfloat fSizes[2];
...
glGetFloatv(GL_POINT_SIZE_RANGE, fSizes);
```

In this instance, the minimum supported point size will be returned in `fSizes[0]`, and the maximum supported size will be stored in `fSizes[1]`. Finally, you can find the smallest supported increment between pixel sizes by calling

```

GLfloat fStepSize;
...
glGetFloatv(GL_POINT_SIZE_GRANULARITY, &fStepSize);

```

For any implementation of OpenGL, the only point size guaranteed to be supported is 1.0. For the Microsoft Windows generic implementation, the point sizes range from 0.5 to 10.0, with a granularity of 0.125.

Parameters:

size **GLfloat**: Sets the diameter of drawn points. The default value is 1.0.

Returns: None.

See Also: [glLineWidth](#)

glPolygonMode

Purpose: Sets the rasterization mode used to draw polygons.

Include File: `<gl.h>`

Syntax:

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

Description: This function allows you to change how polygons are rendered. By default, polygons are filled or shaded with the current color or material properties. However, you may also specify that only the outlines or only the vertices are drawn. Furthermore, you may apply this specification to the front, back, or both sides of polygons.

Parameters:

face **GLenum**: Specifies which face of polygons is affected by the mode change: **GL_FRONT**, **GL_BACK**, or **GL_FRONT_AND_BACK**.

mode **GLenum**: Specifies the new drawing mode. **GL_FILL** is the default, producing filled polygons. **GL_LINE** produces polygon outlines, and **GL_POINT** plots only the points of the vertices. The lines and points drawn by **GL_LINE** and **GL_POINT** are affected by the edge flag set by [glEdgeFlag](#).

Returns: None.

See Also: [glEdgeFlag](#), [glLineStipple](#), [glLineWidth](#), [glPointSize](#), [glPolygonStipple](#)

glPolygonOffset

Purpose: Sets the scale and units used to calculate depth values.

Include File: `<gl.h>`

Syntax:

```
void glPolygonOffset(GLfloat factor, GLfloat units);
```

Description: This function allows you to add or subtract an offset to a fragment's calculated depth value. The amount of offset is calculated by the following formula:

```
offset = (m * factor) + (r * units)
```

The value of m is calculated by OpenGL and represents the maximum depth slope of the polygon. The r value is the minimum value that will create a resolvable difference in the depth buffer, and is implementation dependent. Polygon offset applies only to polygons but affects lines and points when rendered as a result of a call to `glPolygonMode`. You enable or disable the offset value for each mode by using `glEnable/glDisable` with `GL_POLYGON_OFFSET_FILL`, `GL_POLYGON_OFFSET_LINE`, or `GL_POLYGON_OFFSET_POINT`.

Parameters:

`factor` `GLfloat`: Scaling factor used to create a depth buffer offset for each polygon. Zero by default.

`units` `GLfloat`: Value multiplied by an implementation-specific value to create a depth offset. Zero by default.

Returns: None.

See Also: `glDepthFunc`, `glDepthRange`, `glPolygonMode`

glPolygonStipple

Purpose: Sets the pattern used for polygon stippling.

Include File: `<gl.h>`

Syntax:

```
void glPolygonStipple(const GLubyte *mask );
```

Description: You can use a 32-by-32-bit stipple pattern for filled polygons by using this function and enabling polygon stippling by calling `glEnable` (`GL_POLYGON_STIPPLE`). The 1s in the stipple pattern are filled with the current color, and 0s are not drawn.

Parameters:

`*mask` `const GLubyte`: Points to a 32-by-32-bit storage area that contains the stipple pattern. The packing of bits within this storage area is affected by `glPixelStore`. By default, the MSB (most significant bit) is read first when determining the pattern.

Returns: None.

See Also: `glLineStipple`, `glGetPolygonStipple`, `glPixelStore`

glScissor

Purpose: Defines a scissor box in window coordinates outside of which no drawing occurs.

Include File: `<gl.h>`

Syntax:

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

Description: This function defines a rectangle in window coordinates called the scissor box. When the scissor test is enabled with `glEnable(GL_SCISSOR_TEST)`, OpenGL drawing commands and buffer operations occur only within the scissor box.

Parameters:

`x, y` **GLint:** The lower-left corner of the scissor box, in window coordinates.

`width` **GLint:** The width of the scissor box in pixels.

`height` **GLint:** The height of the scissor box in pixels.

Returns: None.

See Also: `glStencilFunc`, `glStencilOp`

glStencilFunc

Purpose: Sets the comparison function, reference value, and mask for a stencil test.

Include File: `<gl.h>`

Syntax:

```
void glStencilFunc(GLenum func, GLint ref, GLuint  
  mask);
```

Description: When the stencil test is enabled using `glEnable(GL_STENCIL_TEST)`, the stencil function is used to determine whether a color fragment should be discarded or kept (drawn). The value in the stencil buffer is compared to the reference value `ref`, using the comparison function specified by `func`. Both the reference value and stencil buffer value may be bitwise **ANDed** with the mask. Valid comparison functions are given in [Table 3.4](#). The result of the stencil test also causes the stencil buffer to be modified according to the behavior specified in the function `glStencilOp`.

Parameters:

`func` **GLenum:** Stencil comparison function. Valid values are listed in [Table 3.4](#).

`ref` **GLint:** Reference value against which the stencil buffer value is compared.

`mask` **GLuint:** Binary mask value applied to both the reference and stencil buffer values.

Table 3.4. Stencil Test Comparison Functions

Constant	Meaning
<code>GL_NEVER</code>	Never pass the stencil test.
<code>GL_ALWAYS</code>	Always pass the stencil test.
<code>GL_LESS</code>	Pass only if the reference value is less than the stencil buffer value.
<code>GL_LEQUAL</code>	Pass only if the reference value is less than or equal to the stencil buffer value.
<code>GL_EQUAL</code>	Pass only if the reference value is equal to the stencil buffer value.
<code>GL_GEQUAL</code>	Pass only if the reference value is greater than or equal to the stencil buffer value.
<code>GL_GREATER</code>	Pass only if the reference value is greater than the stencil buffer value.
<code>GL_NOTEQUAL</code>	Pass only if the reference value is not equal to the stencil buffer value.

Returns: None.

See Also: `glStencilOp`, `glClearStencil`

glStencilMask

Purpose: Selectively allows or disallows changes to the stencil buffer.

Include File: `<gl.h>`

Syntax:

```
void glStencilMask(GLboolean flag);
```

Description: If a stencil buffer is created for an OpenGL rendering context, OpenGL will calculate, store, and make use of stencil values when stenciling is enabled. When the stencil test is disabled, values are still calculated and stored in the stencil buffer. This function allows you to selectively enable and disable writing to the stencil buffer.

Parameters:

flag `GLboolean`: When this parameter is set to `GL_TRUE`, stencil buffer writes are enabled (default). Setting this parameter to `GL_FALSE` disables changes to the depth buffer.

Returns: None.

See Also: `glStencilOp`, `glClearStencil`, `glStencilMask`

glStencilOp

Purpose: Specifies what action to take in regards to the stored value in the stencil buffer for a rendered fragment.

Include File: `<gl.h>`

Syntax:

```
void glStencilOp(GLenum sfail, GLenum zfail,
    ➔ GLenum zpass);
```

Description: This function describes what action to take when a fragment fails the stencil test. Even when fragments do not pass the stencil test and are not produced in the color buffer, the stencil buffer can still be modified by setting the appropriate action for the `sfail` parameter. In addition, even when the stencil test passes, the `zfail` and `zpass` parameters describe the desired action based on that fragment's depth buffer test. The valid actions and their constants are given in [Table 3.5](#).

Parameters:

<code>sfail</code>	GLenum: Operation to perform when the stencil test fails.
<code>zfail</code>	GLenum: Operation to perform when the depth test fails.
<code>zPass</code>	GLenum: Operation to perform when the depth test passes.

Table 3.5. Stencil Operation Constants

Constant	Description
<code>GL_KEEP</code>	Keep the current stencil buffer value.
<code>GL_ZERO</code>	Set the current stencil buffer value to 0.
<code>GL_REPLACE</code>	Replace the stencil buffer value with the reference value specified in <code>glStencilFunc</code> .
<code>GL_INCR</code>	Increment the stencil buffer value. Clamped to the bit range of the stencil buffer.
<code>GL_DECR</code>	Decrement the stencil buffer value. Clamped to the bit range of the stencil buffer.
<code>GL_INVERT</code>	Bitwise-invert the current stencil buffer value.
<code>GL_INCR_WRAP</code>	Increment the current stencil buffer value. When the maximum representable value for the given stencil buffer's bit depth is reached, the value wraps back to 0.
<code>GL_DECR_WRAP</code>	Decrement the current stencil buffer value. When the value is decremented below 0, the stencil buffer value wraps to the highest possible positive representation for its bit depth.

Returns: None.

See Also: `glStencilFunc`, `glClearStencil`

glVertex

Purpose: Specifies the 3D coordinates of a vertex.

Include File: `<gl.h>`

Variations:

```

void glVertex2d(GLdouble x, GLdouble y);
void glVertex2f(GLfloat x, GLfloat y);
void glVertex2i(GLint x, GLint y);
void glVertex2s(GLshort x, GLshort y);
void glVertex3d(GLdouble x, GLdouble y, GLdouble z);
void glVertex3f(GLfloat x, GLfloat y, GLfloat z);
void glVertex3i(GLint x, GLint y, GLint z);
void glVertex3s(GLshort x, GLshort y, GLshort z);
void glVertex4d(GLdouble x, GLdouble y, GLdouble z
  ↪ , GLdouble w);
void glVertex4f(GLfloat x, GLfloat y, GLfloat z,
  ↪ GLfloat w);
void glVertex4i(GLint x, GLint y, GLint z, GLint w);
void glVertex4s(GLshort x, GLshort y, GLshort z,
  ↪ GLshort w);
void glVertex2dv(const GLdouble *v);
void glVertex2fv(const GLfloat *v);
void glVertex2iv(const GLint *v);
void glVertex2sv(const GLshort *v);
void glVertex3dv(const GLdouble *v);
void glVertex3fv(const GLfloat *v);
void glVertex3iv(const GLint *v);
void glVertex3sv(const GLshort *v);
void glVertex4dv(const GLdouble *v);
void glVertex4fv(const GLfloat *v);
void glVertex4iv(const GLint *v);
void glVertex4sv(const GLshort *v);

```

Description: This function is used to specify the vertex coordinates of the points, lines, and polygons specified by a previous call to `glBegin`. You cannot call this function outside the scope of a `glBegin/glEnd` pair.

Parameters:

`x, y, z` The `x`, `y`, and `z` coordinates of the vertex. When `z` is not specified, the default value is 0.0.

`w` The `w` coordinate of the vertex. This coordinate is used for scaling purposes and by default is set to 1.0. Scaling occurs by dividing the other three coordinates by this value.

`*v` An array of values that contain the two, three, or four values needed to specify the vertex.

Returns: None.

See Also: `glBegin`, `glEnd`

Chapter 4. Geometric Transformations: The Pipeline

by Richard S. Wright, Jr.

WHAT YOU'LL LEARN IN THIS CHAPTER:

How To	Functions You'll Use
Establish your position in the scene	<code>gluLookAt</code>
Position objects within the scene	<code>glTranslate/glRotate</code>
Scale objects	<code>glScale</code>
Establish a perspective transformation	<code>gluPerspective</code>
Perform your own matrix transformations	<code>glLoadMatrix/glMultMatrix</code>
Use a camera to move around in a scene	<code>gluLookAt</code>

In [Chapter 3](#), "Drawing in Space: Geometric Primitives and Buffers," you learned how to draw points, lines, and various primitives in 3D. To turn a collection of shapes into a coherent scene, you must arrange them in relation to one another and to the viewer. In this chapter, you start moving shapes and objects around in your coordinate system. (Actually, you don't move the objects, but rather shift the coordinate system to create the view you want.) The ability to place and orient your objects in a scene is a crucial tool for any 3D graphics programmer. As you will see, it is actually convenient to describe your objects' dimensions around the origin and then *transform* the objects into the desired position.

Is This the Dreaded Math Chapter?

In most books on 3D graphics programming, yes, this would be the dreaded math chapter. However, you can relax; we take a more moderate approach to these principles than some texts.

The keys to object and coordinate transformations are two matrices maintained by OpenGL. To familiarize you with these matrices, this chapter strikes a compromise between two extremes in computer graphics philosophy. On one hand, we could warn you, "Please review a textbook on linear algebra before reading this chapter." On the other hand, we could perpetuate the deceptive reassurance that you can "learn to do 3D graphics without all those complex mathematical formulas." But we don't agree with either camp.

In reality, you can get along just fine without understanding the finer mathematics of 3D graphics, just as you can drive your car every day without having to know anything at all about automotive mechanics and the internal combustion engine. But you had better know enough about your car to realize that you need an oil change every so often, that you have to fill the tank with gas regularly, and that you must change the tires when they get bald. This knowledge makes you a responsible (and safe!) automobile owner. If you want to be a responsible and capable OpenGL programmer, the same standards apply. You need to understand at least the basics so you know what can be done and what tools best suit the job. If you are a beginner, you will find that, with some practice, matrix math and vectors will gradually make more and more sense, and you will develop a more intuitive (and powerful) ability to make full use of the concepts we introduce in this chapter.

So, even if you don't already have the ability to multiply two matrices in your head, you need to know what matrices are and that they are the means to OpenGL's 3D magic. But before you go dusting off that old linear algebra textbook (doesn't everyone have one?), have no fear: OpenGL does all the math for you. Think of using OpenGL as using a calculator to do long division when you don't know how to do it on paper. Although you don't have to do it yourself, you still know

what it is and how to apply it. See—you can eat your cake and have it, too!

Understanding Transformations

If you think about it, most 3D graphics aren't really 3D. We use 3D concepts and terminology to describe what something looks like; then this 3D data is "squished" onto a 2D computer screen. We call the process of squishing 3D data down into 2D data *projection*, and we introduced both orthographic and perspective projections back in [Chapter 1](#), "Introduction to 3D Graphics and OpenGL." We refer to the projection whenever we want to describe the type of transformation (orthographic or perspective) that occurs during projection, but projection is only one of the different types of transformations that occurs in OpenGL. Transformations also allow you to rotate objects around; move them about; and even stretch, shrink, and warp them.

Three types of geometric transformations occur between the time you specify your vertices and the time they appear on the screen: viewing, modeling, and projection. In this section, we examine the principles of each type of transformation, which are summarized in [Table 4.1](#).

Table 4.1. Summary of the OpenGL Transformation Terminology

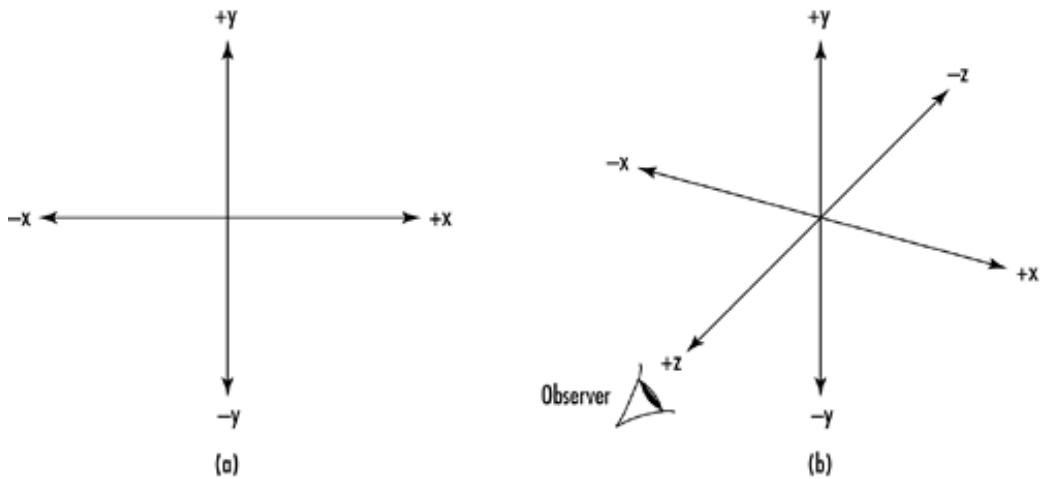
Transformation	Use
Viewing	Specifies the location of the viewer or camera
Modeling	Moves objects around the scene
Modelview	Describes the duality of viewing and modeling transformations
Projection	Clips and sizes the viewing volume
Viewport	A pseudo-transformation that scales the final output to the window

Eye Coordinates

An important concept throughout this chapter is that of *eye coordinates*. Eye coordinates are from the viewpoint of the observer, regardless of any transformations that may occur; you can think of them as "absolute" screen coordinates. Thus, eye coordinates are not real coordinates; instead, they represent a virtual fixed coordinate system that is used as a common frame of reference. All the transformations discussed in this chapter are described in terms of their effects relative to the eye coordinate system.

[Figure 4.1](#) shows the eye coordinate system from two viewpoints. On the left (a), the eye coordinates are represented as seen by the observer of the scene (that is, perpendicular to the monitor). On the right (b), the eye coordinate system is rotated slightly so you can better see the relation of the z-axis. Positive x and y are pointed right and up, respectively, from the viewer's perspective. Positive z travels away from the origin toward the user, and negative z values travel farther away from the viewpoint into the screen.

Figure 4.1. Two perspectives of eye coordinates.



When you draw in 3D with OpenGL, you use the Cartesian coordinate system. In the absence of any transformations, the system in use is identical to the eye coordinate system just described.

Viewing Transformations

The viewing transformation is the first to be applied to your scene. It is used to determine the vantage point of the scene. By default, the point of observation in a perspective projection is at the origin (0,0,0) looking down the negative z-axis ("into" the monitor screen). This point of observation is moved relative to the eye coordinate system to provide a specific vantage point. When the point of observation is located at the origin, objects drawn with positive z values are behind the observer.

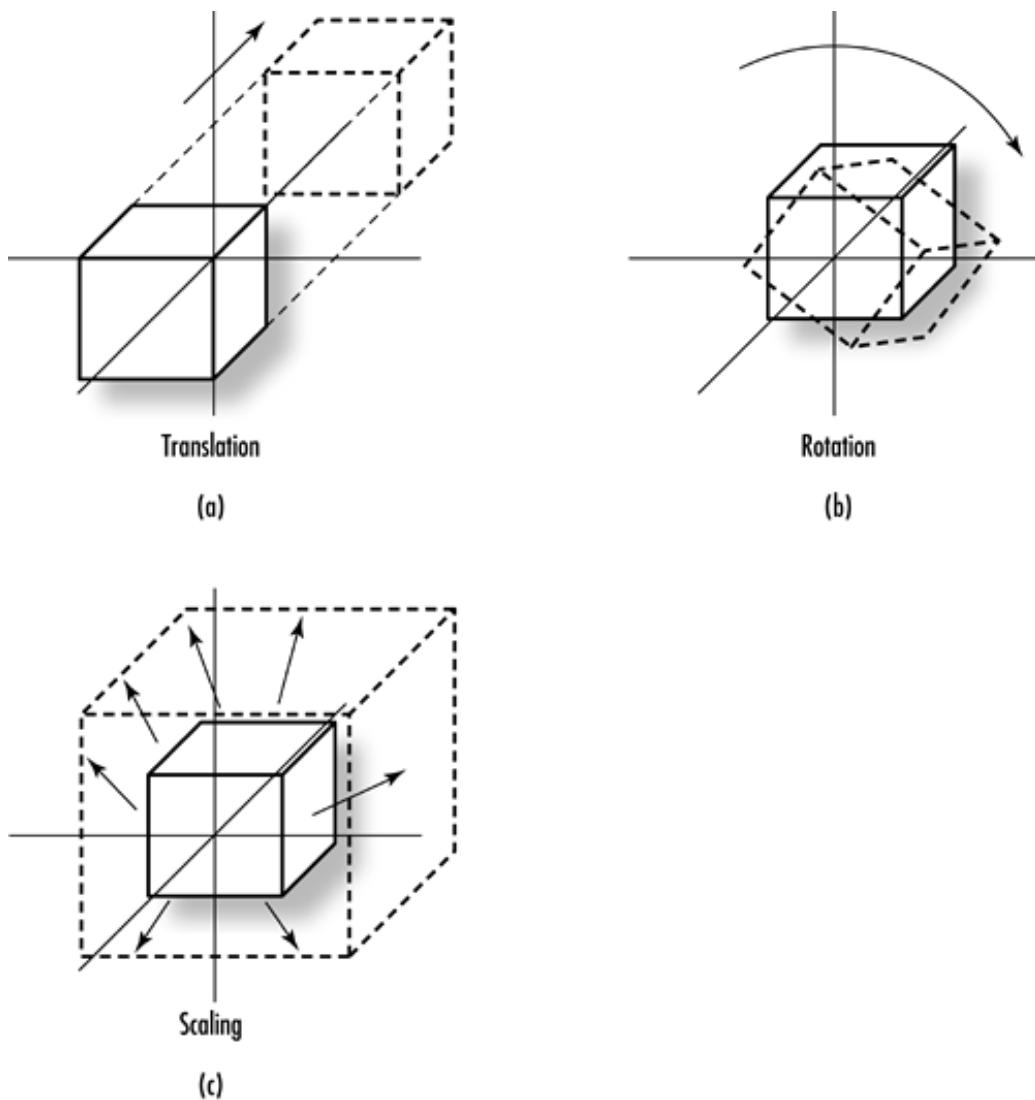
The viewing transformation allows you to place the point of observation anywhere you want and look in any direction. Determining the viewing transformation is like placing and pointing a camera at the scene.

In the grand scheme of things, you must specify the viewing transformation before any other transformations. The reason is that it appears to move the current working coordinate system in respect to the eye coordinate system. All subsequent transformations then occur based on the newly modified coordinate system. Later, you'll see more easily how this works, when we actually start looking at how to make these transformations.

Modeling Transformations

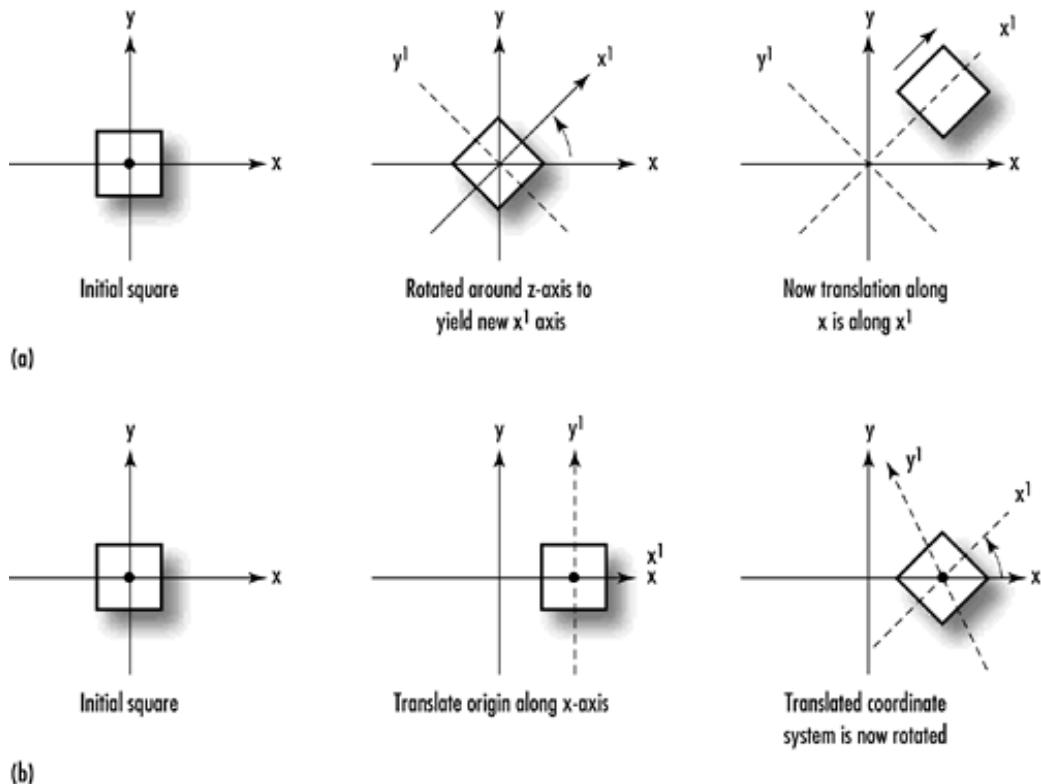
Modeling transformations are used to manipulate your model and the particular objects within it. These transformations move objects into place, rotate them, and scale them. [Figure 4.2](#) illustrates three of the most common modeling transformations that you will apply to your objects. [Figure 4.2a](#) shows translation, where an object is moved along a given axis. [Figure 4.2b](#) shows a rotation, where an object is rotated about one of the axes. Finally, [Figure 4.2c](#) shows the effects of scaling, where the dimensions of the object are increased or decreased by a specified amount. Scaling can occur nonuniformly (the various dimensions can be scaled by different amounts), so you can use scaling to stretch and shrink objects.

Figure 4.2. The modeling transformations.



The final appearance of your scene or object can depend greatly on the order in which the modeling transformations are applied. This is particularly true of translation and rotation. [Figure 4.3a](#) shows the progression of a square rotated first about the z-axis and then translated down the newly transformed x-axis. In [Figure 4.3b](#), the same square is first translated down the x-axis and then rotated around the z-axis. The difference in the final dispositions of the square occurs because each transformation is performed with respect to the last transformation performed. In [Figure 4.3a](#), the square is rotated with respect to the origin first. In 4.3b, after the square is translated, the rotation is performed around the newly translated origin.

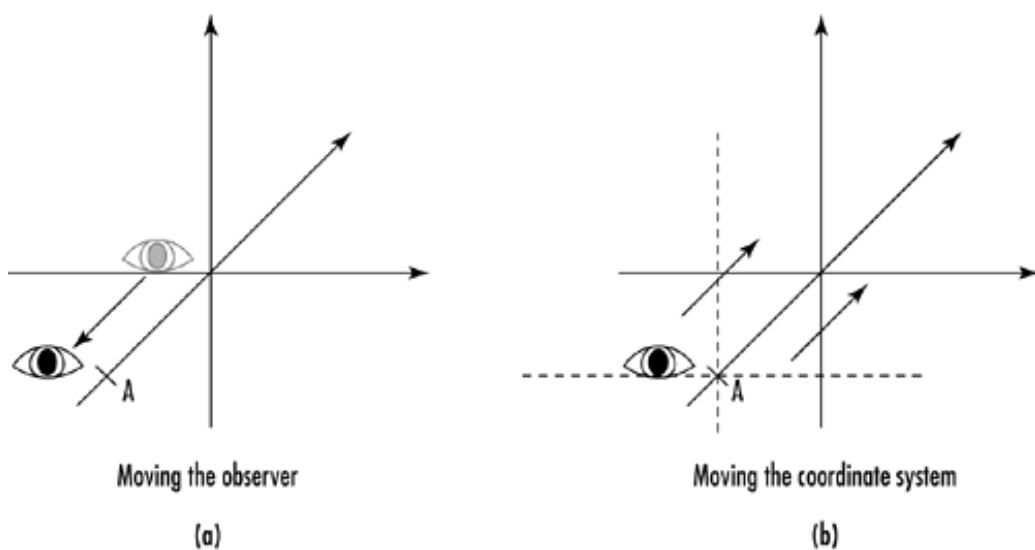
Figure 4.3. Modeling transformations: rotation/translation and translation/rotation.



The Modelview Duality

The viewing and modeling transformations are, in fact, the same in terms of their internal effects as well as their effects on the final appearance of the scene. The distinction between the two is made purely as a convenience for the programmer. There is no real difference between moving an object backward and moving the reference system forward; as shown in Figure 4.4, the net effect is the same. (You experience this effect firsthand when you're sitting in your car at an intersection and you see the car next to you roll forward; it might seem to you that your own car is rolling backward.) The term *modelview* indicates that you can think of this transformation either as the modeling transformation or the viewing transformation, but in fact there is no distinction; thus, it is the modelview transformation.

Figure 4.4. Two ways of looking at the viewing transformation.



The viewing transformation, therefore, is essentially nothing but a modeling transformation that you apply to a virtual object (the viewer) before drawing objects. As you will soon see, new

transformations are repeatedly specified as you place more objects in the scene. By convention, the initial transformation provides a reference from which all other transformations are based.

Projection Transformations

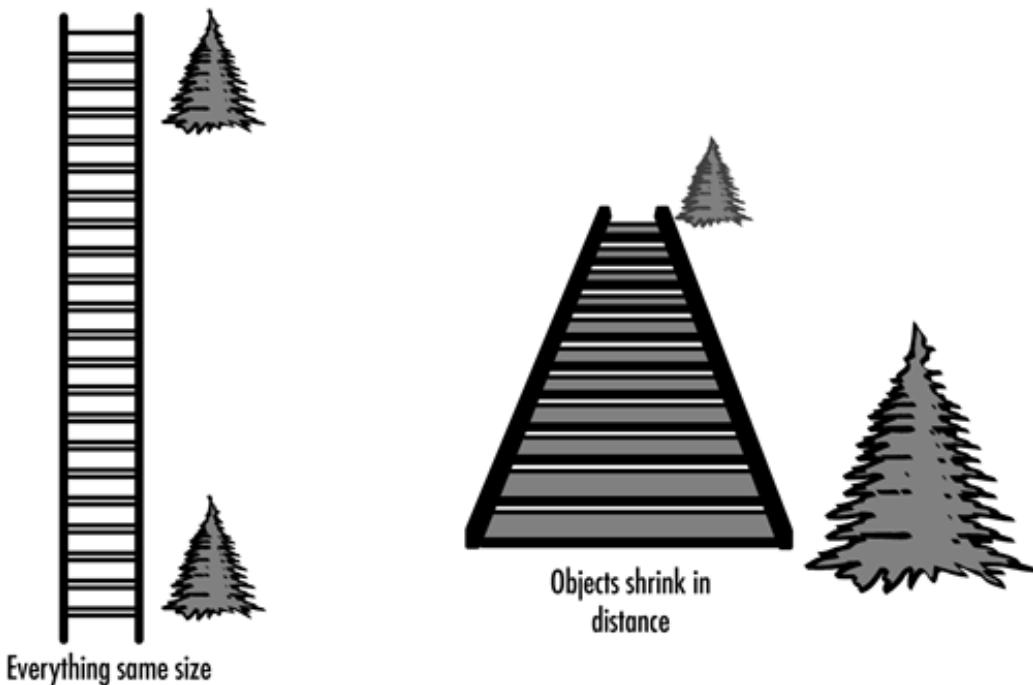
The projection transformation is applied to your vertices after the modelview transformation. This projection actually defines the viewing volume and establishes clipping planes. The clipping planes are plane equations in 3D space that OpenGL uses to determine whether geometry can be seen by the viewer.. More specifically, the projection transformation specifies how a finished scene (after all the modeling is done) is projected to the final image on the screen. You learn about two types of projections in this chapter: orthographic and perspective.

In an *orthographic*, or parallel, projection, all the polygons are drawn onscreen with exactly the relative dimensions specified. Lines and polygons are mapped directly to the 2D screen using parallel lines, which means no matter how far away something is, it is still drawn the same size, just flattened against the screen. This type of projection is typically used for CAD or for rendering two-dimensional images such as blueprints or two-dimensional graphics such as text or onscreen menus.

A *perspective* projection shows scenes more as they appear in real life instead of as a blueprint. The trademark of perspective projections is *foreshortening*, which makes distant objects appear smaller than nearby objects of the same size. Lines in 3D space that might be parallel do not always appear parallel to the viewer. With a railroad track, for instance, the rails are parallel, but using perspective projection, they appear to converge at some distant point.

The benefit of perspective projection is that you don't have to figure out where lines converge or how much smaller distant objects are. All you need to do is specify the scene using the modelview transformations and then apply the perspective projection. OpenGL works all the magic for you. [Figure 4.5](#) compares orthographic and perspective projections on two different scenes.

Figure 4.5. Side-by-side example of an orthographic versus perspective projection.



Orthographic projections are used most often for 2D drawing purposes where you want an exact correspondence between pixels and drawing units. You might use them for a schematic layout, text, or perhaps a 2D graphing application. You also can use an orthographic projection for 3D renderings when the depth of the rendering has a very small depth in comparison to the distance from the viewpoint. Perspective projections are used for rendering scenes that contain wide open

spaces or objects that need to have the foreshortening applied. For the most part, perspective projections are typical for 3D graphics. In fact, looking at a 3D object with an orthographic projection can be somewhat unsettling.

Viewport Transformations

When all is said and done, you end up with a two-dimensional projection of your scene that will be mapped to a window somewhere on your screen. This mapping to physical window coordinates is the last transformation that is done, and it is called the *viewport* transformation. Usually, a one-to-one correspondence exists between the color buffer and window pixels, but this is not always strictly the case. In some circumstances, the viewport transformation remaps what are called "normalized" device coordinates to window coordinates. Fortunately, this is something you don't need to worry about.

The Matrix: Mathematical Currency for 3D Graphics

Now that you're armed with some basic vocabulary and definitions of transformations, you're ready for some simple matrix mathematics. Let's examine how OpenGL performs these transformations and get to know the functions you call to achieve the desired effects.

The mathematics behind these transformations are greatly simplified by the mathematical notation of the matrix. You can achieve each of the transformations we have discussed by multiplying a matrix that contains the vertices (usually, this is a simple vector) by a matrix that describes the transformation. Thus, all the transformations achievable with OpenGL can be described as the product of two or more matrix multiplications.

What Is a Matrix?

The Matrix is not only a wildly popular Hollywood movie trilogy, but an exceptionally powerful mathematical tool that greatly simplifies the process of solving one or more equations with variables that have complex relationships to one another. One common example of this, near and dear to the hearts of graphics programmers, is coordinate transformations. For example, if you have a point in space represented by x, y, and z coordinates, and you need to know where that point is if you rotate it some number of degrees around some arbitrary point and orientation, you would use a matrix. Why? Because the new x coordinate depends not only on the old x coordinate and the other rotation parameters, but also on what the y and z coordinates were as well. This kind of dependency between the variables and solution is just the sort of problem that matrices excel at. For fans of the *Matrix* movies who have a mathematical inclination, the term *matrix* is indeed an appropriate title.

Mathematically, a matrix is nothing more than a set of numbers arranged in uniform rows and columns—in programming terms, a two-dimensional array. A matrix doesn't have to be square, but each row or column must have the same number of elements as every other row or column in the matrix. [Figure 4.6](#) presents some examples of matrices. They don't represent anything in particular, but serve only to demonstrate matrix structure. Note that it is also valid for a matrix to have a single column or row. A single row or column of numbers is also more simply called a *vector*, and vectors also have some interesting and useful applications all their own.

Figure 4.6. Three examples of matrices.

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}
 \begin{bmatrix} 0 & 42 \\ 1.5 & 0.877 \\ 2 & 14 \end{bmatrix}
 \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Matrix and *vector* are two important terms that you will see often in 3D graphics programming literature. When dealing with these quantities, you will also see the term *scalar*. A scalar is just an ordinary single number used to represent magnitude or a specific quantity (you know—a regular old plain simple number...like before you cared or had all this jargon added to your vocabulary).

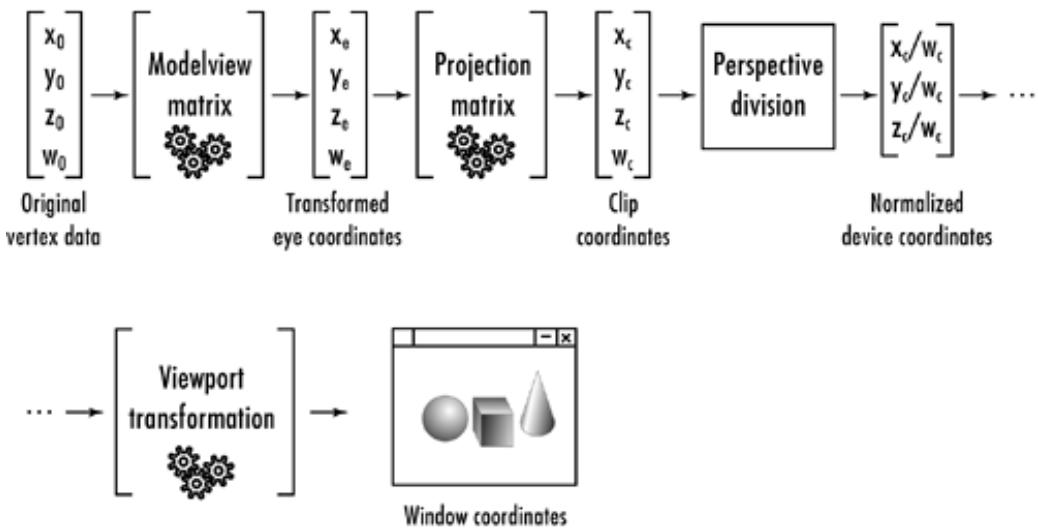
Matrices can be multiplied and added together, but they can also be multiplied by vectors and scalar values. Multiplying a point (a vector) by a matrix (a transformation) yields a new transformed point (a vector). Matrix transformations are actually not too difficult to understand but can be intimidating at first. Because an understanding of matrix transformations is fundamental to many 3D tasks, you should still make an attempt to become familiar with them. Fortunately, only a little understanding is enough to get you going and doing some pretty incredible things with OpenGL. Over time, and with a little more practice and study (see [Appendix A](#), "Further Reading"), you will master this mathematical tool yourself. In the meantime, you can find a number of useful matrix and vector functions and features available, with source code, in the `glTools` library (see the `\common` directory under the `samples` directory) on the CD that accompanies this book.

The Transformation Pipeline

To effect the types of transformations described in this chapter, you modify two matrices in particular: the modelview matrix and projection matrix. Don't worry; OpenGL provides some high-level functions that you can call for these transformations. After you've mastered the basics of the OpenGL API, you will undoubtedly start trying some of the more advanced 3D rendering techniques. Only then will you need to call the lower-level functions that actually set the values contained in the matrices.

The road from raw vertex data to screen coordinates is a long one. [Figure 4.7](#) provides a flowchart of this process. First, your vertex is converted to a 1×4 matrix in which the first three values are the x , y , and z coordinates. The fourth number is a scaling factor that you can apply manually by using the vertex functions that take four values. This is the w coordinate, usually 1.0 by default. You will seldom modify this value directly.

Figure 4.7. The vertex transformation pipeline.



The vertex is then multiplied by the modelview matrix, which yields the transformed eye coordinates. The eye coordinates are then multiplied by the projection matrix to yield clip coordinates. This effectively eliminates all data outside the viewing volume. The clip coordinates are then divided by the w coordinate to yield normalized device coordinates. The w value may have been modified by the projection matrix or the modelview matrix, depending on the transformations that occurred. Again, OpenGL and the high-level matrix functions hide this process from you.

Finally, your coordinate triplet is mapped to a 2D plane by the viewport transformation.

The Modelview Matrix

The modelview matrix is a 4x4 matrix that represents the transformed coordinate system you are using to place and orient your objects. The vertices you provide for your primitives are used as a single-column matrix and multiplied by the modelview matrix to yield new transformed coordinates in relation to the eye coordinate system.

In [Figure 4.8](#), a matrix containing data for a single vertex is multiplied by the modelview matrix to yield new eye coordinates. The vertex data is actually four elements with an extra value, w, that represents a scaling factor. This value is set by default to 1.0, and rarely will you change it yourself.

Figure 4.8. Matrix equation that applies the modelview transformation to a single vertex.

$$\begin{bmatrix} X & Y & Z & W \end{bmatrix} \begin{bmatrix} & 4 \times 4 \\ M \end{bmatrix} = \begin{bmatrix} X_e & Y_e & Z_e & W_e \end{bmatrix}$$

Translation

Let's consider an example that modifies the modelview matrix. Say you want to draw a cube using the GLUT library's `glutWireCube` function. You simply call

```
glutWireCube(10.0f);
```

A cube that measures 10 units on a side is then centered at the origin. To move the cube up the y-axis by 10 units before drawing it, you multiply the modelview matrix by a matrix that describes a translation of 10 units up the y-axis and then do your drawing. In skeleton form, the code looks like this:

```
// Construct a translation matrix for positive 10 Y
...
// Multiply it by the modelview matrix
...
// Draw the cube
glutWireCube(10.0f);
```

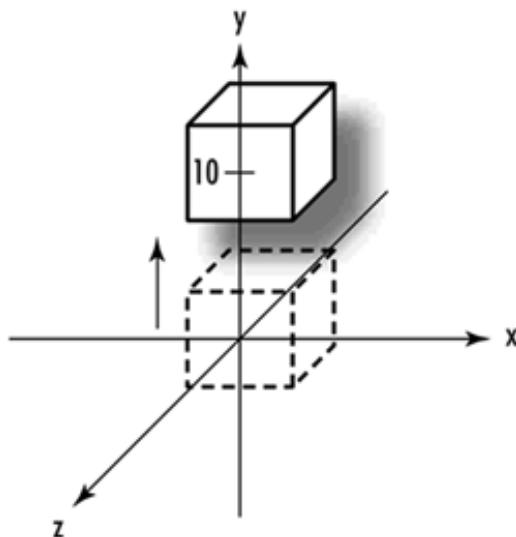
Actually, such a matrix is fairly easy to construct, but it requires quite a few lines of code. Fortunately, OpenGL provides a high-level function that performs this task for you:

```
void glTranslatef(GLfloat x, GLfloat y, GLfloat z);
```

This function takes as parameters the amount to translate along the x, y, and z directions. It then constructs an appropriate matrix and does the multiplication. The pseudocode looks like the following, and the effect is illustrated in [Figure 4.9](#):

```
// Translate up the y-axis 10 units
glTranslatef(0.0f, 10.0f, 0.0f);
// Draw the cube
glutWireCube(10.0f);
```

Figure 4.9. A cube translated 10 units in the positive y direction.



IS TRANSLATION ALWAYS A MATRIX OPERATION?

The studious reader may note that translations do not always require a full matrix multiplication, but can be simplified with a simple scalar addition to the vertex position. However, for more complex transformations that include combined simultaneous operations, it is correct to describe translation as a matrix operation. Fortunately, if you let OpenGL do the heavy lifting for you, such as we have done here, the implementation can usually figure out the optimum method to use.

Rotation

To rotate an object about one of the three coordinate axes, or indeed any arbitrary vector, you have to devise a rotation matrix. Again, a high-level function comes to the rescue:

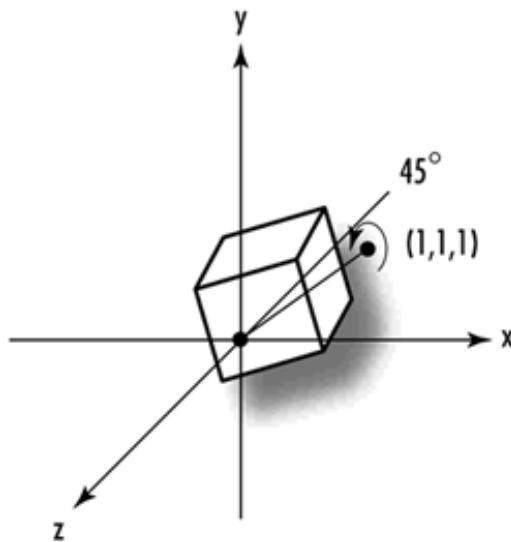
```
glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);
```

Here, we perform a rotation around the vector specified by the *x*, *y*, and *z* arguments. The angle of rotation is in the counterclockwise direction measured in degrees and specified by the argument *angle*. In the simplest of cases, the rotation is around only one of the axes.

You can also perform a rotation around an arbitrary axis by specifying *x*, *y*, and *z* values for that vector. To see the axis of rotation, you can just draw a line from the origin to the point represented by (*x*,*y*,*z*). The following code rotates the cube by 45° around an arbitrary axis specified by (1,1,1), as illustrated in Figure 4.10:

```
// Perform the transformation
glRotatef(45.0f, 1.0f, 1.0f, 1.0f);
// Draw the cube
glutWireCube(10.0f);
```

Figure 4.10. A cube rotated about an arbitrary axis.



Scaling

A scaling transformation increases the size of your object by expanding all the vertices along the three axes by the factors specified. The function

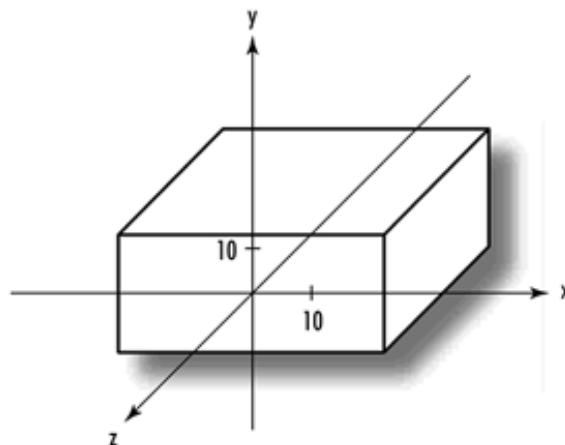
```
glScalef(GLfloat x, GLfloat y, GLfloat z);
```

multiplies the *x*, *y*, and *z* values by the scaling factors specified.

Scaling does not have to be uniform, and you can use it to both stretch or squeeze objects along different directions. For example, the following code produces a cube that is twice as large along the *x*- and *z*-axes as the cubes discussed in the previous examples, but still the same along the *y*-axis. The result is shown in [Figure 4.11](#).

```
// Perform the scaling transformation
glScalef(2.0f, 1.0f, 2.0f);
// Draw the cube
glutWireCube(10.0f);
```

Figure 4.11. A nonuniform scaling of a cube.



The Identity Matrix

About now, you might be wondering why we had to bother with all this matrix stuff in the first place. Can't we just call these transformation functions to move our objects around and be done

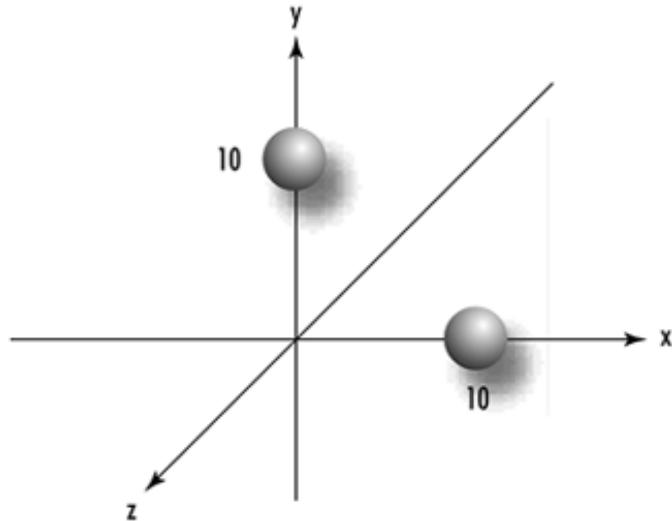
with it? Do we really need to know that it is the modelview matrix that is modified?

The answer is yes and no (but it's no only if you are drawing a single object in your scene). The reason is that the effects of these functions are cumulative. Each time you call one, the appropriate matrix is constructed and multiplied by the current modelview matrix. The new matrix then becomes the current modelview matrix, which is then multiplied by the next transformation, and so on.

Suppose you want to draw two spheres—one 10 units up the positive y-axis and one 10 units out the positive x-axis, as shown in [Figure 4.12](#). You might be tempted to write code that looks something like this:

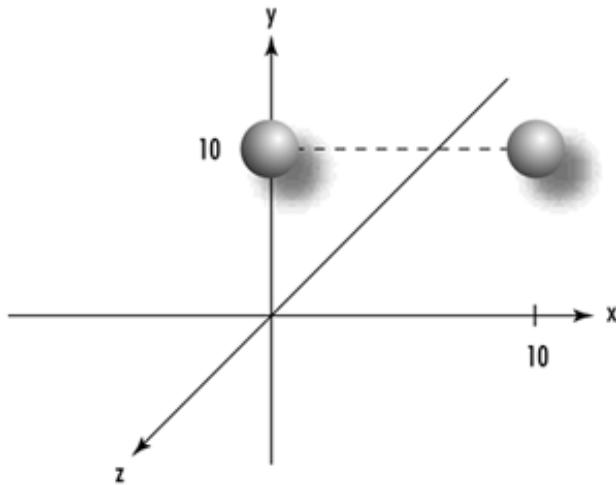
```
// Go 10 units up the y-axis
glTranslatef(0.0f, 10.0f, 0.0f);
// Draw the first sphere
glutSolidSphere(1.0f,15,15);
// Go 10 units out the x-axis
glTranslatef(10.0f, 0.0f, 0.0f);
// Draw the second sphere
glutSolidSphere(1.0f);
```

Figure 4.12. Two spheres drawn on the y- and x-axes.



Consider, however, that each call to `glTranslate` is cumulative on the modelview matrix, so the second call translates 10 units in the positive x direction from the previous translation in the y direction. This yields the results shown in [Figure 4.13](#).

Figure 4.13. The result of two consecutive translations.



You can make an extra call to `glTranslate` to back down the y-axis 10 units in the negative direction, but this makes some complex scenes difficult to code and debug—not to mention that you throw extra transformation math at the CPU. A simpler method is to reset the modelview matrix to a known state—in this case, centered at the origin of the eye coordinate system.

You reset the origin by loading the modelview matrix with the *identity matrix*. The identity matrix specifies that no transformation is to occur, in effect saying that all the coordinates you specify when drawing are in eye coordinates. An identity matrix contains all 0s, with the exception of a diagonal row of 1s. When this matrix is multiplied by any vertex matrix, the result is that the vertex matrix is unchanged. [Figure 4.14](#) shows this equation. Later in the chapter, we discuss in more detail why these numbers are where they are.

Figure 4.14. Multiplying a vertex by the identity matrix yields the same vertex matrix.

$$\begin{bmatrix} 8.0 & 4.5 & -2.0 & 1.0 \end{bmatrix} \begin{bmatrix} 1.0 & 0 & 0 & 0 \\ 0 & 1.0 & 0 & 0 \\ 0 & 0 & 1.0 & 0 \\ 0 & 0 & 0 & 1.0 \end{bmatrix} = \begin{bmatrix} 8.0 & 4.5 & -2.0 & 1.0 \end{bmatrix}$$

As we've already stated, the details of performing matrix multiplication are outside the scope of this book. For now, just remember this: Loading the identity matrix means that no transformations are performed on the vertices. In essence, you are resetting the modelview matrix back to the origin.

The following two lines load the identity matrix into the modelview matrix:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

The first line specifies that the current operating matrix is the modelview matrix. After you set the current operating matrix (the matrix that your matrix functions are affecting), it remains the active matrix until you change it. The second line loads the current matrix (in this case, the modelview matrix) with the identity matrix.

Now, the following code produces results as shown earlier in [Figure 4.12](#):

```
// Set current matrix to modelview and reset
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

```

// Go 10 units up the y-axis
glTranslatef(0.0f, 10.0f, 0.0f);
// Draw the first sphere
glutSolidSphere(1.0f, 15, 15);
// Reset modelview matrix again
glLoadIdentity();
// Go 10 units out the x-axis
glTranslatef(10.0f, 0.0f, 0.0f);
// Draw the second sphere
glutSolidSphere(1.0f, 15, 15);

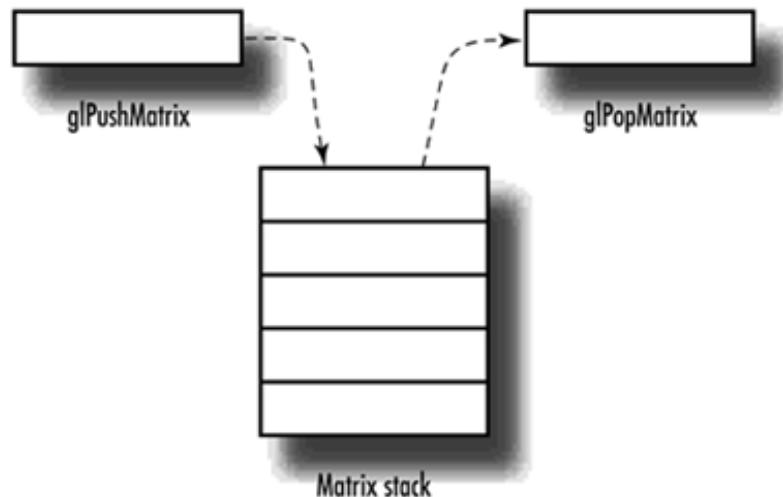
```

The Matrix Stacks

Resetting the modelview matrix to identity before placing every object is not always desirable. Often, you want to save the current transformation state and then restore it after some objects have been placed. This approach is most convenient when you have initially transformed the modelview matrix as your viewing transformation (and thus are no longer located at the origin).

To facilitate this procedure, OpenGL maintains a *matrix stack* for both the modelview and projection matrices. A matrix stack works just like an ordinary program stack. You can push the current matrix onto the stack to save it and then make your changes to the current matrix. Popping the matrix off the stack restores it. [Figure 4.15](#) shows the stack principle in action.

Figure 4.15. The matrix stack in action.



TEXTURE MATRIX STACK

The texture stack is another matrix stack available to you. You use it to transform texture coordinates. [Chapter 8](#), "Texture Mapping: The Basics," examines texture mapping and texture coordinates and contains a discussion of the texture matrix stack.

The stack depth can reach a maximum value that you can retrieve with a call to either

```
glGet(GL_MAX_MODELVIEW_STACK_DEPTH);
```

or

```
glGet(GL_MAX_PROJECTION_STACK_DEPTH);
```

If you exceed the stack depth, you get a `GL_STACK_OVERFLOW` error; if you try to pop a matrix

value off the stack when there is none, you generate a `GL_STACK_UNDERFLOW` error. The stack depth is implementation dependent. For the Microsoft software implementation, the values are 32 for the modelview and 2 for the projection stack.

A Nuclear Example

Let's put to use what we have learned. In the next example, we build a crude, animated model of an atom. This atom has a single sphere at the center to represent the nucleus and three electrons in orbit about the atom. We use an orthographic projection, as we have previously in this book.

Our ATOM program uses the GLUT timer callback mechanism (discussed in [Chapter 2](#), "Using OpenGL") to redraw the scene about 10 times per second. Each time the `RenderScene` function is called, the angle of revolution about the nucleus is incremented. Also, each electron lies in a different plane. [Listing 4.1](#) shows the `RenderScene` function for this example, and the output from the ATOM program is shown in [Figure 4.16](#).

Listing 4.1. `RenderScene` Function from ATOM Sample Program

```
// Called to draw scene
void RenderScene(void)
{
    // Angle of revolution around the nucleus
    static GLfloat fElect1 = 0.0f;
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // Reset the modelview matrix
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    // Translate the whole scene out and into view
    // This is the initial viewing transformation
    glTranslatef(0.0f, 0.0f, -100.0f);
    // Red Nucleus
    glColor3ub(255, 0, 0);
    glutSolidSphere(10.0f, 15, 15);
    // Yellow Electrons
    glColor3ub(255,255,0);
    // First Electron Orbit
    // Save viewing transformation
    glPushMatrix();
    // Rotate by angle of revolution
    glRotatef(fElect1, 0.0f, 1.0f, 0.0f);
    // Translate out from origin to orbit distance
    glTranslatef(90.0f, 0.0f, 0.0f);

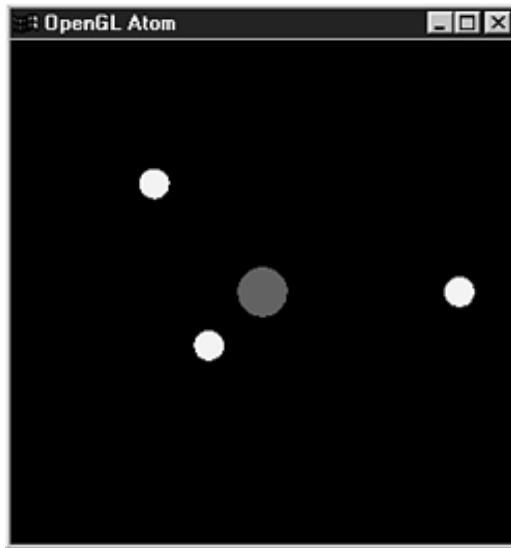
    // Draw the electron
    glutSolidSphere(6.0f, 15, 15);
    // Restore the viewing transformation
    glPopMatrix();
    // Second Electron Orbit
    glPushMatrix();
    glRotatef(45.0f, 0.0f, 0.0f, 1.0f);
    glRotatef(fElect1, 0.0f, 1.0f, 0.0f);
    glTranslatef(-70.0f, 0.0f, 0.0f);
    glutSolidSphere(6.0f, 15, 15);
    glPopMatrix();
    // Third Electron Orbit
    glPushMatrix();
    glRotatef(360.0f, -45.0f, 0.0f, 0.0f, 1.0f);
    glRotatef(fElect1, 0.0f, 1.0f, 0.0f);
    glTranslatef(0.0f, 0.0f, 60.0f);
```

```

glutSolidSphere(6.0f, 15, 15);
glPopMatrix();
// Increment the angle of revolution
fElect1 += 10.0f;
if(fElect1 > 360.0f)
    fElect1 = 0.0f;
// Show the image
glutSwapBuffers();
}

```

Figure 4.16. Output from the ATOM sample program.



Let's examine the code for placing one of the electrons, a couple of lines at a time. The first line saves the current modelview matrix by pushing the current transformation on the stack:

```

// First Electron Orbit
// Save viewing transformation
glPushMatrix();

```

Now the coordinate system appears to be rotated around the y-axis by an angle, `fElect1`:

```

// Rotate by angle of revolution
glRotatef(fElect1, 0.0f, 1.0f, 0.0f);

```

The electron is drawn by translating down the newly rotated coordinate system:

```

// Translate out from origin to orbit distance
glTranslatef(90.0f, 0.0f, 0.0f);

```

Then the electron is drawn (as a solid sphere), and we restore the modelview matrix by popping it off the matrix stack:

```

// Draw the electron
glutSolidSphere(6.0f, 15, 15);
// Restore the viewing transformation
glPopMatrix();

```

The other electrons are placed similarly.

Using Projections

In our examples so far, we have used the modelview matrix to position our vantage point of the viewing volume and to place our objects therein. The projection matrix actually specifies the size and shape of our viewing volume.

Thus far in this book, we have created a simple parallel viewing volume using the function `glOrtho`, setting the near and far, left and right, and top and bottom clipping coordinates. When the projection matrix is loaded with the identity matrix, the diagonal line of 1s specifies that the clipping planes extend from the origin to +1 or -1 in all directions. The projection matrix by itself does no scaling or perspective adjustments unless you load a perspective projection matrix.

The next two sample programs, ORTHO and PERSPECT, are not covered in detail from the standpoint of their source code. These examples use lighting and shading that we haven't covered yet to help highlight the differences between an orthographic and a perspective projection. These interactive samples make it much easier for you to see firsthand how the projection can distort the appearance of an object. If possible, you should run these examples while reading the next two sections.

Orthographic Projections

An orthographic projection, used for most of this book so far, is square on all sides. The logical width is equal at the front, back, top, bottom, left, and right sides. This produces a parallel projection, which is useful for drawings of specific objects that do not have any foreshortening when viewed from a distance. This is good for CAD, 2D graphics such as text, or architectural drawings for which you want to represent the exact dimensions and measurements onscreen.

[Figure 4.17](#) shows the output from the sample program ORTHO in this chapter's subdirectory on the CD. To produce this hollow, tube-like box, we used an orthographic projection just as we did for all our previous examples. [Figure 4.18](#) shows the same box rotated more to the side so you can see how long it actually is.

Figure 4.17. A hollow square tube shown with an orthographic projection.

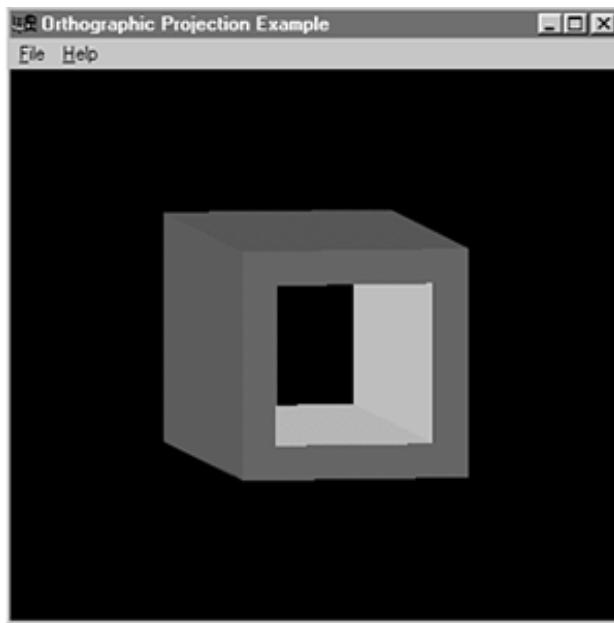
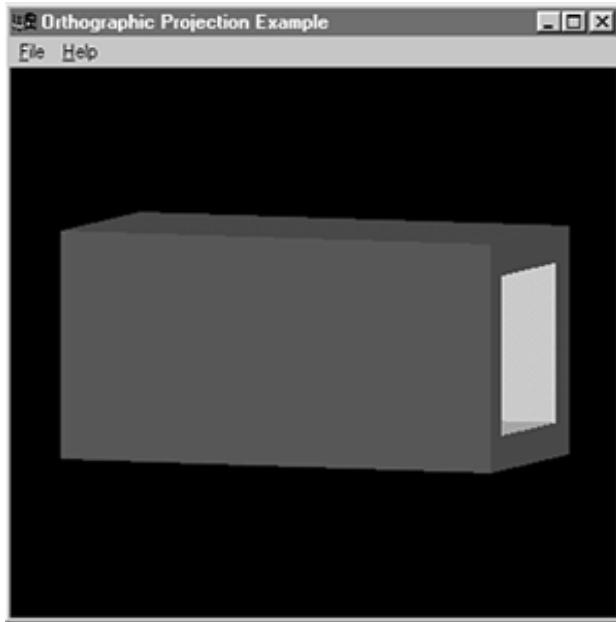
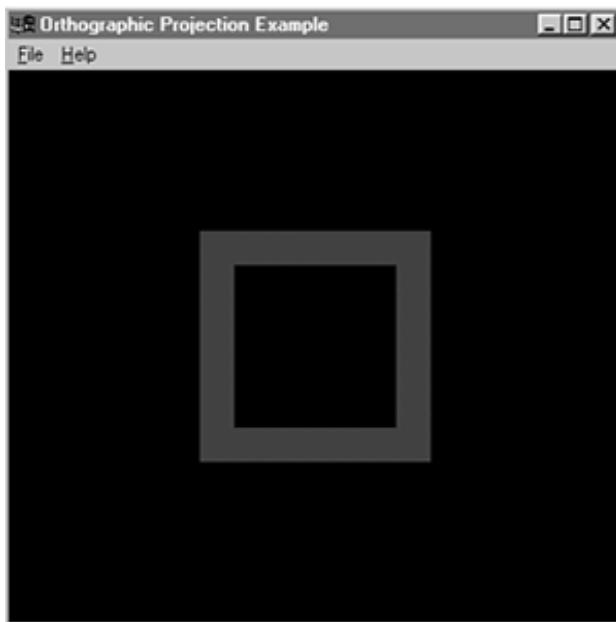


Figure 4.18. A side view showing the length of the square tube.



In [Figure 4.19](#), you're looking directly down the barrel of the tube. Because the tube does not converge in the distance, this is not an entirely accurate view of how such a tube appears in real life. To add some perspective, we must use a perspective projection.

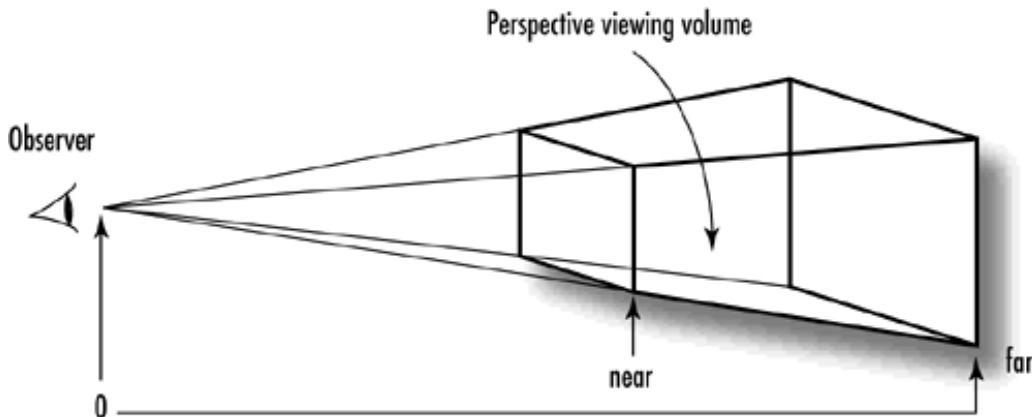
Figure 4.19. Looking down the barrel of the tube.



Perspective Projections

A perspective projection performs perspective division to shorten and shrink objects that are farther away from the viewer. The width of the back of the viewing volume does not have the same measurements as the front of the viewing volume after being projected to the screen. Thus, an object of the same logical dimensions appears larger at the front of the viewing volume than if it were drawn at the back of the viewing volume.

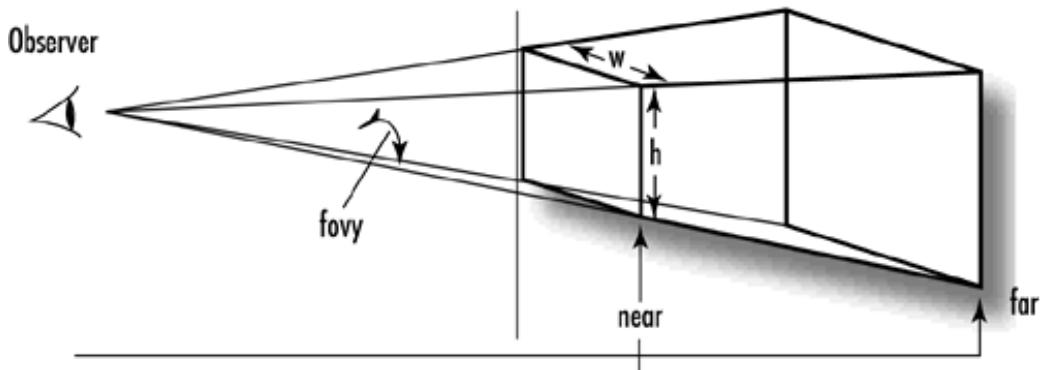
The picture in our next example is of a geometric shape called a *frustum*. A frustum is a truncated section of a pyramid viewed from the narrow end to the broad end. [Figure 4.20](#) shows the frustum, with the observer in place.

Figure 4.20. A perspective projection defined by a frustum.

You can define a frustum with the function `glFrustum`. Its parameters are the coordinates and distances between the front and back clipping planes. However, `glFrustum` is not intuitive about setting up your projection to get the desired effects. The utility function `gluPerspective` is easier to use and somewhat more intuitive for most purposes:

```
void gluPerspective(GLdouble fovy, GLdouble aspect,
                    GLdouble zNear, GLdouble zFar);
```

Parameters for the `gluPerspective` function are a field-of-view angle in the vertical direction, the aspect ratio of the height to width, and the distances to the near and far clipping planes (see [Figure 4.21](#)). You find the aspect ratio by dividing the width (w) by the height (h) of the window or viewport.

Figure 4.21. The frustum as defined by `gluPerspective`.

[Listing 4.2](#) shows how we change our orthographic projection from the previous examples to use a perspective projection. Foreshortening adds realism to our earlier orthographic projections of the square tube (see [Figures 4.22](#), [4.23](#), and [4.24](#)). The only substantial change we made for our typical projection code in [Listing 4.2](#) was substituting the call to `gluOrtho2D` with `gluPerspective`.

Listing 4.2. Setting Up the Perspective Projection for the PERSPECT Sample Program

```
// Change viewing volume and viewport.  Called when window is resized
void ChangeSize(GLsizei w, GLsizei h)
{
    GLfloat fAspect;
    // Prevent a divide by zero
    if(h == 0)
```

```
h = 1;  
// Set viewport to window dimensions  
glViewport(0, 0, w, h);  
fAspect = (GLfloat)w/(GLfloat)h;  
// Reset coordinate system  
glMatrixMode(GL_PROJECTION);  
glLoadIdentity();  
// Produce the perspective projection  
gluPerspective(60.0f, fAspect, 1.0, 400.0);  
glMatrixMode(GL_MODELVIEW);  
glLoadIdentity();  
}
```

Figure 4.22. The square tube with a perspective projection.

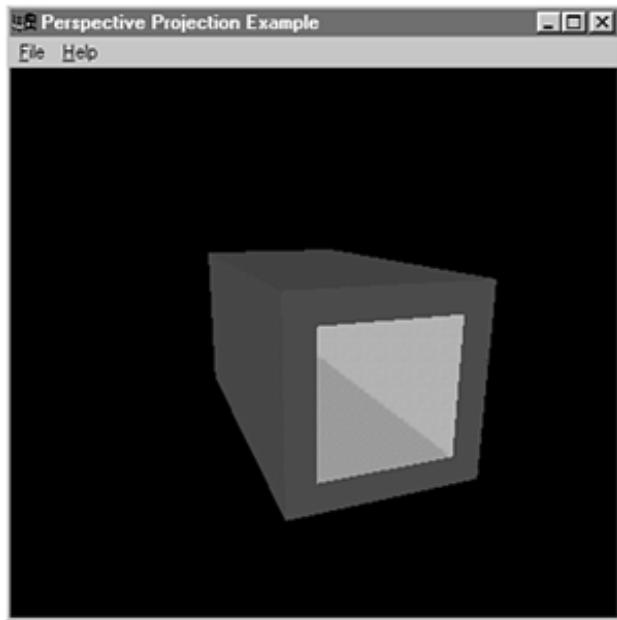


Figure 4.23. Side view with foreshortening.

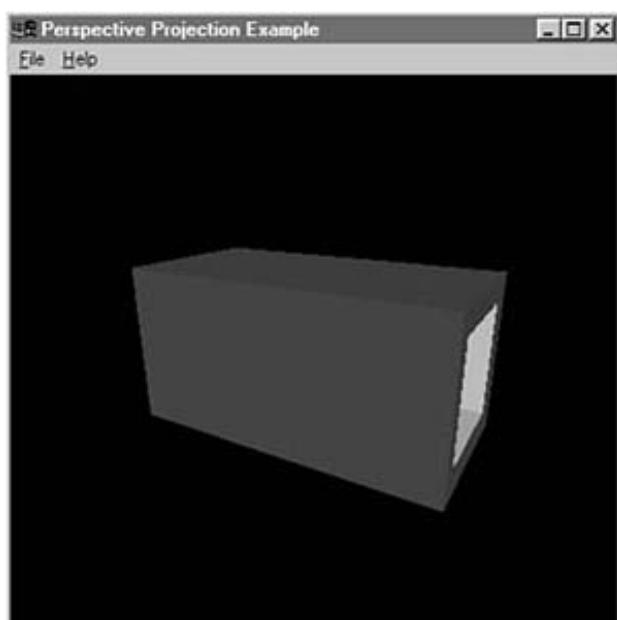


Figure 4.24. Looking down the barrel of the tube with perspective added.



We made the same changes to the ATOM example in ATOM2 to add perspective. Run the two side by side, and you see how the electrons appear to be smaller as they swing far away behind the nucleus.

A Far-Out Example

For a more complete example showing modelview manipulation and perspective projections, we have modeled the Sun and the Earth/Moon system in revolution in the SOLAR example program. This is a classic example of nested transformations with objects being transformed relative to one another using the matrix stack. We have enabled some lighting and shading for drama so you can more easily see the effects of our operations. You'll learn about shading and lighting in the next two chapters.

In our model, the Earth moves around the Sun, and the Moon revolves around the Earth. A light source is placed at the center of the Sun, which is drawn without lighting to make it appear to be the glowing light source. This powerful example shows how easily you can produce sophisticated effects with OpenGL.

[Listing 4.3](#) shows the code that sets up the projection and the rendering code that keeps the system in motion. A timer elsewhere in the program triggers a window redraw 10 times a second to keep the `RenderScene` function in action. Notice in [Figures 4.25](#) and [4.26](#) that when the Earth appears larger, it's on the near side of the Sun; on the far side, it appears smaller.

Listing 4.3. Code That Produces the Sun/Earth/Moon System

```
// Change viewing volume and viewport.  Called when window is resized
void ChangeSize(GLsizei w, GLsizei h)
{
    GLfloat fAspect;
    // Prevent a divide by zero
    if(h == 0)
        h = 1;
    // Set viewport to window dimensions
    glViewport(0, 0, w, h);
    // Calculate aspect ratio of the window
    fAspect = (GLfloat)w/(GLfloat)h;
    // Set the perspective coordinate system
    glMatrixMode(GL_PROJECTION);
```

```

glLoadIdentity();
// Field of view of 45 degrees, near and far planes 1.0 and 425
gluPerspective(45.0f, fAspect, 1.0, 425.0);
// Modelview matrix reset
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}

// Called to draw scene
void RenderScene(void)
{
    // Earth and Moon angle of revolution
    static float fMoonRot = 0.0f;
    static float fEarthRot = 0.0f;
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // Save the matrix state and do the rotations
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    // Translate the whole scene out and into view
    glTranslatef(0.0f, 0.0f, -300.0f);
    // Set material color, to yellow
    // Sun
    glColor3ub(255, 255, 0);
    glDisable(GL_LIGHTING);
    glutSolidSphere(15.0f, 15, 15);
    glEnable(GL_LIGHTING);
    // Position the light after we draw the sun!
    glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
    // Rotate coordinate system
    glRotatef(fEarthRot, 0.0f, 1.0f, 0.0f);
    // Draw the Earth
    glColor3ub(0,0,255);
    glTranslatef(105.0f,0.0f,0.0f);
    glutSolidSphere(15.0f, 15, 15);
    // Rotate from Earth-based coordinates and draw Moon
    glColor3ub(200,200,200);
    glRotatef(fMoonRot,0.0f, 1.0f, 0.0f);
    glTranslatef(30.0f, 0.0f, 0.0f);
    fMoonRot+= 15.0f;
    if(fMoonRot > 360.0f)
        fMoonRot = 0.0f;
    glutSolidSphere(6.0f, 15, 15);
    // Restore the matrix state
    glPopMatrix();    // Modelview matrix
    // Step earth orbit 5 degrees
    fEarthRot += 5.0f;
    if(fEarthRot > 360.0f)
        fEarthRot = 0.0f;
    // Show the image    glutSwapBuffers();
}

```

Figure 4.25. The Sun/Earth/Moon system with the Earth on the near side.

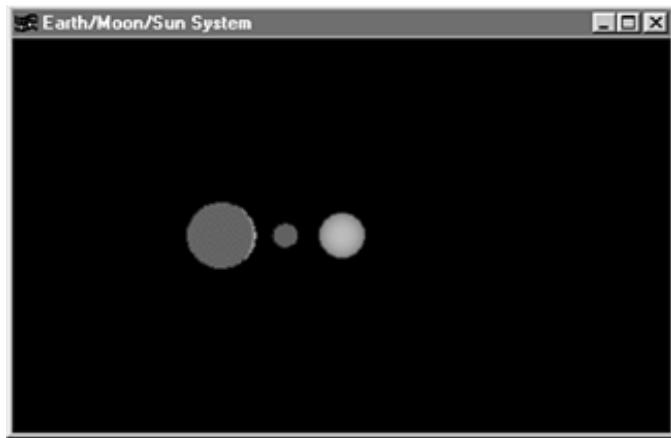
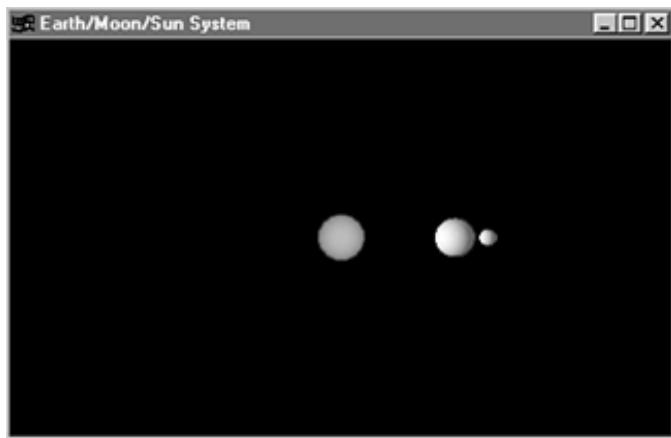


Figure 4.26. The Sun/Earth/Moon system with the Earth on the far side.



Advanced Matrix Manipulation

These higher-level "canned" transformations (for rotation, scaling, and translation) are great for many simple transformation problems. Real power and flexibility, however, are afforded to those who take the time to understand using matrices directly. Doing so is not as hard as it sounds, but first you need to understand the magic behind those 16 numbers that make up a 4x4 transformation matrix.

OpenGL represents a 4x4 matrix not as a two-dimensional array of floating-point values, but as a single array of 16 floating-point values. This approach is different from many math libraries, which do take the two-dimensional array approach. For example, OpenGL prefers the first of these two examples:

```
GLfloat matrix[16];           // Nice OpenGL friendly matrix
GLfloat matrix[4][4];          // Popular, but not as efficient for OpenGL
```

OpenGL can use the second variation, but the first is a more efficient representation. The reason for this will become clear in a moment. These 16 elements represent the 4x4 matrix, as shown in [Figure 4.27](#). When the array elements traverse down the matrix columns one by one, we call this *column-major matrix ordering*. In memory, the 4x4 approach of the two-dimensional array (the second option in the preceding code) is laid out in a *row-major order*. In math terms, the two orientations are the *transpose* of one another.

Figure 4.27. Column-major matrix ordering.

$$\begin{bmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{bmatrix}$$

The real magic lies in the fact that these 16 values represent a particular position in space and an orientation of the three axes with respect to the eye coordinate system (remember that fixed, unchanging coordinate system we talked about earlier). Interpreting these numbers is not hard at all. The four columns each represent a four-element vector. To keep things simple for this book, we focus our attention to just the first three elements of these vectors. The fourth column vector contains the x, y, and z values of the transformed coordinate system. When you call `glTranslate` on the identity matrix, all it does is put your values for x, y, and z in the 12th, 13th, and 14th position of the matrix.

The first three columns are just directional vectors that represent the orientation (vectors here are used to represent a direction) of the x-, y-, and z-axes in space. For most purposes, these three vectors are always at 90° angles from each other. The mathematical term for this (in case you want to impress your friends) is *orthonormal*. Figure 4.28 shows the 4x4 transformation matrix with the column vectors highlighted. Notice the last row of the matrix is all 0s with the exception of the very last element, which is 1.

Figure 4.28. How a 4x4 matrix represents a position and orientation in 3D space.

X axis direction Y axis direction Z axis direction Translation/location
 ↓ ↓ ↓ ↓

$$\begin{bmatrix} X_x & Y_x & Z_x & T_x \\ X_y & Y_y & Z_y & T_y \\ X_z & Y_z & Z_z & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The most amazing thing is that if you have a 4x4 matrix that contains the position and orientation of a different coordinate system, and you multiply a vertex (as a column matrix or vector) by this matrix, the result is a new vertex that has been transformed to the new coordinate system. This means that any position in space and any desired orientation can be uniquely defined by a 4x4 matrix, and if you multiply all of an object's vertices by this matrix, you transform the entire object to the given location and orientation in space!

HARDWARE TRANSFORMATIONS

Many OpenGL implementations have what is called *hardware T&L* (for Transform and Lighting). This means that the transformation matrix multiplies many thousands of vertices on special graphics hardware that performs this operation very, very fast. (Intel and AMD can eat their hearts out!) However, functions such as `glRotate` and `glScale`, which create transformation matrices for you, are usually not hardware accelerated because typically they represent an exceedingly small fraction of the enormous amount of matrix math that must be done to draw a scene.

So why does OpenGL insist on using column-major ordering? Simple. To get an axis-directional vector or the translation from a matrix, the implementation simply does one copy from memory to get all the data found in one place. In row-major ordering, the software must access three different memory locations (or four) to get just a single vector from the matrix.

Loading a Matrix

After you have a handle on the way the 4x4 matrix represents a given location and orientation, you may want to compose and load your own transformation matrices. You can load an arbitrary column-major matrix into the projection, modelview, or texture matrix stacks by using the following function:

```
glLoadMatrixf(GLfloat m);
```

or

```
glLoadMatrixd(GLfloat m);
```

Most OpenGL implementations store and manipulate pipeline data as floats and not doubles; consequently, using the second variation may incur some performance penalty because 16 double-precision numbers must be converted into single-precision floats.

The following code shows an array being loaded with the identity matrix and then being loaded into the modelview matrix stack. This example is equivalent to calling `glLoadIdentity` using the higher-level functions:

```
// Load an identity matrix
GLfloat m[] = { 1.0f, 0.0f, 0.0f, 0.0f,           // X Column
                 0.0f, 1.0f, 0.0f, 0.0f,           // Y Column
                 0.0f, 0.0f, 1.0f, 0.0f,           // Z Column
                 0.0f, 0.0f, 0.0f, 1.0f };        // Translation
glMatrixMode(GL_MODELVIEW);
glLoadMatrixf(m);
```

Although internally OpenGL implementations prefer column-major ordering (and for good reason!), OpenGL does provide functions to load a matrix in row-major ordering. The following two functions perform the transpose operation on the matrix when loading it on the matrix stack:

```
void glLoadTransposeMatrixf(GLfloat m);
```

and

```
void glLoadTransposeMatrixd(GLdouble m);
```

Performing Your Own Transformations

Let's look at an example now that shows how to create and load your own transformation matrix—the hard way! In the sample program TRANSFORM, we draw a *torus* (a doughnut-shaped object) in front of our viewing location and make it rotate in place. The function `DrawTorus` does the necessary math to generate the torus's geometry and takes as an argument a 4x4 transformation matrix to be applied to the vertices. We create the matrix and apply the transformation manually to each vertex to transform the torus. Let's start with the main rendering function in [Listing 4.4](#).

Listing 4.4. Code to Set Up the Transformation Matrix While Drawing

```
void RenderScene(void)
{
    GLTMatrix    transformationMatrix;    // Storage for rotation matrix
    static GLfloat yRot = 0.0f;           // Rotation angle for animation
    yRot += 0.5f;
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // Build a rotation matrix
    gltRotationMatrix(gltDegToRad(yRot), 0.0f, 1.0f, 0.0f,
                      transformationMatrix);
    transformationMatrix[12] = 0.0f;
    transformationMatrix[13] = 0.0f;
    transformationMatrix[14] = -2.5f;
    DrawTorus(transformationMatrix);
    // Do the buffer Swap
    glutSwapBuffers();
}
```

We begin by declaring storage for the matrix here:

```
GLTMatrix    transformationMatrix;    // Storage for rotation matrix
```

The data type `GLTMatrix` is of our own design and is simply a `typedef` declared in `gltools.h` for a floating-point array 16 elements long:

```
typedef GLfloat GLTMatrix[16];        // A column major 4x4 matrix of type GLfloat
```

The animation in this sample works by continually incrementing the variable `yRot` that represents the rotation around the y-axis. After clearing the color and depth buffer, we compose our transformation matrix as follows:

```
gltRotationMatrix(gltDegToRad(yRot), 0.0f, 1.0f, 0.0f, transformationMatrix);
transformationMatrix[12] = 0.0f;
transformationMatrix[13] = 0.0f;
transformationMatrix[14] = -2.5f;
```

Here, the first line contains a call to another `glTools` function, `gltRotationMatrix`. This function takes a rotation angle in radians (for more efficient calculations) and three arguments specifying a vector around which you want the rotation to occur. With the exception of the angle being in radians instead of degrees, this is almost exactly like the OpenGL function `glRotate`. The last argument is a matrix into which you want to store the resulting rotation matrix. The macro function `gltDegToRad` does an in-place conversion from degrees to radians.

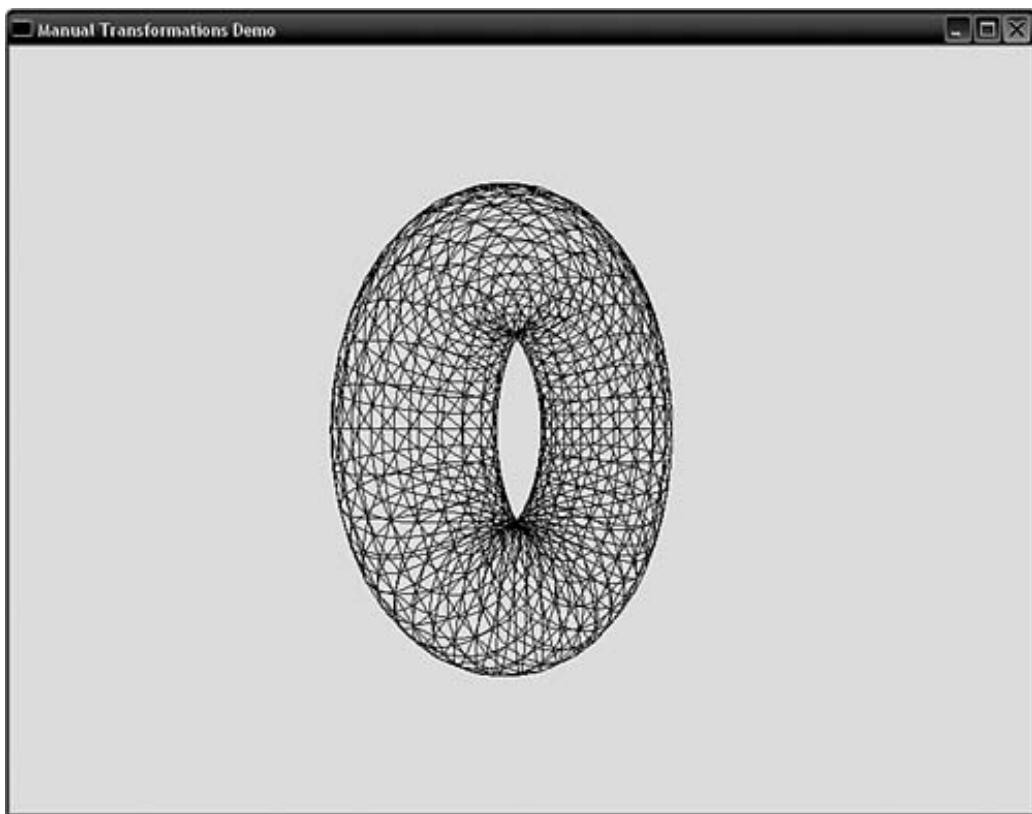
As you saw in [Figure 4.28](#), the last column of the matrix represents the translation of the transformation. Rather than do a full matrix multiplication, we can simply inject the desired translation directly into the matrix. Now the resulting matrix represents both a translation in space (a location to place the torus) and then a rotation of the object's coordinate system applied at that location.

Next, we pass this transformation matrix to the `DrawTorus` function. You do not need to list the entire function to create a torus here, but focus your attention to these lines:

```
objectVertex[0] = x0*r;
objectVertex[1] = y0*r;
objectVertex[2] = z;
glTransformPoint(objectVertex, mTransform, transformedVertex);
glVertex3fv(transformedVertex);
```

The three components of the vertex are loaded into an array and passed to the function `glTransformPoint`. This `glTools` function performs the multiplication of the vertex against the matrix and returns the transformed vertex in the array `transformedVertex`. We then use the vector version of `glVertex` and send the vertex data down to OpenGL. The result is a spinning torus, as shown in [Figure 4.29](#).

Figure 4.29. The spinning torus, doing our own transformations.



It is important that you see at least once the real mechanics of how vertices are transformed by a matrix using such a drawn-out example. As you progress as an OpenGL programmer, you will find that the need to transform points manually will arise for tasks that are not specifically related to rendering operations, such as collision detection (bumping into objects), frustum culling (throwing away and not drawing things you can't see), and some other special effects algorithms.

For geometry processing, however, the TRANSFORM sample program is very inefficient. We are letting the CPU do all the matrix math instead of letting OpenGL's dedicated hardware do the work for us (which is much faster than the CPU!). In addition, because OpenGL has the modelview matrix, all our transformed points are being multiplied yet again by the identity matrix. This does not change the value of our transformed vertices, but it is still a wasted operation.

For the sake of completeness, we provide an improved example, TRANSFORMGL, that instead uses our transformation matrix but hands it over to OpenGL using the function `glLoadMatrixf`. We eliminate our `DrawTorus` function with its dedicated transformation code and use a more general-

purpose torus drawing function, `gltDrawTorus`, from the `glTools` library. The relevant code is shown in [Listing 4.5](#).

Listing 4.5. Loading the Transformation Matrix Directly into OpenGL

```
// Build a rotation matrix
gltRotationMatrix(gltDegToRad(yRot), 0.0f, 1.0f, 0.0f,
                  transformationMatrix);

transformationMatrix[12] = 0.0f;
transformationMatrix[13] = 0.0f;
transformationMatrix[14] = -2.5f;
glLoadMatrixf(transformationMatrix);
gltDrawTorus(0.35, 0.15, 40, 20);
```

Adding Transformations Together

In the preceding example, we simply constructed a single transformation matrix and loaded it into the modelview matrix. This technique had the effect of transforming any and all geometry that followed by that matrix before being rendered. As you've seen in the other previous examples, we often add one transformation to another. For example, we used `glTranslate` followed by `glRotate` to first translate and then rotate an object before being drawn. Behind the scenes, when you call multiple transformation functions, OpenGL performs a matrix multiplication between the existing transformation matrix and the one you are adding or appending to it. For example, in the TRANSFORMGL example, we might replace the code in [Listing 4.5](#) with something like the following:

```
glPushMatrix();
    glTranslatef(0.0f, 0.0f, -2.5f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);
    gltDrawTorus(0.35, 0.15, 40, 20);
glPopMatrix();
```

Using this approach has the effect of saving the current identity matrix, multiplying the translation matrix, multiplying the rotation matrix, and then drawing the torus by the result. You can do these multiplications yourself by using the `glTools` function `gltMultiplyMatrix`, as shown here:

```
GLTMatrix rotationMatrix, translationMatrix, transformationMatrix;
...
    gltRotationMatrix(gltDegToRad(yRot), 0.0f, 1.0f, 0.0f, rotationMatrix);
    gltTranslationMatrix(0.0f, 0.0f, -2.5f, translationMatrix);
    gltMultiplyMatrix(translationMatrix, rotationMatrix, transformationMatrix);
    glLoadMatrixf(transformationMatrix);
    gltDrawTorus(0.35f, 0.15f, 40, 20);
```

OpenGL also has its own matrix multiplication function, `glMultMatrix`, that takes a matrix and multiplies it by the currently loaded matrix and stores the result at the top of the matrix stack. In our final code fragment, we once again show code equivalent to the preceding, but this time we let OpenGL do the actual multiplication:

```
GLTMatrix rotationMatrix, translationMatrix, transformationMatrix;
...
    glPushMatrix();
        gltRotationMatrix(gltDegToRad(yRot), 0.0f, 1.0f, 0.0f, rotationMatrix);
        gltTranslationMatrix(0.0f, 0.0f, -2.5f, translationMatrix);
        glMultMatrixf(translationMatrix);
        glMultMatrixf(rotationMatrix);
        gltDrawTorus(0.35f, 0.15f, 40, 20);
    glPopMatrix();
```

Once again, you should remember that the `glMultMatrix` functions and other high-level functions that do matrix multiplication (`glRotate`, `glScale`, `glTranslate`) are not being performed by the OpenGL hardware, but usually by your CPU.

Moving Around in OpenGL Using Cameras and Actors

To represent a location and orientation of any object in your 3D scene, you can use a single 4x4 matrix that represents its transform. Working with matrices directly, however, can still be somewhat awkward, so programmers have always sought ways to represent a position and orientation in space more succinctly. Fixed objects such as terrain are often untransformed, and their vertices usually specify exactly where the geometry should be drawn in space. Objects that move about in the scene are often called *actors*, paralleling the idea of actors on a stage.

Actors have their own transformations, and often other actors are transformed not only with respect to the world coordinate system (eye coordinates), but with respect to other actors. Each actor with its own transformation is said to have its own frame of reference, or local object coordinate system. It is often useful to translate between local and world coordinate systems and back again for many nonrendering-related geometric tests.

An Actor Frame

A simple and flexible way to represent a frame of reference is to use a data structure (or class in C++) that contains a position in space, a vector that points forward, and a vector that points upward. Using these quantities, you can uniquely identify a given position and orientation in space. The following example from the `glTools` library is a `GLFrame` data structure that can store this information all in one place:

```
typedef struct{
    GLTVector3f vLocation;
    GLTVector3f vUp;
    GLTVector3f vForward;
} GLTFrame;
```

Using a frame of reference such as this to represent an object's position and orientation is a very powerful mechanism. To begin with, you can use this data directly to create a 4x4 transformation matrix. Referring to [Figure 4.28](#), the up vector becomes the y column of the matrix, whereas the forward-looking vector becomes the z column vector and the position is the translation column vector. This leaves only the x column vector, and because we know all three axes are perpendicular to one another (orthonormal), we can calculate the x column vector by performing the cross product of the y and z vectors. [Listing 4.6](#) shows the `glTools` function `gltGetMatrixFromFrame`, which does exactly that.

Listing 4.6. Code to Derive a 4x4 Matrix from a Frame

```
////////////////////////////////////////////////////////////////
// Derives a 4x4 transformation matrix from a frame of reference
void gltGetMatrixFromFrame(GLTFrame *pFrame, GLTMatrix mMatrix)
{
    GLTVector3f vXAxis;           // Derived X Axis
    // Calculate X Axis
    gltVectorCrossProduct(pFrame->vUp, pFrame->vForward, vXAxis);
    // Just populate the matrix
    // X column vector
    memcpy(mMatrix, vXAxis, sizeof(GLTVector));
    mMatrix[3] = 0.0f;
    // y column vector
    memcpy(mMatrix+4, pFrame->vUp, sizeof(GLTVector));
    mMatrix[7] = 0.0f;
```

```

// z column vector
memcpy(mMatrix+8, pFrame->vForward, sizeof(GLTVector));
mMatrix[11] = 0.0f;
// Translation/Location vector
memcpy(mMatrix+12, pFrame->vLocation, sizeof(GLTVector));
mMatrix[15] = 1.0f;
}

```

Applying an actor's transform is as simple as calling `glMultMatrixf` with the resulting matrix.

Euler Angles: "Use the Frame, Luke!"

Many graphics programming books recommend an even simpler mechanism for storing an object's position and orientation: Euler angles. Euler angles require less space because you essentially store an object's position and then just three angles—representing a rotation around the x-, y-, and z-axes—sometimes called *yaw*, *pitch*, and *roll*. A structure such as this might represent an airplane's location and orientation:

```

struct EULER {
    GLTVector3f    vPosition;
    GLfloat        fRoll;
    GLfloat        fPitch;
    GLfloat        fYaw;
};

```

Euler angles are a bit slippery and are sometimes called "oily angles" by some in the industry. The first problem is that a given position and orientation can be represented by more than one set of Euler angles. Having multiple sets of angles can lead to problems as you try to figure out how to smoothly move from one orientation to another. Occasionally, a second problem called "gimbal lock" comes up; this problem makes it impossible to achieve a rotation around one of the axes. Lastly, Euler angles make it more tedious to calculate new coordinates for simply moving forward along your line of sight or trying to figure out new Euler angles if you want to rotate around one of your own local axes.

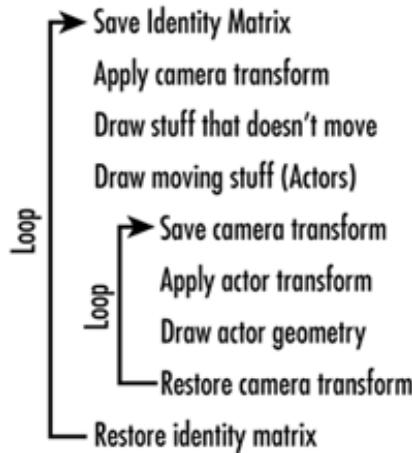
Some literature today tries to solve the problems of Euler angles by using a mathematical tool called *quaternions*. Quaternions, which can be difficult to understand, really don't solve any problems with Euler angles that you can't solve on your own by just using the frame of reference method covered previously. We already promised that this book would not get too heavy on the math, so we will not debate the merits of each system here. But we should say that the quaternion versus linear algebra (matrix) debate is more than 100 years old and by far predates their application to computer graphics!

Camera Management

There is really no such thing as a camera transformation in OpenGL. We use the camera as a useful metaphor to help us manage our point of view in some sort of immersive 3D environment. If we envision a camera as an object that has some position in space and some given orientation, we find that our current frame of reference system can represent both actors and our camera in a 3D environment.

To apply a camera transformation, we take the camera's actor transform and flip it so that moving the camera backward is equivalent to moving the whole world forward. Similarly, turning to the left is equivalent to rotating the whole world to the right. To render a given scene, we usually take the approach outlined in [Figure 4.30](#).

Figure 4.30. Typical rendering loop for a 3D environment.



The OpenGL utility library contains a function that uses the same data we stored in our frame structure to create our camera transformation:

```
void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez,
               GLdouble centerx, GLdouble centery, GLdouble centerz,
               GLdouble upx, GLdouble upy, GLdouble upz);
```

This function takes the position of the eye point, a point directly in front of the eye point, and the direction of the up vector. The `glTools` library also contains a shortcut function that performs the equivalent action using a frame of reference:

```
void gltApplyCameraTransform(GLTFrame *pCamera);
```

Bringing It All Together

Now let's work through one final example for this chapter to bring together all the concepts we have discussed so far. In the sample program SPHEREWORLD, we create a world populated by a number of spheres (Sphere World) placed at random locations on the ground. Each sphere is represented by an individual `GLTFrame` structure for its location and orientation. We also use the frame to represent a camera that can be moved about Sphere World using the keyboard arrow keys. In the middle of Sphere World, we use the simpler high-level transformation routines to draw a spinning torus with another sphere in orbit around it.

This example combines all the ideas we have discussed thus far and shows them working together. In addition to the main source file `sphereworld.c`, the project also includes the `torus.c`, `matrixmath.c`, and `framemath.c` modules from the `glTools` library found in the `\common` subdirectory. We do not provide the entire listing here because it uses the same GLUT framework as all the other samples, but the important functions are shown in [Listing 4.7](#).

Listing 4.7. Main Functions for the SPHEREWORLD Sample

```
#define NUM_SPHERES      50
GLTFrame    spheres[NUM_SPHERES];
GLTFrame    frameCamera;
///////////////////////////////
// This function does any needed initialization on the rendering
// context.
void SetupRC()
{
    int iSphere;
    // Bluish background
    glClearColor(0.0f, 0.0f, .50f, 1.0f );
```

```

// Draw everything as wire frame
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
gltInitFrame(&frameCamera); // Initialize the camera
// Randomly place the sphere inhabitants
for(iSphere = 0; iSphere < NUM_SPHERES; iSphere++)
{
    gltInitFrame(&spheres[iSphere]); // Initialize the frame
    // Pick a random location between -20 and 20 at .1 increments
    spheres[iSphere].vLocation[0] = (float)((rand() % 400) - 200) * 0.1f;
    spheres[iSphere].vLocation[1] = 0.0f;
    spheres[iSphere].vLocation[2] = (float)((rand() % 400) - 200) * 0.1f;
}
}

///////////////////////////////
// Draw a gridded ground
void DrawGround(void)
{
    GLfloat fExtent = 20.0f;
    GLfloat fStep = 1.0f;
    GLfloat y = -0.4f;
    GLint iLine;
    glBegin(GL_LINES);
    for(iLine = -fExtent; iLine <= fExtent; iLine += fStep)
    {
        glVertex3f(iLine, y, fExtent); // Draw Z lines
        glVertex3f(iLine, y, -fExtent);
        glVertex3f(fExtent, y, iLine);
        glVertex3f(-fExtent, y, iLine);
    }
    glEnd();
}
// Called to draw scene
void RenderScene(void)
{
    int i;
    static GLfloat yRot = 0.0f; // Rotation angle for animation
    yRot += 0.5f;
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix();
    gltApplyCameraTransform(&frameCamera);
    // Draw the ground
    DrawGround();
    // Draw the randomly located spheres
    for(i = 0; i < NUM_SPHERES; i++)
    {
        glPushMatrix();
        gltApplyActorTransform(&spheres[i]);
        glutSolidSphere(0.1f, 13, 26);
        glPopMatrix();
    }
    glPushMatrix();
    glTranslatef(0.0f, 0.0f, -2.5f);
    glPushMatrix();
        glRotatef(-yRot * 2.0f, 0.0f, 1.0f, 0.0f);
        glTranslatef(1.0f, 0.0f, 0.0f);
        glutSolidSphere(0.1f, 13, 26);
    glPopMatrix();
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);
    gltDrawTorus(0.35, 0.15, 40, 20);
}

```

```

    glPopMatrix();
    glPopMatrix();
    // Do the buffer Swap
    glutSwapBuffers();
}

// Respond to arrow keys by moving the camera frame of reference
void SpecialKeys(int key, int x, int y)
{
    if(key == GLUT_KEY_UP)
        gltMoveFrameForward(&frameCamera, 0.1f);
    if(key == GLUT_KEY_DOWN)
        gltMoveFrameForward(&frameCamera, -0.1f);
    if(key == GLUT_KEY_LEFT)
        gltRotateFrameLocally(&frameCamera, 0.1);
    if(key == GLUT_KEY_RIGHT)
        gltRotateFrameLocally(&frameCamera, -0.1);
    // Refresh the Window
    glutPostRedisplay();
}

```

The first few lines contain a macro to define the number of spherical inhabitants as 50. Then we declare an array of frames and another frame to represent the camera:

```

#define NUM_SPHERES      50
GLTFrame    spheres[NUM_SPHERES];
GLTFrame    frameCamera;

```

The `SetupRC` function calls the `glTools` function `gltInitFrame` on the camera to initialize it as being at the origin and pointing down the negative z-axis (the OpenGL default viewing orientation):

```
gltInitFrame(&frameCamera); // Initialize the camera
```

You can use this function to initialize any frame structure, or you can initialize the structure yourself to have any desired position and orientation. Next, a loop initializes the array of sphere frames and selects a random x and z location for their positions:

```

// Randomly place the sphere inhabitants
for(iSphere = 0; iSphere < NUM_SPHERES; iSphere++)
{
    gltInitFrame(&spheres[iSphere]); // Initialize the frame
    // Pick a random location between -20 and 20 at .1 increments
    spheres[iSphere].vLocation[0] = (float)((rand() % 400) - 200) * 0.1f;
    spheres[iSphere].vLocation[1] = 0.0f;
    spheres[iSphere].vLocation[2] = (float)((rand() % 400) - 200) * 0.1f;
}

```

The `DrawGround` function then draws the ground as a series of criss-cross grids using a series of `GL_LINE` segments:

```

///////////////////////////////
// Draw a gridded ground
void DrawGround(void)
{
    GLfloat fExtent = 20.0f;
    GLfloat fStep = 1.0f;
    GLfloat y = -0.4f;
    GLint iLine;
    glBegin(GL_LINES);

```

```

for(iLine = -fExtent; iLine <= fExtent; iLine += fStep)
{
    glVertex3f(iLine, y, fExtent);      // Draw Z lines
    glVertex3f(iLine, y, -fExtent);
    glVertex3f(fExtent, y, iLine);
    glVertex3f(-fExtent, y, iLine);
}
glEnd();
}

```

The `RenderScene` function draws the world from our point of view. Note that we first save the identity matrix and then apply the camera transformation using the `glTools` helper function `gltApplyCameraTransform`. The ground is static and is transformed by the camera only to appear that you are moving over it:

```

glPushMatrix();
    gltApplyCameraTransform(&frameCamera);
    // Draw the ground
    DrawGround();

```

Then we draw each of the randomly located spheres. The `gltApplyActorTransform` function creates a transformation matrix from the frame of reference and multiplies it by the current matrix (which is the camera matrix). Each sphere must have its own transform relative to the camera, so the camera is saved each time with a call to `glPushMatrix` and restored again with `glPopMatrix` to get ready for the next sphere or transformation:

```

// Draw the randomly located spheres
for(i = 0; i < NUM_SPHERES; i++)
{
    glPushMatrix();
    gltApplyActorTransform(&spheres[i]);
    glutSolidSphere(0.1f, 13, 26);
    glPopMatrix();
}

```

Now for some fancy footwork! First, we move the coordinate system a little further down the z-axis so that we can see what we are going to draw next. We save this location and then perform a rotation, followed by a translation and the drawing of a sphere. This effect makes the sphere appear to revolve around the origin in front of us. We then restore our transformation matrix, but only so that the location of the origin is $z = -2.5$. Then another rotation is performed before the torus is drawn. This has the effect of making a torus that spins in place:

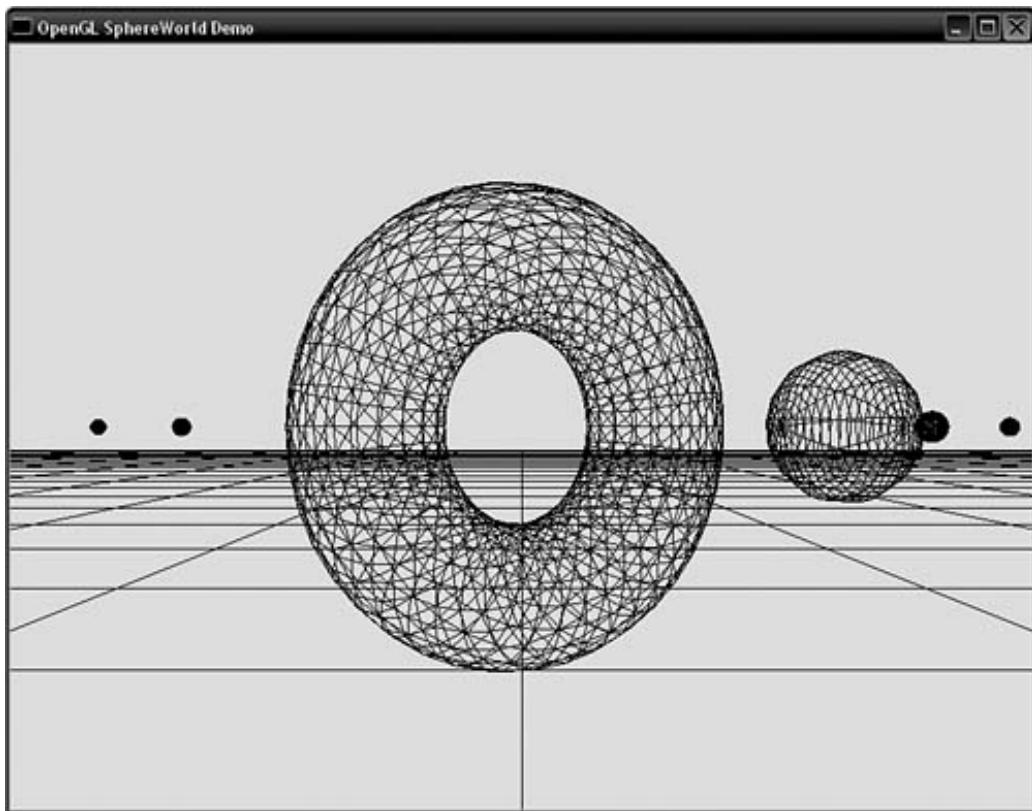
```

glPushMatrix();
    glTranslatef(0.0f, 0.0f, -2.5f);
    glPushMatrix();
        glRotatef(-yRot * 2.0f, 0.0f, 1.0f, 0.0f);
        glTranslatef(1.0f, 0.0f, 0.0f);
        glutSolidSphere(0.1f, 13, 26);
    glPopMatrix();
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);
    gltDrawTorus(0.35, 0.15, 40, 20);
    glPopMatrix();
glPopMatrix();

```

The total effect is that we see a grid on the ground with many spheres scattered about at random locations. Out in front, we see a spinning torus, with a sphere moving rapidly in orbit around it. [Figure 4.31](#) shows the result.

Figure 4.31. The output from the SPHEREWORLD program.



Finally, the `SpecialKeys` function is called whenever one of the arrow keys is pressed. The up-and down-arrow keys call the `glTools` function `gltMoveFrameForward`, which simply moves the frame forward along its line of sight. The `gltRotateFrameLocally` function rotates a frame of reference around its local y-axis (regardless of orientation) in response to the left-and right-arrow keys:

```
void SpecialKeys(int key, int x, int y)
{
    if(key == GLUT_KEY_UP)
        gltMoveFrameForward(&frameCamera, 0.1f);
    if(key == GLUT_KEY_DOWN)
        gltMoveFrameForward(&frameCamera, -0.1f);
    if(key == GLUT_KEY_LEFT)
        gltRotateFrameLocally(&frameCamera, 0.1);
    if(key == GLUT_KEY_RIGHT)
        gltRotateFrameLocally(&frameCamera, -0.1);
    // Refresh the Window
    glutPostRedisplay();
}
```

A NOTE ON KEYBOARD POLLING

Moving the camera in response to keystroke messages can sometimes result in less than the smoothest possible animation. The reason is that the keyboard repeat rate is usually no more than about 20 times per second. For best results, you should render at least 30 frames per second (with 60 being more optimal) and poll the keyboard once for each frame of animation. Doing this with a portability library like GLUT is somewhat tricky, but in the OS-specific chapters later in this book, we will cover ways to achieve the smoothest possible animation and methods to best create time-based animation instead of the frame-based animation (moving by a fixed amount each time the scene is redrawn) done here.

Summary

In this chapter, you learned concepts crucial to using OpenGL for creation of 3D scenes. Even if you can't juggle matrices in your head, you now know what matrices are and how they are used to perform the various transformations. You also learned how to manipulate the modelview and projection matrix stacks to place your objects in the scene and to determine how they are viewed onscreen.

We also showed you the functions needed to perform your own matrix magic, if you are so inclined. These functions allow you to create your own matrices and load them on to the matrix stack or multiply them by the current matrix first. The chapter also introduced the powerful concept of a frame of reference, and you saw how easy it is to manipulate frames and convert them into transformations.

Finally, we began to make more use of the `glTools` library that accompanies this book. This library is written entirely in portable ANSI C and provides you with a handy toolkit of miscellaneous math and helper routines that can be used along with OpenGL.

Reference

glFrustum

Purpose: Multiplies the current matrix by a perspective matrix.

Include File: `<gl.h>`

Syntax:

```
void glFrustum(GLdouble left, GLdouble right,
  GLdouble bottom, GLdouble top, GLdouble zNear,
  GLdouble zFar);
```

Description: This function creates a perspective matrix that produces a perspective projection. The eye is assumed to be located at (0,0,0), with `zFar` being the distance of the far clipping plane and `zNear` specifying the distance to the near clipping plane. Both values must be positive. This function can adversely affect the precision of the depth buffer if the ratio of far to near (`far/near`) is large.

Parameters:

`left, right` `GLdouble`: Coordinates for the left and right clipping planes.

`bottom, top` `GLdouble`: Coordinates for the bottom and top clipping planes.

`zNear, zFar` `GLdouble`: Distance to the near and far clipping planes. Both of these values must be positive.

Returns: None.

See Also: `glOrtho`, `glMatrixMode`, `glMultMatrix`, `glViewport`

glLoadIdentity

Purpose: Sets the current matrix to identity.

Include File: `<gl.h>`**Syntax:**`void glLoadIdentity(void);`**Description:** This function replaces the current transformation matrix with the identity matrix. This essentially resets the coordinate system to eye coordinates.**Returns:** None.**See Also:** `glLoadMatrix`, `glMatrixMode`, `glMultMatrix`, `glPushMatrix`

glLoadMatrix

Purpose: Sets the current matrix to the one specified.**Include File:** `<gl.h>`**Variations:**`void glLoadMatrixd(const GLdouble *m);`
`void glLoadMatrixf(const GLfloat *m);`**Description:** This function replaces the current transformation matrix with an arbitrary matrix supplied. Using some of the other matrix manipulation functions, such as `glLoadIdentity`, `glRotate`, `glTranslate`, and `glScale`, might be more efficient.**Parameters:**`*m` **GLdouble** or **GLfloat**: This array represents a 4x4 matrix that will be used for the current transformation matrix. The array is stored in column-major order as 16 consecutive values.**Returns:** None.**See Also:** `glLoadIdentity`, `glMatrixMode`, `glMultMatrix`, `glPushMatrix`

glLoadTransposeMatrix

Purpose: Allows a transposed 4x4 matrix to be loaded onto the matrix stack.**Include File:** `<gl.h>`**Variations:**`void LoadTransposeMatrixf(GLfloat *m);`
`void LoadTransposeMatrixd(GLdouble *m);`

Description: OpenGL uses 4x4 matrices in a single one-dimensional column-major array. A row-major ordered array, which is the transpose of the column-major array, may be loaded onto the stack using this function. This function takes the matrix, transposes it, and then loads a properly formatted array to the top of the current matrix stack. Some OpenGL libraries may not export this function, even if the implementation supports it. In this case, a pointer to this function may be obtained via the OpenGL extension mechanism.

Parameters:

m* **GLfloat or **GLdouble**: 4x4 transposed matrix to be loaded.

Returns: None.

See Also: [glLoadMatrix](#), [glMultTransposeMatrix](#)

glMatrixMode

Purpose: Specifies the current matrix (**GL_PROJECTION**, **GL_MODELVIEW**, or **GL_TEXTURE**).

Include File: `<gl.h>`

Syntax:

```
void glMatrixMode(GLenum mode);
```

Description: This function determines which matrix stack (**GL_MODELVIEW**, **GL_PROJECTION**, or **GL_TEXTURE**) is used for matrix operations.

Parameters:

mode **GLenum**: Identifies which matrix stack is used for subsequent matrix operations. Any of the values in [Table 4.2](#) are accepted.

Table 4.2. Valid Matrix Mode Identifiers for `glMatrixMode`

Mode	Matrix Stack
GL_MODELVIEW	Matrix operations affect the modelview matrix stack. (Used to move objects around the scene.)
GL_PROJECTION	Matrix operations affect the projection matrix stack. (Used to define clipping volume.)
GL_TEXTURE	Matrix operations affect the texture matrix stack. (Manipulates texture coordinates.)

Returns: None.

See Also: [glLoadMatrix](#), [glPushMatrix](#)

glMultMatrix**Purpose:** Multiplies the current matrix by the one specified.**Include File:** `<gl.h>`**Variations:**

```
void glMultMatrixd(const GLdouble *m);
void glMultMatrixf(const GLfloat *m);
```

Description: This function multiplies the currently selected matrix stack with the one specified. The resulting matrix is then stored as the current matrix at the top of the matrix stack.**Parameters:**

m* **GLdouble or **GLfloat**: This array represents a 4x4 matrix that will be multiplied by the current matrix. The array is stored in column-major order as 16 consecutive values.

Returns: None.**See Also:** `glMatrixMode`, `glLoadIdentity`, `glLoadMatrix`, `glPushMatrix`**glMultTransposeMatrix****Purpose:** Allows a transposed 4x4 matrix to be multiplied onto the matrix stack.**Include File:** `<gl.h>`**Variations:**

```
void MultTransposeMatrixf(GLfloat *m);
void MultTransposeMatrixd(GLdouble *m);
```

Description: OpenGL uses 4x4 matrices in a single one-dimensional column-major array. A row major ordered array, which is the transpose of the column major array, may be multiplied onto the stack using this function. This function takes the matrix, transposes it, and then multiplies a properly formatted array with the top of the current matrix stack. Some OpenGL libraries may not export this function, even if the implementation supports it. In this case, a pointer to this function may be obtained via the OpenGL extension mechanism.**Parameters:**

m* **GLfloat or **GLdouble**: 4x4 transposed matrix to be multiplied onto the current stack.

Returns: None.**See Also:** `glMultMatrix`, `glLoadTransposeMatrix`

glPopMatrix**Purpose:** Pops the current matrix off the matrix stack.**Include File:** `<gl.h>`**Syntax:**`void glPopMatrix(void);`**Description:** This function pops the last (topmost) matrix off the current matrix stack. This function is most often used to restore the previous condition of the current transformation matrix if it was saved with a call to `glPushMatrix`.**Returns:** None.**See Also:** `glPushMatrix`**glPushMatrix****Purpose:** Pushes the current matrix onto the matrix stack.**Include File:** `<gl.h>`**Syntax:**`void glPushMatrix(void);`**Description:** This function pushes the current matrix onto the current matrix stack. This function is most often used to save the current transformation matrix so that it can be restored later with a call to `glPopMatrix`.**Returns:** None.**See Also:** `glPopMatrix`**glRotate****Purpose:** Rotates the current matrix by a rotation matrix.**Include File:** `<gl.h>`**Variations:**

```
void glRotated(GLdouble angle, GLdouble x,
  GLdouble y, GLdouble z);
void glRotatef(GLfloat angle, GLfloat x, GLfloat y
  , GLfloat z);
```

Description: This function multiplies the current matrix by a rotation matrix that performs a counterclockwise rotation around a directional vector that passes from the origin through the point (x,y,z). The newly rotated matrix becomes the current transformation matrix.

Parameters:

`angle` **GLdouble** or **GLfloat**: The angle of rotation in degrees. The angle produces a counterclockwise rotation.

`x, y, z` **GLdouble** or **GLfloat**: A direction vector from the origin that is used as the axis of rotation.

Returns: None.

See Also: `glScale`, `glTranslate`

glScale

Purpose: Multiplies the current matrix by a scaling matrix.

Include File: `<gl.h>`

Variations:

```
void glScaled(GLdouble x, GLdouble y, GLdouble z);
void glScalef(GLfloat x, GLfloat y, GLfloat z);
```

Description: This function multiplies the current matrix by a scaling matrix. The newly scaled matrix becomes the current transformation matrix.

Parameters:

`x, y, z` **GLdouble** or **GLfloat**: Scale factors along the x-, y-, and z-axes.

Returns: None.

See Also: `glRotate`, `glTranslate`

glTranslate

Purpose: Multiplies the current matrix by a translation matrix.

Include File: `<gl.h>`

Variations:

```
void glTranslated(GLdouble x, GLdouble y, GLdouble z);
void glTranslatef(GLfloat x, GLfloat y, GLfloat z);
```

Description: This function multiplies the current matrix by a translation matrix. The newly translated matrix becomes the current transformation matrix.

Parameters:

`x, y, z` **GLdouble** or **GLfloat**: The x, y, and z coordinates of a translation vector.

Returns: None.**See Also:** [glRotate](#), [glScale](#)

gluLookAt

Purpose: Defines a viewing transformation.**Include File:** [<glu.h>](#)**Syntax:**

```
void gluLookAt(GLdouble eyex, GLdouble eyey,
  GLdouble eyez, GLdouble centerx,
  GLdouble centery, GLdouble centerz, GLdouble upx,
  GLdouble upy, GLdouble upz );
```

Description: Defines a viewing transformation based on the position of the eye, the position of the center of the scene, and a vector pointing up from the viewer's perspective.

Parameters:*eyex, eyey, eyez* **GLdouble:** X, y, and z coordinates of the eye point.*centerx, centery, centerz* **GLdouble:** X, y, and z coordinates of the center of the scene being looked at.*upx, upy, upz* **GLdouble:** X, y, and z coordinates that specify the up vector.**Returns:** None.**See Also:** [glFrustum](#), [gluPerspective](#)

gluOrtho2D

Purpose: Defines a two-dimensional orthographic projection.**Include File:** [<glu.h>](#)**Syntax:**

```
void gluOrtho2D(GLdouble left, GLdouble right,
  GLdouble bottom, GLdouble top);
```

Description: This function defines a 2D orthographic projection matrix. This projection matrix is equivalent to calling [glOrtho](#) with *near* and *far* set to 0 and 1, respectively.

Parameters:*left, right* **GLdouble:** Specifies the far-left and far-right clipping planes.*bottom, top* **GLdouble:** Specifies the top and bottom clipping planes.**Returns:** None.

See Also: [glOrtho](#), [gluPerspective](#)

gluPerspective

Purpose: Defines a viewing perspective projection matrix.

Include File: `<glu.h>`

Syntax:

```
void gluPerspective(GLdouble fovy, GLdouble aspect
→ , GLdouble zNear, GLdouble zFar);
```

Description: This function creates a matrix that describes a viewing frustum in world coordinates. The aspect ratio should match the aspect ratio of the viewport (specified with [glViewport](#)). The perspective division is based on the field-of-view angle and the distance to the near and far clipping planes.

Parameters:

<i>fovy</i>	<code>GLdouble</code> : The field of view in degrees, in the y direction.
<i>aspect</i>	<code>GLdouble</code> : The aspect ratio. This is used to determine the field of view in the x direction. The aspect ratio is x/y.
<i>zNear</i> , <i>zFar</i>	<code>GLdouble</code> : The distance from the viewer to the near and far clipping plane. These values are always positive.

Returns: None.

See Also: [glFrustum](#), [gluOrtho2D](#)

Chapter 5. Color, Materials, and Lighting: The Basics

By Richard S. Wright, Jr.

WHAT YOU'LL LEARN IN THIS CHAPTER:

How To	Functions You'll Use
Specify a color in terms of RGB components	<code> glColor</code>
Set the shading model	<code>glShadeModel</code>
Set the lighting model	<code>glLightModel</code>
Set lighting parameters	<code>glLight</code>
Set material reflective properties	<code>glColorMaterial/glMaterial</code>
Use surface normals	<code>glNormal</code>

This is the chapter where 3D graphics really start to look interesting (unless you really dig wireframe models!), and it only gets better from here. You've been learning OpenGL from the

ground up—how to put programs together and then how to assemble objects from primitives and manipulate them in 3D space. Until now, we've been laying the foundation, and you still can't tell what the house is going to look like! To recoin a phrase, "Where's the beef?"

To put it succinctly, the beef starts here. For most of the rest of this book, science takes a back seat and magic rules. According to Arthur C. Clarke, "Any sufficiently advanced technology is indistinguishable from magic." Of course, there is no real magic involved in color and lighting, but it sure can seem that way at times. If you want to dig into the "sufficiently advanced technology" (mathematics), see [Appendix A](#), "Further Reading."

Another name for this chapter might be "Adding Realism to Your Scenes." You see, there is more to an object's color in the real world than just what color we might tell OpenGL to make it. In addition to having a color, objects can appear shiny or dull or can even glow with their own light. An object's apparent color varies with bright or dim lighting, and even the color of the light hitting an object makes a difference. An illuminated object can even be shaded across its surface when lit or viewed from an angle.

What Is Color?

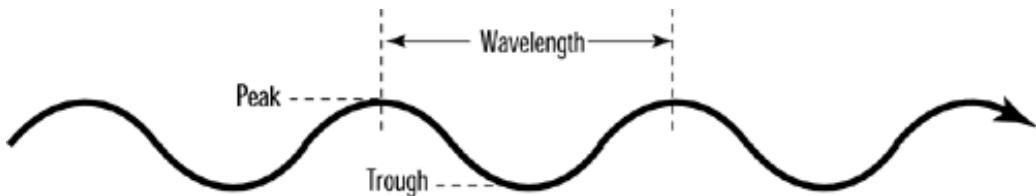
First let's talk a little bit about color itself. How is a color made in nature, and how do we see colors? Understanding color theory and how the human eye sees a color scene will lend some insight into how you create a color programmatically. (If color theory is old hat to you, you can probably skip this section.)

Light as a Wave

Color is simply a wavelength of light that is visible to the human eye. If you had any physics classes in school, you might remember something about light being both a wave and a particle. It is modeled as a wave that travels through space much like a ripple through a pond, and it is modeled as a particle, such as a raindrop falling to the ground. If this concept seems confusing, you know why most people don't study quantum mechanics!

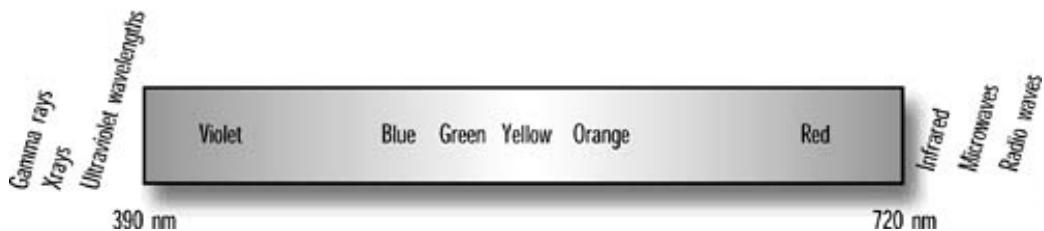
The light you see from nearly any given source is actually a mixture of many different kinds of light. These kinds of light are identified by their wavelengths. The wavelength of light is measured as the distance between the peaks of the light wave, as illustrated in [Figure 5.1](#).

Figure 5.1. How a wavelength of light is measured.



Wavelengths of visible light range from 390 nanometers (one billionth of a meter) for violet light to 720 nanometers for red light; this range is commonly called the *visible spectrum*. You've undoubtedly heard the terms *ultraviolet* and *infrared*; they represent light not visible to the naked eye, lying beyond the ends of the spectrum. You will recognize the spectrum as containing all the colors of the rainbow (see [Figure 5.2](#)).

Figure 5.2. The spectrum of visible light.



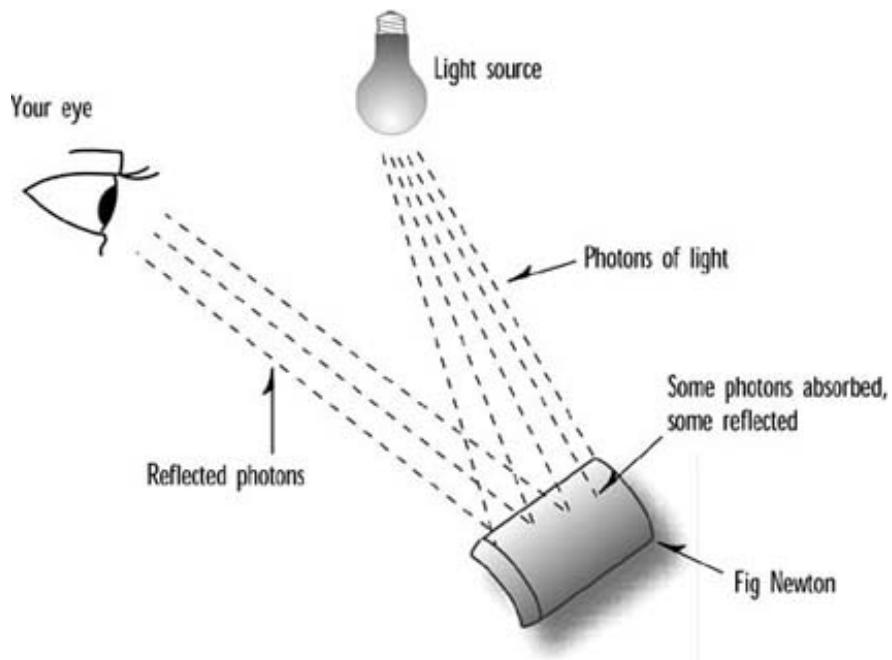
Light as a Particle

"Okay, Mr. Smart Brain," you might ask. "If color is a wavelength of light and the only visible light is in this 'rainbow' thing, where is the brown for my Fig Newtons or the black for my coffee or even the white of this page?" We begin answering that question by telling you that black is not a color, nor is white. Actually, black is the absence of color, and white is an even combination of all the colors at once. That is, a white object reflects all wavelengths of colors evenly, and a black object absorbs all wavelengths evenly.

As for the brown of those fig bars and the many other colors that you see, they are indeed colors. Actually, at the physical level, they are composite colors. They are made of varying amounts of the "pure" colors found in the spectrum. To understand how this concept works, think of light as a particle. Any given object when illuminated by a light source is struck by "billions and billions" (my apologies to the late Carl Sagan) of photons, or tiny light particles. Remembering our physics mumbo jumbo, each of these photons is also a wave, which has a wavelength and thus a specific color in the spectrum.

All physical objects consist of atoms. The reflection of photons from an object depends on the kinds of atoms, the number of each kind, and the arrangement of atoms (and their electrons) in the object. Some photons are reflected and some are absorbed (the absorbed photons are usually converted to heat), and any given material or mixture of materials (such as your fig bar) reflects more of some wavelengths than others. Figure 5.3 illustrates this principle.

Figure 5.3. An object reflects some photons and absorbs others.



Your Personal Photon Detector

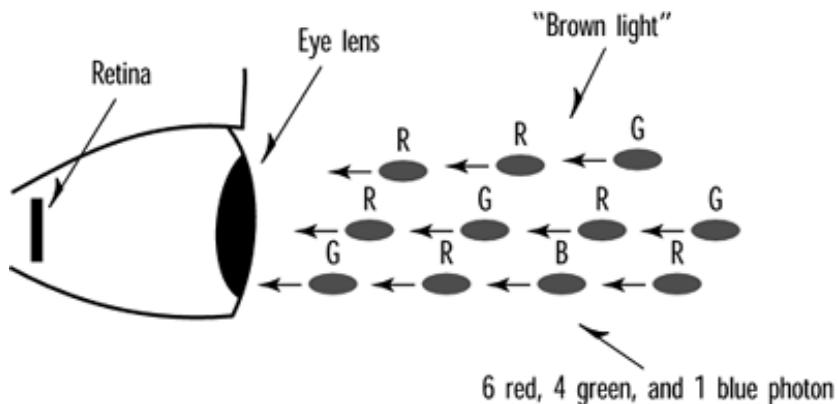
The reflected light from your fig bar, when seen by your eye, is interpreted as color. The billions of photons enter your eye and are focused onto the back of your eye, where your retina acts as sort

of a photographic plate. The retina's millions of cone cells are excited when struck by the photons, and this causes neural energy to travel to your brain, which interprets the information as light and color. The more photons that strike the cone cells, the more excited they get. This level of excitation is interpreted by your brain as the brightness of the light, which makes sense; the brighter the light, the more photons there are to strike the cone cells.

The eye has three kinds of cone cells. All of them respond to photons, but each kind responds most to a particular wavelength. One is more excited by photons that have reddish wavelengths; one, by green wavelengths; and one, by blue wavelengths. Thus, light that is composed mostly of red wavelengths excites red-sensitive cone cells more than the other cells, and your brain receives the signal that the light you are seeing is mostly reddish. You do the math: A combination of different wavelengths of various intensities will, of course, yield a mix of colors. All wavelengths equally represented thus are perceived as white, and no light of any wavelength is black.

You can see that any "color" that your eye perceives actually consists of light all over the visible spectrum. The "hardware" in your eye detects what it sees in terms of the relative concentrations and strengths of red, green, and blue light. Figure 5.4 shows how brown is composed of a photon mix of 60% red photons, 40% green photons, and 10% blue photons.

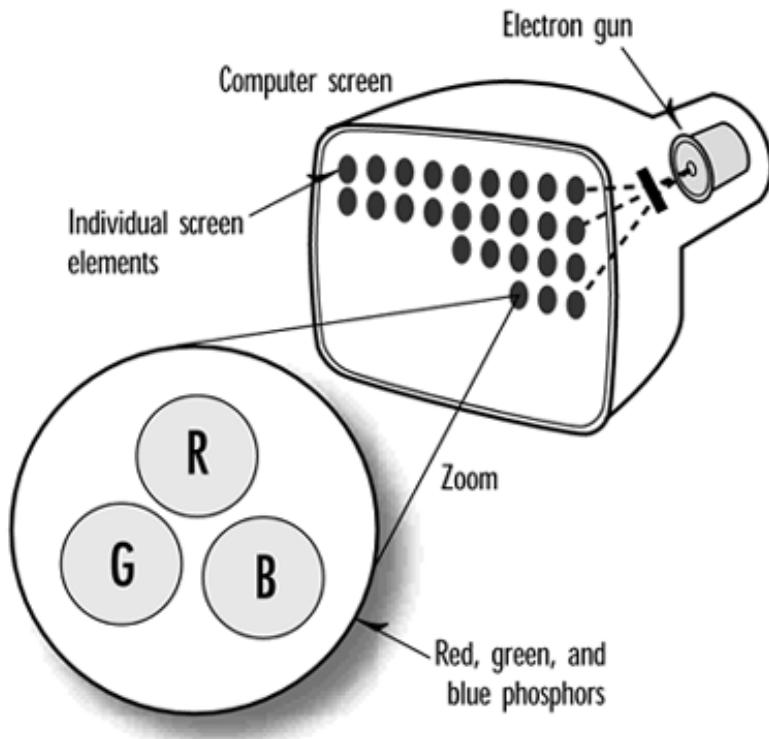
Figure 5.4. How the "color" brown is perceived by the eye.



The Computer as a Photon Generator

Now that you understand how the human eye discerns colors, it makes sense that when you want to generate a color with a computer, you do so by specifying separate intensities for the red, green, and blue components of the light. It so happens that color computer monitors are designed to produce three kinds of light (can you guess which three?), each with varying degrees of intensity. In the back of your computer monitor is an electron gun that shoots electrons at the back of the screen you view. This screen contains phosphors that emit red, green, and blue light when struck by the electrons. The intensity of the light emitted varies with the intensity of the electron beam. (Okay, then, how do color LCDs work? We leave this question as an exercise for you!) These three color phosphors are packed closely together to make up a single physical dot on the screen (see Figure 5.5).

Figure 5.5. How a computer monitor generates colors.



You might recall that in [Chapter 2](#), "Using OpenGL," we explained how OpenGL defines a color exactly as intensities of red, green, and blue, with the `glColor` command.

PC Color Hardware

There once was a time when state-of-the-art PC graphics hardware meant the Hercules graphics card. This card could produce bitmapped images with a resolution of 720x348. The drawback was that each pixel had only two states: on and off. At that time, bitmapped graphics of any kind on a PC were a big deal, and you could produce some great monochrome graphics—even 3D!

Actually predating the Hercules card was the Color Graphics Adapter (CGA) card. Introduced with the first IBM PC, this card could support resolutions of 320x200 pixels and could place any 4 of 16 colors on the screen at once. A higher resolution (640x200) with 2 colors was also possible but wasn't as effective or cost conscious as the Hercules card. (Color monitors = \$\$\$.) CGA was puny by today's standards; it was even outmatched by the graphics capabilities of a \$200 Commodore 64 or Atari home computer at the time. Lacking adequate resolution for business graphics or even modest modeling, CGA was used primarily for simple PC games or business applications that could benefit from colored text. Generally, it was hard to make a good business justification for this more expensive hardware.

The next big breakthrough for PC graphics came when IBM introduced the Enhanced Graphics Adapter (EGA) card. This one could do more than 25 lines of colored text in new text modes, and for graphics, it could support 640x350-pixel bitmapped graphics in 16 colors! Other technical improvements eliminated some flickering problems of the CGA ancestor and provided for better and smoother animation. Now arcade-style games, real business graphics, and even simple 3D graphics became not only possible but even reasonable on the PC. This advance was a giant move beyond CGA, but still PC graphics were in their infancy.

The last mainstream PC graphics standard set by IBM was the VGA card (which stood for Vector Graphics Array rather than the commonly held Video Graphics Adapter). This card was significantly faster than the EGA; it could support 16 colors at a higher resolution (640x480) and 256 colors at a lower resolution of 320x200. These 256 colors were selected from a palette of more than 16 million possible colors. That's when the floodgates opened for PC graphics. Near photo-realistic graphics became possible on PCs. Ray tracers, 3D games, and photo-editing software began to pop up in the PC market.

IBM, as well, had a high-end graphics card—the 8514—for its "workstations." This card could do 1,024x768 graphics at 256 colors. IBM thought this card would be used only by CAD and scientific applications! But one thing is certain about consumers: They always want more. It was this short-sightedness that cost IBM its role as standard setter in the PC graphics market. Other vendors began to ship "Super-VGA" cards that could display higher and higher resolutions, with more and more colors. First, we saw 800x600, then 1,024x768 and even higher, with first 256 colors, and then 32,000, and 65,000. Today, 24-bit color cards can display 16 million colors at resolutions far greater than 1,024x768. Even entry-level Windows PCs sold today can support at least 16 million colors at resolutions of 1,024x768 or more.

All this power makes for some really cool possibilities—photo-realistic 3D graphics, to name just one. When Microsoft ported OpenGL to the Windows platform, that move enabled creation of high-end graphics applications for PCs. Combine today's fast processors with 3D-graphics accelerated graphics cards, and you can get the kind of performance possible only a few years ago on \$100,000 graphics workstations—for the cost of a Wal-Mart Christmas special! Today's typical home machines are capable of sophisticated simulations, games, and more. Already the term *virtual reality* has become as antiquated as those old Buck Rogers rocket ships as we begin to take advanced 3D graphics for granted.

PC Display Modes

Microsoft Windows and the Apple Macintosh revolutionized the world of PC graphics in two respects. First, they created mainstream graphical operating environments that were adopted by the business world at large and, soon thereafter, the consumer market. Second, they made PC graphics significantly easier for programmers to do. The graphics hardware was "virtualized" by display device drivers. Instead of having to write instructions directly to the video hardware, programmers today can write to a single API (such as OpenGL!), and the operating system handles the specifics of talking to the hardware.

Screen Resolution

Screen resolution for today's computers can vary from 640x480 pixels up to 1,600x1,200 or more. The lower resolutions of, say, 640x480 are considered adequate for some graphics display tasks, and people with eye problems often run at the lower resolutions, but on a large monitor or display. You must always take into account the size of the window with the clipping volume and viewport settings (see [Chapter 2](#)). By scaling the size of the drawing to the size of the window, you can easily account for the various resolutions and window size combinations that can occur. Well-written graphics applications display the same approximate image regardless of screen resolution. The user should automatically be able to see more and sharper details as the resolution increases.

Color Depth

If an increase in screen resolution or in the number of available drawing pixels in turn increases the detail and sharpness of the image, so too should an increase in available colors improve the clarity of the resulting image. An image displayed on a computer that can display millions of colors should look remarkably better than the same image displayed with only 16 colors. In programming, you really need to worry about only three color depths: 4-bit, 8-bit, and 24-bit.

The 4-Bit Color Mode

On the low end, your program might run in 16 colors—called 4-bit mode because 4 bits are devoted to color information for each pixel. These 4 bits represent a value from 0 to 15 that provides an index into a set of 16 predefined colors. (When you have a limited number of colors that are accessed by an index, this is called a *palette*.) With only 16 colors at your disposal, you can do little to improve the clarity and sharpness of your image. It is generally accepted that most serious graphics applications can safely ignore the 16-color mode. We can be thankful that most of the newer display hardware available does not support this display mode any longer.

The 8-Bit Color Mode

The 8-bit color mode supports up to 256 colors on the screen. This is a substantial improvement over 4-bit color, but still limiting. Most PC OpenGL hardware accelerators do not accelerate 8-bit color, but for software rendering, you can obtain satisfactory results under Windows with certain considerations. The most important consideration is the construction of the correct color palette. This topic is covered briefly in [Chapter 13](#), "Wiggle: OpenGL on Windows."

The 24-Bit Color Mode

The best quality image production available today on PCs is 24-bit color mode. In this mode, a full 24 bits are devoted to each pixel to hold 8 bits of color data for each of the red, green, and blue color components ($8 + 8 + 8 = 24$). You have the capability to put any of more than 16 million possible colors in every pixel on the screen. The most obvious drawback to this mode is the amount of memory required for high-resolution screens (more than 2MB for a 1,024x768 screen). Indirectly, moving larger chunks of memory around is also much slower when you're doing animation or just drawing on the screen. Fortunately, today's accelerated graphics adapters are optimized for these types of operations and are shipping with larger amounts of onboard memory to accommodate the extra memory usage.

The 16- and 32-Bit Color Modes

For saving memory or improving performance, many display cards also support various other color modes. In the area of performance improvement, some cards support a 32-bit color mode sometimes called *true color mode*. Actually, the 32-bit color mode cannot display any more colors than the 24-bit mode, but it improves performance by aligning the data for each pixel on a 32-bit address boundary. Unfortunately, this results in a wasted 8 bits (1 byte) per pixel. On today's 32-bit Intel PCs, a memory address evenly divisible by 32 results in much faster memory access. Modern OpenGL accelerators also support 32-bit mode, with 24 bits being reserved for the RGB colors and 8 bits being used for destination alpha storage. You will learn more about the alpha channel in the next chapter.

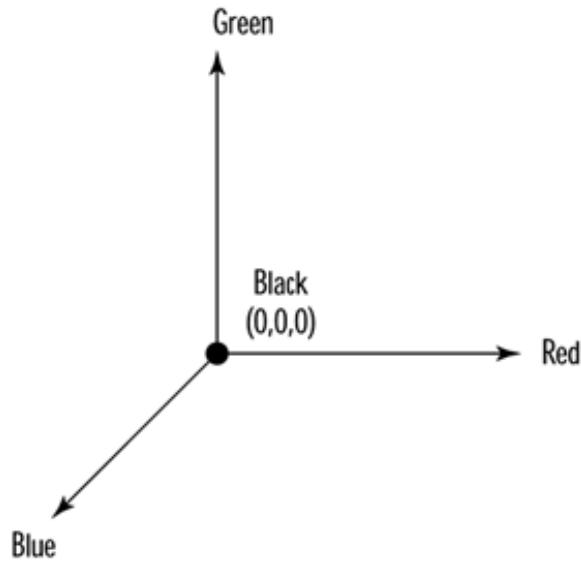
Another popular display mode, 16-bit color, is sometimes supported to use memory more efficiently. This allows one of 65,536 possible colors for each pixel. This display mode is practically as effective as 24-bit color for photographic image reproduction, as it can be difficult to tell the difference between 16-bit and 24-bit color modes for most photographic images. The savings in memory and increase in display speed have made this popular for the first generation of consumer "game" 3D accelerators. The added color fidelity of 24-bit mode, however, really adds to an image's quality, especially with shading and blending operations.

Using Color in OpenGL

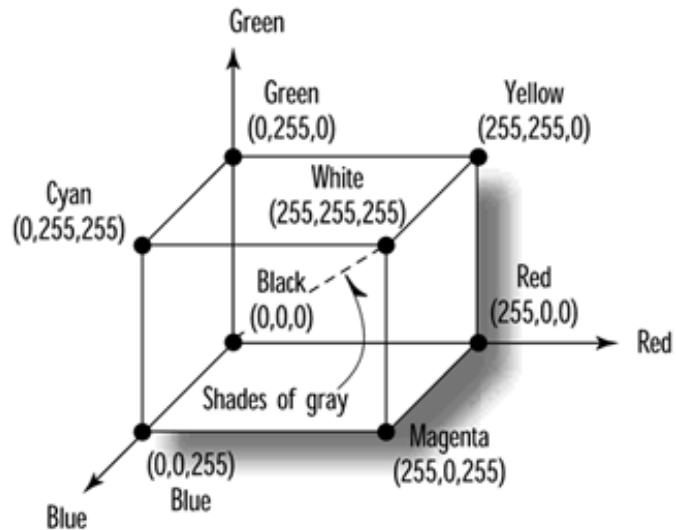
You now know that OpenGL specifies an exact color as separate intensities of red, green, and blue components. You also know that modern PC hardware might be able to display nearly all these combinations or only a very few. How, then, do we specify a desired color in terms of these red, green, and blue components?

The Color Cube

Because a color is specified by three positive color values, we can model the available colors as a volume that we call the *RGB colorspace*. [Figure 5.6](#) shows what this colorspace looks like at the origin with red, green, and blue as the axes. The red, green, and blue coordinates are specified just like x, y, and z coordinates. At the origin (0,0,0), the relative intensity of each component is zero, and the resulting color is black. The maximum available on the PC for storage information is 24 bits, so with 8 bits for each component, let's say that a value of 255 along the axis represents full saturation of that component. We then end up with a cube measuring 255 on each side. The corner directly opposite black, where the concentrations are (0,0,0), is white, with relative concentrations of (255,255,255). At full saturation (255) from the origin along each axis lies the pure colors of red, green, and blue.

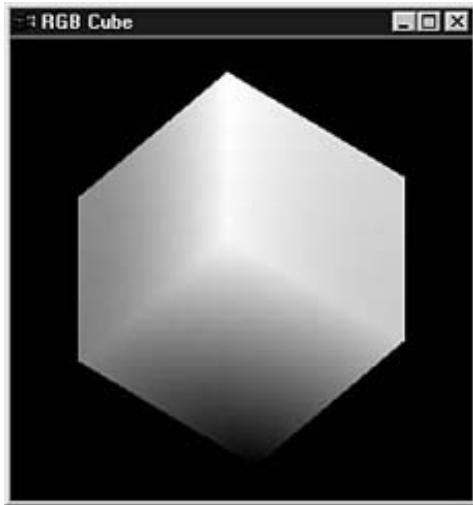
Figure 5.6. The origin of RGB colorspace.

This "color cube" (see [Figure 5.7](#)) contains all the possible colors, either on the surface of the cube or within the interior of the cube. For example, all possible shades of gray between black and white lie internally on the diagonal line between the corner at $(0,0,0)$ and $(255,255,255)$.

Figure 5.7. The RGB colorspace.

[Figure 5.8](#) shows the smoothly shaded color cube produced by a sample program from this chapter, CCUBE. The surface of this cube shows the color variations from black on one corner to white on the opposite corner. Red, green, and blue are present on their corners 255 units from black. Additionally, the colors yellow, cyan, and magenta have corners showing the combination of the other three primary colors. You can also spin the color cube around to examine all its sides by pressing the arrow keys.

Figure 5.8. Output from CCUBE is this color cube.



Setting the Drawing Color

Let's briefly review the `glColor` function. It is prototyped as follows:

```
void glColor<x><t>(red, green, blue, alpha);
```

In the function name, the `<x>` represents the number of arguments; it might be `3` for three arguments of red, green, and blue or `4` for four arguments to include the alpha component. The alpha component specifies the translucency of the color and is covered in more detail in the next chapter. For now, just use a three-argument version of the function.

The `<t>` in the function name specifies the argument's data type and can be `b`, `d`, `f`, `i`, `s`, `ub`, `ui`, or `us`, for byte, double, float, integer, short, unsigned byte, unsigned integer, and unsigned short data types, respectively. Another version of the function has a `v` appended to the end; this version takes an array that contains the arguments (the `v` stands for vectored). In the reference section, you will find an entry with more details on the `glColor` function.

Most OpenGL programs that you'll see use `glColor3f` and specify the intensity of each component as 0.0 for none or 1.0 for full intensity. However, it might be easier, if you have Windows programming experience, to use the `glColor3ub` version of the function. This version takes three unsigned bytes, from 0 to 255, to specify the intensities of red, green, and blue. Using this version of the function is like using the Windows RGB macro to specify a color:

```
glColor3ub(0,255,128) = RGB(0,255,128)
```

In fact, this approach might make it easier for you to match your OpenGL colors to existing RGB colors used by your program for other non-OpenGL drawing tasks. However, we should say that, internally, OpenGL represents color values as floating-point values, and you may incur some performance penalties due to the constant conversion to floats that must take place at runtime. It is also possible that in the future, higher resolution color buffers may evolve (in fact, floating-point color buffers are already starting to appear), and your color values specified as floats will be more faithfully represented by the color hardware.

Shading

Our previous working definition for `glColor` was that this function sets the current drawing color, and all objects drawn after this command have the last color specified. After discussing the OpenGL drawing primitives in the preceding chapter, we can now expand this definition as follows: The `glColor` function sets the current color that is used for all vertices drawn after the command. So far, all our examples have drawn wireframe objects or solid objects with each face a different

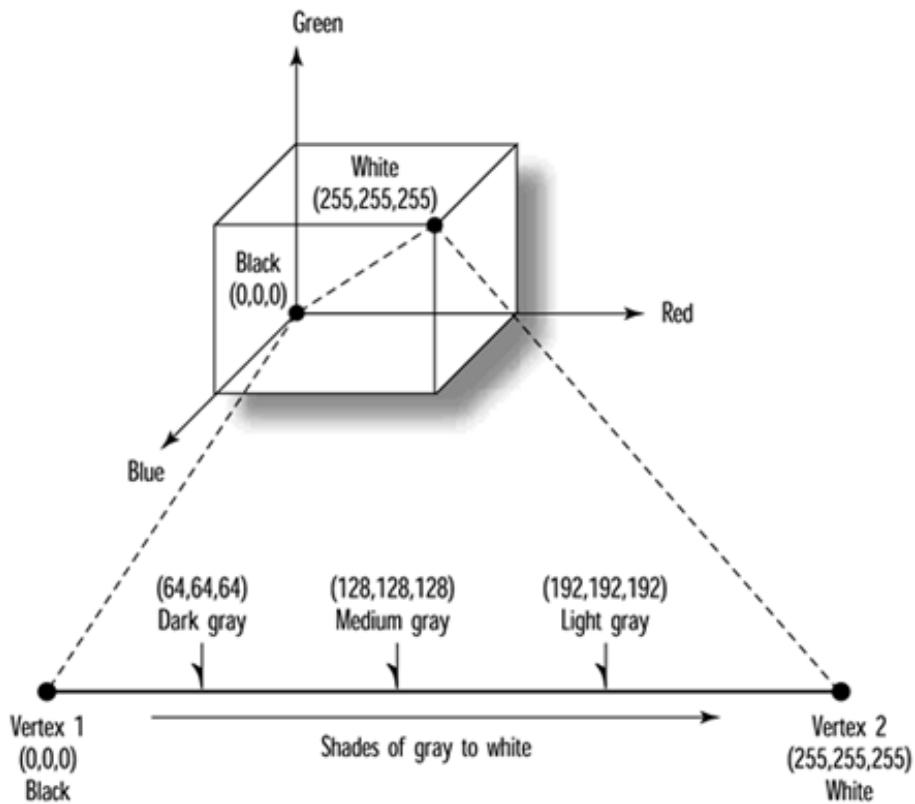
solid color. If we specify a different color for each vertex of a primitive (either point, line, or polygon), what color is the interior?

Let's answer this question first regarding points. A point has only one vertex, and whatever color you specify for that vertex is the resulting color for that point. Easy enough.

A line, however, has two vertices, and each can be set to a different color. The color of the line depends on the shading model. Shading is simply defined as the smooth transition from one color to the next. Any two points in the RGB colorspace (refer to [Figure 5.7](#)) can be connected by a straight line.

Smooth shading causes the colors along the line to vary as they do through the color cube from one color point to the other. [Figure 5.9](#) shows the color cube with the black and white corners identified. Below it is a line with two vertices, one black and one white. The colors selected along the length of the line match the colors along the straight line in the color cube, from the black to the white corners. This results in a line that progresses from black through lighter shades of gray and eventually to white.

Figure 5.9. How a line is shaded from black to white.



You can do shading mathematically by finding the equation of the line connecting two points in the three-dimensional RGB colorspace. Then you can simply loop through from one end of the line to the other, retrieving coordinates along the way to provide the color of each pixel on the screen. Many good books on computer graphics explain the algorithm to accomplish this effect, scale your color line to the physical line on the screen, and so on. Fortunately, OpenGL does all this work for you!

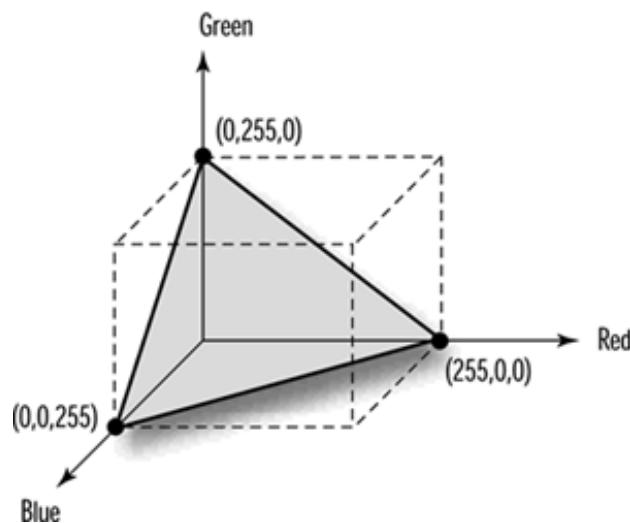
The shading exercise becomes slightly more complex for polygons. A triangle, for instance, can also be represented as a plane within the color cube. [Figure 5.10](#) shows a triangle with each vertex at full saturation for the red, green, and blue color components. The code to display this triangle is shown in [Listing 5.1](#) and in the sample program titled TRIANGLE on the CD that accompanies this book.

Listing 5.1. Drawing a Smooth-Shaded Triangle with Red, Green, and Blue Corners

```

// Enable smooth shading
glShadeModel(GL_SMOOTH);
// Draw the triangle
glBegin(GL_TRIANGLES);
    // Red Apex
    glColor3ub(GLubyte)255, (GLubyte)0, (GLubyte)0);
    glVertex3f(0.0f, 200.0f, 0.0f);
    // Green on the right bottom corner
    glColor3ub(GLubyte)0, (GLubyte)255, (GLubyte)0);
    glVertex3f(200.0f, -70.0f, 0.0f);
    // Blue on the left bottom corner
    glColor3ub(GLubyte)0, (GLubyte)0, (GLubyte)255);
    glVertex3f(-200.0f, -70.0f, 0.0f);
glEnd();

```

Figure 5.10. A triangle in RGB colorspace.**Setting the Shading Model**

The first line of [Listing 5.1](#) actually sets the shading model OpenGL uses to do smooth shading—the model we have been discussing. This is the default shading model, but it's a good idea to call this function anyway to ensure that your program is operating the way you intended.

The other shading model that can be specified with `glShadeModel` is `GL_FLAT` for flat shading. *Flat shading* means that no shading calculations are performed on the interior of primitives. Generally, with flat shading, the color of the primitive's interior is the color that was specified for the last vertex. The only exception is for a `GL_POLYGON` primitive, in which case the color is that of the first vertex.

Next, the code in [Listing 5.1](#) sets the top of the triangle to be pure red, the lower-right corner to be green, and the remaining bottom-left corner to be blue. Because smooth shading is specified, the interior of the triangle is shaded to provide a smooth transition between each corner.

The output from the `TRIANGLE` program is shown in [Figure 5.11](#). This output represents the plane shown graphically in [Figure 5.10](#).

Figure 5.11. Output from the TRIANGLE program.



Polygons, more complex than triangles, can also have different colors specified for each vertex. In these instances, the underlying logic for shading can become more intricate. Fortunately, you never have to worry about it with OpenGL. No matter how complex your polygon, OpenGL successfully shades the interior points between each vertex.

Color in the Real World

Real objects don't appear in a solid or shaded color based solely on their RGB values. [Figure 5.12](#) shows the output from the program titled JET from the CD. It's a simple jet airplane, hand plotted with triangles using only the methods covered so far in this book. As usual, jet and the other programs in this chapter allow you to spin the object around by using the arrow keys to better see the effects.

Figure 5.12. A simple jet built by setting a different color for each triangle.



The selection of colors is meant to highlight the three-dimensional structure of the jet. Aside from the crude assemblage of triangles, however, you can see that the jet looks hardly anything like a real object. Suppose you constructed a model of this airplane and painted each flat surface the colors represented. The model would still appear glossy or flat depending on the kind of paint used, and the color of each flat surface would vary with the angle of your view and any sources of light.

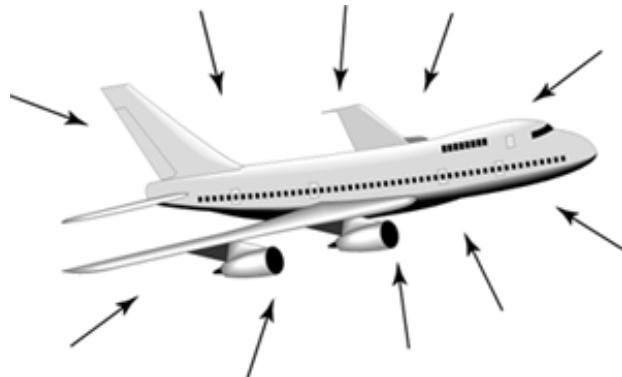
OpenGL does a reasonably good job of approximating the real world in terms of lighting conditions. Unless an object emits its own light, it is illuminated by three different kinds of light:

ambient, diffuse, and specular.

Ambient Light

Ambient light doesn't come from any particular direction. It has a source, but the rays of light have bounced around the room or scene and become directionless. Objects illuminated by ambient light are evenly lit on all surfaces in all directions. You can think of all previous examples in this book as being lit by a bright ambient light because the objects were always visible and evenly colored (or shaded) regardless of their rotation or viewing angle. [Figure 5.13](#) shows an object illuminated by ambient light.

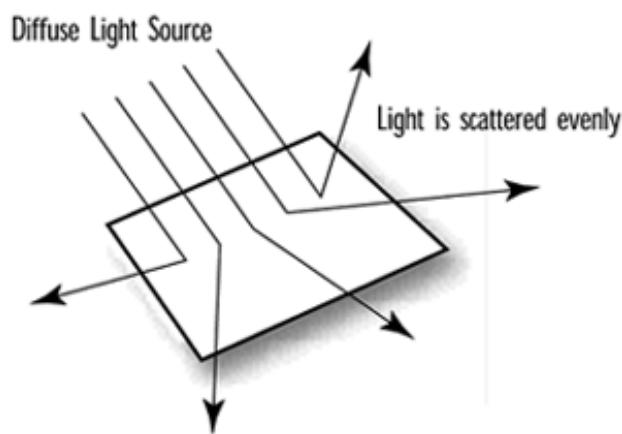
Figure 5.13. An object illuminated purely by ambient light.



Diffuse Light

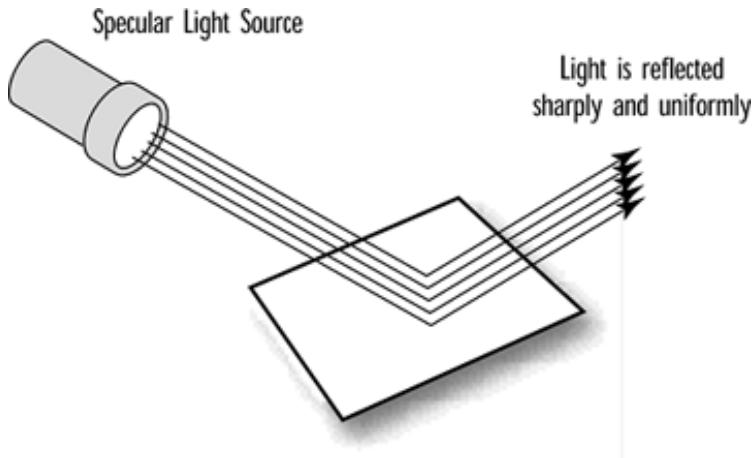
Diffuse light comes from a particular direction but is reflected evenly off a surface. Even though the light is reflected evenly, the object surface is brighter if the light is pointed directly at the surface than if the light grazes the surface from an angle. A good example of a diffuse light source is fluorescent lighting or sunlight streaming in a side window at noon. In [Figure 5.14](#), the object is illuminated by a diffuse light source.

Figure 5.14. An object illuminated by a purely diffuse light source.



Specular Light

Like light comes from a particular direction but is reflected evenly off diffuse light, specular light is directional, but it is reflected sharply and in a particular direction. A highly specular light tends to cause a bright spot on the surface it shines upon, which is called the *specular highlight*. A spotlight and the sun are examples of specular light. [Figure 5.15](#) shows an object illuminated by a purely specular light source.

Figure 5.15. An object illuminated by a purely specular light source.

Putting It All Together

No single light source is composed entirely of any of the three types of light just described. Rather, it is made up of varying intensities of each. For example, a red laser beam in a lab is composed of almost a pure-red specular component. However, smoke or dust particles scatter the beam, so it can be seen traveling across the room. This scattering represents the diffuse component of the light. If the beam is bright and no other light sources are present, you notice objects in the room taking on a red hue. This is a very small ambient component of that light.

Thus, a light source in a scene is said to be composed of three lighting components: ambient, diffuse, and specular. Just like the components of a color, each lighting component is defined with an RGBA value that describes the relative intensities of red, green, and blue light that make up that component. (For the purposes of light color, the alpha value is ignored.) For example, our red laser light might be described by the component values in [Table 5.1](#).

Table 5.1. Color and Light Distribution for a Red Laser Light Source

	Red	Green	Blue	Alpha
Specular	0.99	0.0	0.0	1.0
Diffuse	0.10	0.0	0.0	1.0
Ambient	0.05	0.0	0.0	1.0

Note that the red laser beam has no green or blue light. Also, note that specular, diffuse, and ambient light can each range in intensity from 0.0 to 1.0. You could interpret this table as saying that the red laser light in some scenes has a very high specular component, a small diffuse component, and a very small ambient component. Wherever it shines, you are probably going to see a reddish spot. Also, because of conditions (smoke, dust, and so on) in the room, the diffuse component allows you to see the beam traveling through the air. Finally, the ambient component—likely due to smoke or dust particles, as well—scatters a tiny bit of light all about the room. Ambient and diffuse components of light are frequently combined because they are so similar in nature.

Materials in the Real World

Light is only part of the equation. In the real world, objects do have a color of their own. Earlier in this chapter, we described the color of an object as defined by its reflected wavelengths of light. A blue ball reflects mostly blue photons and absorbs most others. This assumes that the light shining on the ball has blue photons in it to be reflected and detected by the observer. Generally, most scenes in the real world are illuminated by a white light containing an even mixture of all the colors. Under white light, therefore, most objects appear in their proper or "natural" colors. However, this is not always so; put the blue ball in a dark room with only a yellow light, and the ball appears black to the viewer because all the yellow light is absorbed and there is no blue to be reflected.

Material Properties

When we use lighting, we do not describe polygons as having a particular color, but rather as consisting of materials that have certain reflective properties. Instead of saying that a polygon is red, we say that the polygon is made of a material that reflects mostly red light. We are still saying that the surface is red, but now we must also specify the material's reflective properties for ambient, diffuse, and specular light sources. A material might be shiny and reflect specular light very well, while absorbing most of the ambient or diffuse light. Conversely, a flat colored object might absorb all specular light and not look shiny under any circumstances. Another property to be specified is the emission property for objects that emit their own light, such as taillights or glow-in-the-dark watches.

Adding Light to Materials

Setting lighting and material properties to achieve the desired effect takes some practice. There are no color cubes or rules of thumb to give you quick and easy answers. This is the point at which analysis gives way to art, and science yields to magic. When drawing an object, OpenGL decides which color to use for each pixel in the object. That object has reflective "colors," and the light source has "colors" of its own. How does OpenGL determine which colors to use? Understanding these principles is not difficult, but it does take some simple grade-school multiplication. (See, that teacher told you you'd need it one day!)

Each vertex of your primitives is assigned an RGB color value based on the net effect of the ambient, diffuse, and specular illumination multiplied by the ambient, diffuse, and specular reflectance of the material properties. Because you make use of smooth shading between the vertices, the illusion of illumination is achieved.

Calculating Ambient Light Effects

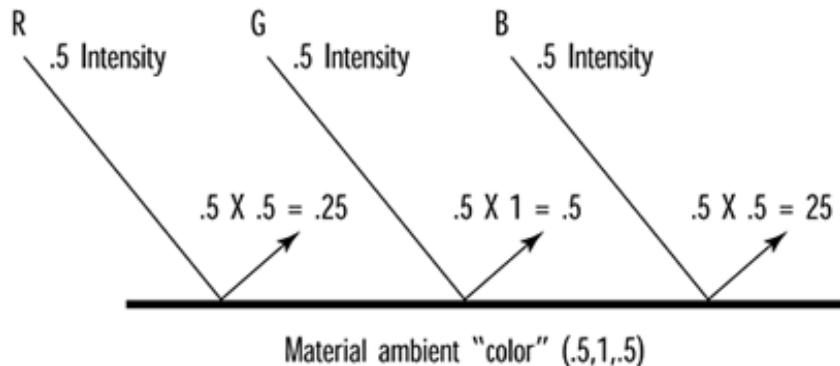
To calculate ambient light effects, you first need to put away the notion of color and instead think only in terms of red, green, and blue intensities. For an ambient light source of half-intensity red, green, and blue components, you have an RGB value for that source of (0.5, 0.5, 0.5). If this ambient light illuminates an object with ambient reflective properties specified in RGB terms of (0.5, 1.0, 0.5), the net "color" component from the ambient light is

$$(0.5 * 0.5, 0.5 * 1.0, 0.5 * 0.5) = (0.25, 0.5, 0.25)$$

This is the result of multiplying each of the ambient light source terms by each of the ambient material property terms (see [Figure 5.16](#)).

Figure 5.16. Calculating the ambient color component of an object.

Ambient Light Source



Thus, the material color components actually determine the percentage of incident light that is reflected. In our example, the ambient light had a red component that was at one-half intensity, and the material ambient property of 0.5 specified that one half of that one-half intensity light was reflected. Half of a half is a fourth, or 0.25.

Diffuse and Specular Effects

Calculating ambient light is as simple as it gets. Diffuse light also has RGB intensities that interact in the same way with material properties. However, diffuse light is directional, and the intensity at the surface of the object varies depending on the angle be

Adding Light to a Scene

This text might seem like a lot of theory to digest all of a sudden. Let's slow down and start exploring some examples of the OpenGL code needed for lighting; this exploration will also help reinforce what you've just learned. We demonstrate some additional features and requirements of lighting in OpenGL. The next few examples build on our JET program. The initial version contains no lighting code and just draws triangles with hidden surface elimination (depth testing) enabled. When we're done, the jet's metallic surface will glisten in the sunlight as you rotate it with the arrow keys.

Enabling the Lighting

To tell OpenGL to use lighting calculations, call `glEnable` with the `GL_LIGHTING` parameter:

```
glEnable(GL_LIGHTING);
```

This call alone tells OpenGL to use material properties and lighting parameters in determining the color for each vertex in your scene. However, without any specified material properties or lighting parameters, your object remains dark and unlit, as shown in [Figure 5.17](#). Look at the code for any of the JET-based sample programs, and you can see that we have called the function `SetupRC` right after creating the rendering context. This is the place where we do any initialization of lighting parameters.

Figure 5.17. An unlit jet reflects no light.



Setting Up the Lighting Model

After you enable lighting calculations, the first thing you should do is set up the lighting model. The three parameters that affect the lighting model are set with the `glLightModel` function.

The first lighting parameter used in our next example (the AMBIENT program) is `GL_LIGHT_MODEL_AMBIENT`. It lets you specify a global ambient light that illuminates all objects evenly from all sides. The following code specifies a bright white light:

```
// Bright white light - full intensity RGB values
GLfloat ambientLight[] = { 1.0f, 1.0f, 1.0f, 1.0f };
// Enable lighting
 glEnable(GL_LIGHTING);
// Set light model to use ambient light specified by ambientLight[]
 glLightModelfv(GL_LIGHT_MODEL_AMBIENT,ambientLight);
```

The variation of `glLightModel` shown here, `glLightModelfv`, takes as its first parameter the lighting model parameter being modified or set and then an array of the RGBA values that make up the light. The default RGBA values of this global ambient light are (0.2, 0.2, 0.2, 1.0), which is fairly dim. Other lighting model parameters allow you to determine whether the front, back, or both sides of polygons are illuminated and how the calculation of specular lighting angles is performed. See the reference section at the end of the chapter for more information on these parameters.

Setting Material Properties

Now that we have an ambient light source, we need to set some material properties so that our polygons reflect light and we can see our jet. There are two ways to set material properties. The first is to use the function `glMaterial` before specifying each polygon or set of polygons. Examine the following code fragment:

```
GLfloat gray[] = { 0.75f, 0.75f, 0.75f, 1.0f };
...
...
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, gray);
glBegin(GL_TRIANGLES);
    glVertex3f(-15.0f, 0.0f, 30.0f);
    glVertex3f(0.0f, 15.0f, 30.0f);
    glVertex3f(0.0f, 0.0f, -56.0f);
glEnd();
```

The first parameter to `glMaterialfv` specifies whether the front, back, or both (`GL_FRONT`,

`GL_BACK`, or `GL_FRONT_AND_BACK`) take on the material properties specified. The second parameter tells which properties are being set; in this instance, both the ambient and diffuse reflectances are set to the same values. The final parameter is an array containing the RGBA values that make up these properties. All primitives specified after the `glMaterial` call are affected by the last values set, until another call to `glMaterial` is made.

Under most circumstances, the ambient and diffuse components are the same, and unless you want specular highlights (sparkling, shiny spots), you don't need to define specular reflective properties. Even so, it would still be quite tedious if we had to define an array for every color in our object and call `glMaterial` before each polygon or group of polygons.

Now we are ready for the second and preferred way of setting material properties, called *color tracking*. With color tracking, you can tell OpenGL to set material properties by only calling `glColor`. To enable color tracking, call `glEnable` with the `GL_COLOR_MATERIAL` parameter:

```
glEnable(GL_COLOR_MATERIAL);
```

Then the function `glColorMaterial` specifies the material parameters that follow the values set by `glColor`. For example, to set the ambient and diffuse properties of the fronts of polygons to track the colors set by `glColor`, call

```
glColorMaterial(GL_FRONT,GL_AMBIENT_AND_DIFFUSE);
```

The earlier code fragment setting material properties would then be as follows. This approach looks like more code, but it actually saves many lines of code and executes faster as the number of different colored polygons grows:

```
// Enable color tracking
glEnable(GL_COLOR_MATERIAL);
// Front material ambient and diffuse colors track glColor
glColorMaterial(GL_FRONT,GL_AMBIENT_AND_DIFFUSE);
...
...
glcolor3f(0.75f, 0.75f, 0.75f);
glBegin(GL_TRIANGLES);
    glVertex3f(-15.0f,0.0f,30.0f);
    glVertex3f(0.0f, 15.0f, 30.0f);
    glVertex3f(0.0f, 0.0f, -56.0f);
glEnd();
```

Listing 5.2 contains the code we add with the `SetupRC` function to our jet example to set up a bright ambient light source and to set the material properties that allow the object to reflect light and be seen. We have also changed the colors of the jet so that each section is a different color rather than each polygon. The final output, shown in Figure 5.18, is not much different from the image before we had lighting. However, if we reduce the ambient light by half, we get the image shown in Figure 5.19. To reduce it by half, we set the ambient light RGBA values to the following:

```
GLfloat ambientLight[] = { 0.5f, 0.5f, 0.5f, 1.0f };
```

Figure 5.18. Output from completed AMBIENT sample program.

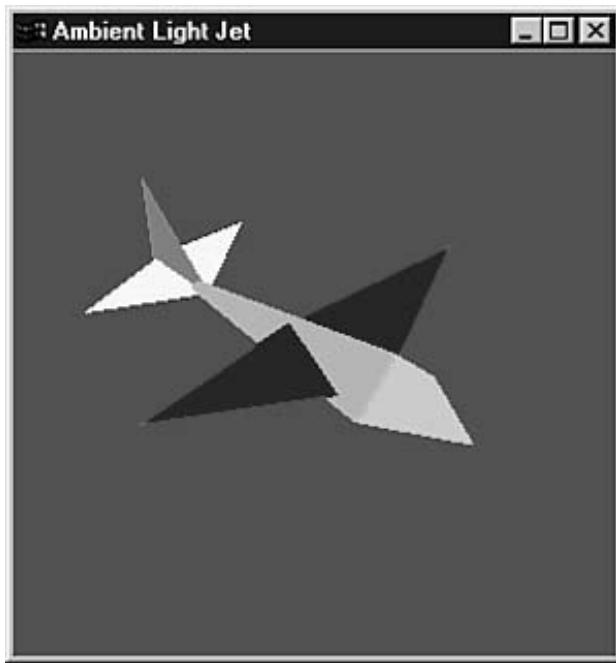
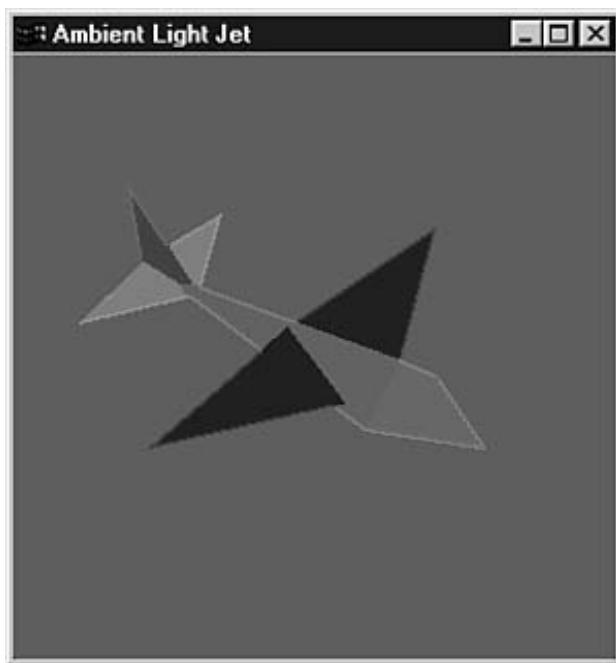


Figure 5.19. Output from the AMBIENT program when the light source is cut in half.



You can see how we might reduce the ambient light in a scene to produce a dimmer image. This capability is useful for simulations in which dusk approaches gradually or when a more direct light source is blocked, as when an object is in the shadow of another, larger object.

Listing 5.2. Setup for Ambient Lighting Conditions

```
// This function does any needed initialization on the rendering
// context. Here it sets up and initializes the lighting for
// the scene.
void SetupRC()
{
    // Light values
    // Bright white light
    GLfloat ambientLight[] = { 1.0f, 1.0f, 1.0f, 1.0f };
```

```

glEnable(GL_DEPTH_TEST);           // Hidden surface removal
glEnable(GL_CULL_FACE);          // Do not calculate inside of jet
glFrontFace(GL_CCW);             // Counterclockwise polygons face out
// Lighting stuff
glEnable(GL_LIGHTING);           // Enable lighting
// Set light model to use ambient light specified by ambientLight[]
glLightModelfv(GL_LIGHT_MODEL_AMBIENT,ambientLight);
glEnable(GL_COLOR_MATERIAL);      // Enable material color tracking
// Front material ambient and diffuse colors track glColor
glColorMaterial(GL_FRONT,GL_AMBIENT_AND_DIFFUSE);
// Nice light blue background
glClearColor(0.0f, 0.0f, 05.f,1.0f);
}

```

Using a Light Source

Manipulating the ambient light has its uses, but for most applications attempting to model the real world, you must specify one or more specific sources of light. In addition to their intensities and colors, these sources have a location and/or a direction. The placement of these lights can dramatically affect the appearance of your scene.

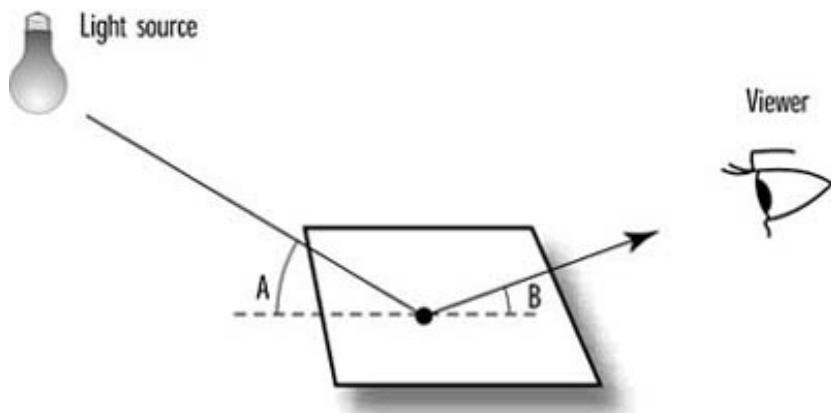
OpenGL supports at least eight independent light sources located anywhere in your scene or out of the viewing volume. You can locate a light source an infinite distance away and make its light rays parallel or make it a nearby light source radiating outward. You can also specify a spotlight with a specific cone of light radiating from it, as well as manipulate its characteristics.

Which Way Is Up?

When you specify a light source, you tell OpenGL where it is and in which direction it's shining. Often, the light source shines in all directions, but it can be directional. Either way, for any object you draw, the rays of light from any source (other than a pure ambient source) strike the surface of the polygons that make up the object at an angle. Of course, in the case of a directional light, the surfaces of all polygons might not necessarily be illuminated. To calculate the shading effects across the surface of the polygons, OpenGL must be able to calculate the angle.

In [Figure 5.20](#), a polygon (a square) is being struck by a ray of light from some source. The ray makes an angle (A) with the plane as it strikes the surface. The light is then reflected at an angle (B) toward the viewer (or you wouldn't see it). These angles are used in conjunction with the lighting and material properties we have discussed thus far to calculate the apparent color of that location. It happens by design that the locations used by OpenGL are the vertices of the polygon. Because OpenGL calculates the apparent colors for each vertex and then does smooth shading between them, the illusion of lighting is created. Magic!

Figure 5.20. Light is reflected off objects at specific angles.

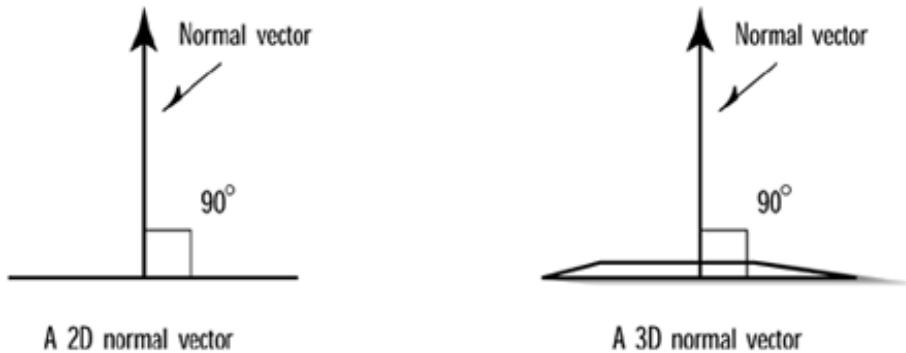


From a programming standpoint, these lighting calculations present a slight conceptual difficulty. Each polygon is created as a set of vertices, which are nothing more than points. Each vertex is then struck by a ray of light at some angle. How then do you (or OpenGL) calculate the angle between a point and a line (the ray of light)? Of course, you can't geometrically find the angle between a single point and a line in 3D space because there are an infinite number of possibilities. Therefore, you must associate with each vertex some piece of information that denotes a direction upward from the vertex and away from the surface of the primitive.

Surface Normals

A line from the vertex in the upward direction starts in some imaginary plane (or your polygon) at a right angle. This line is called a normal vector. The term *normal vector* might sound like something the Star Trek crew members toss around, but it just means a line perpendicular to a real or imaginary surface. A vector is a line pointed in some direction, and the word *normal* is just another way for eggheads to say perpendicular (intersecting at a 90° angle). As if the word *perpendicular* weren't bad enough! Therefore, a normal vector is a line pointed in a direction that is at a 90° angle to the surface of your polygon. [Figure 5.21](#) presents examples of 2D and 3D normal vectors.

Figure 5.21. A 2D and a 3D normal vector.

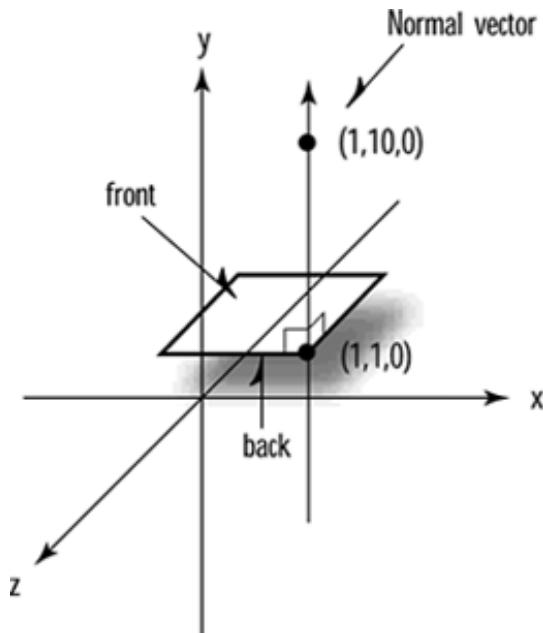


You might already be asking why we must specify a normal vector for each vertex. Why can't we just specify a single normal for a polygon and use it for each vertex? We can—and for our first few examples, we do. However, sometimes you don't want each normal to be exactly perpendicular to the surface of the polygon. You may have noticed that many surfaces are not flat! You can approximate these surfaces with flat, polygonal sections, but you end up with a jagged or multifaceted surface. Later, we discuss a technique to produce the illusion of smooth curves with flat polygons by "tweaking" surface normals (more magic!). But first things first...

Specifying a Normal

To see how we specify a normal for a vertex, let's look at [Figure 5.22](#)—a plane floating above the xz plane in 3D space. We've made this illustration simple to demonstrate the concept. Notice the line through the vertex $(1, 1, 0)$ that is perpendicular to the plane. If we select any point on this line, say $(1, 10, 0)$, the line from the first point $(1, 1, 0)$ to the second point $(1, 10, 0)$ is our normal vector. The second point specified actually indicates that the direction from the vertex is up in the y direction. This convention is also used to indicate the front and back sides of polygons, as the vector travels up and away from the front surface.

Figure 5.22. A normal vector traveling perpendicular from the surface.

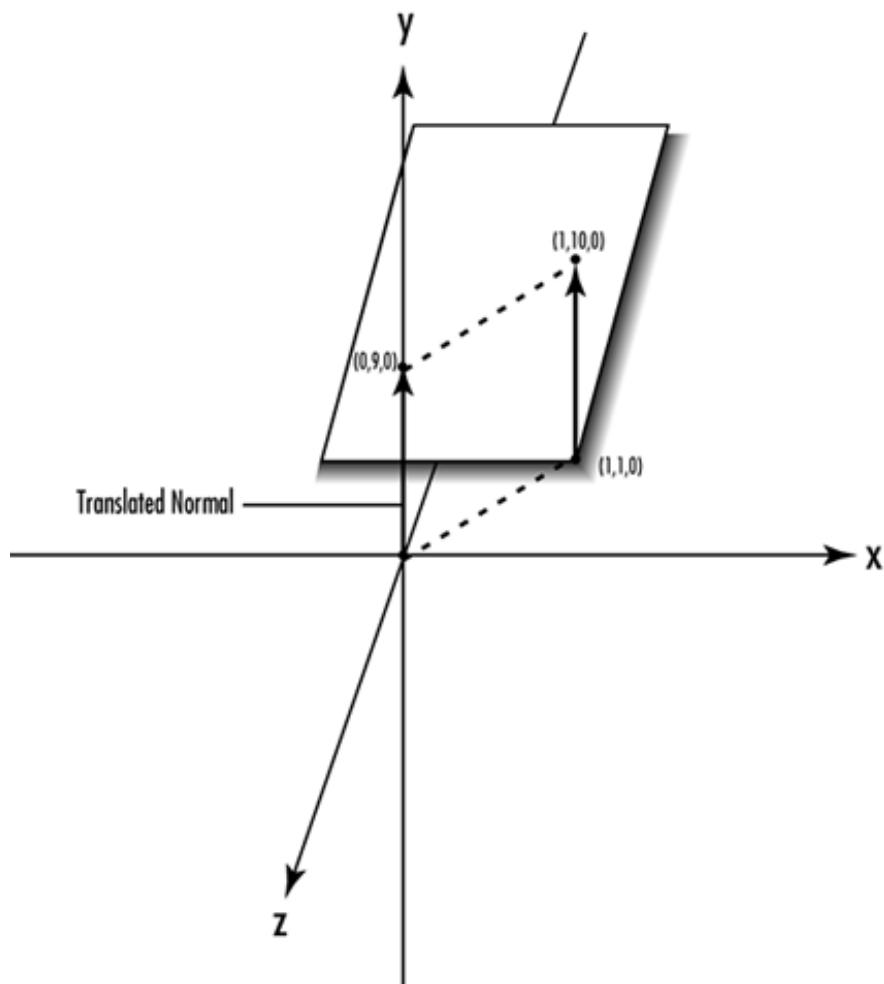


You can see that this second point is the number of units in the x, y, and z directions for some point on the normal vector away from the vertex. Rather than specify two points for each normal vector, we can subtract the vertex from the second point on the normal, yielding a single coordinate triplet that indicates the x, y, and z steps away from the vertex. For our example, this is

$$(1, 10, 0) - (1, 1, 0) = (1 - 1, 10 - 1, 0) = (0, 9, 0)$$

Here's another way of looking at this example: If the vertex were translated to the origin, the point specified by subtracting the two original points would still specify the direction pointing away and at a 90° angle from the surface. [Figure 5.23](#) shows the newly translated normal vector.

Figure 5.23. The newly translated normal vector.



The vector is a directional quantity that tells OpenGL which direction the vertices (or polygon) face. This next code segment shows a normal vector being specified for one of the triangles in the JET sample program:

```
glBegin(GL_TRIANGLES);
  glNormal3f(0.0f, -1.0f, 0.0f);
  glVertex3f(0.0f, 0.0f, 60.0f);
  glVertex3f(-15.0f, 0.0f, 30.0f);
  glVertex3f(15.0f, 0.0f, 30.0f);
glEnd();
```

The function `glNormal3f` takes the coordinate triplet that specifies a normal vector pointing in the direction perpendicular to the surface of this triangle. In this example, the normals for all three vertices have the same direction, which is down the negative y-axis. This is a simple example because the triangle is lying flat in the xz plane, and it actually represents a bottom section of the jet. You'll see later that often we want to specify a different normal for each vertex.

The prospect of specifying a normal for every vertex or polygon in your drawing might seem daunting, especially because few surfaces lie cleanly in one of the major planes. Never fear! We shortly present a reusable function that you can call again and again to calculate your normals for you.

POLYGON WINDING

Take special note of the order of the vertices in the jet's triangle. If you view this triangle being drawn from the direction in which the normal vector points, the corners appear counterclockwise around the triangle. This is called *polygon winding*. By default, the front of a polygon is defined as the side from which the vertices appear to be wound in a counterclockwise fashion.

Unit Normals

As OpenGL does its magic, all surface normals must eventually be converted to unit normals. A unit normal is just a normal vector that has a length of 1. The normal in [Figure 5.23](#) has a length of 9. You can find the length of any normal by squaring each component, adding them together, and taking the square root. Divide each component of the normal by the length, and you get a vector pointed in exactly the same direction, but only 1 unit long. In this case, our new normal vector is specified as (0, 1, 0). This is called *normalization*. Thus, for lighting calculations, all normal vectors must be normalized. Talk about jargon!

You can tell OpenGL to convert your normals to unit normals automatically, by enabling normalization with `glEnable` and a parameter of `GL_NORMALIZE`:

```
glEnable(GL_NORMALIZE);
```

This approach does, however, have performance penalties. It's far better to calculate your normals ahead of time as unit normals instead of relying on OpenGL to perform this task for you.

You should note that calls to the `glScale` transformation function also scale the length of your normals. If you use `glScale` and lighting, you can obtain undesired results from your OpenGL lighting. If you have specified unit normals for all your geometry and used a constant scaling factor with `glScale` (all geometry is scaled by the same amount), a new alternative to `GL_NORMALIZE` (new to OpenGL 1.2) is `GL_RESCALE_NORMALS`. You enable this parameter with a call such as

```
glEnable(GL_RESCALE_NORMALS);
```

This call tells OpenGL that your normals are not unit length, but they can all be scaled by the same amount to make them unit length. OpenGL figures this out by examining the modelview matrix. The result is fewer mathematical operations per vertex than are otherwise required.

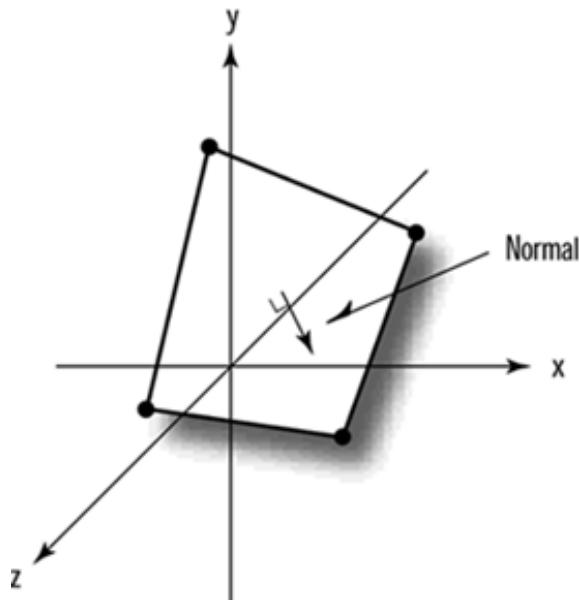
Because it is better to give OpenGL unit normals to begin with, the `glTools` library comes with a function that will take any normal vector and "normalize" it for you:

```
void gltNormalizeVector(GLTVector vNormal);
```

Finding a Normal

[Figure 5.24](#) presents another polygon that is not simply lying in one of the axis planes. The normal vector pointing away from this surface is more difficult to guess, so we need an easy way to calculate the normal for any arbitrary polygon in 3D coordinates.

Figure 5.24. A nontrivial normal problem.



You can easily calculate the normal vector for any polygon by taking three points that lie in the plane of that polygon. [Figure 5.25](#) shows three points—P1, P2, and P3—that you can use to define two vectors: vector V1 from P1 to P2, and vector V2 from P1 to P3. Mathematically, two vectors in three-dimensional space define a plane. (Your original polygon lies in this plane.) If you take the cross product of those two vectors (written mathematically as $V1 \times V2$), the resulting vector is perpendicular to that plane. [Figure 5.26](#) shows the vector V3 derived by taking the cross product of V1 and V2.

Figure 5.25. Two vectors defined by three points on a plane.

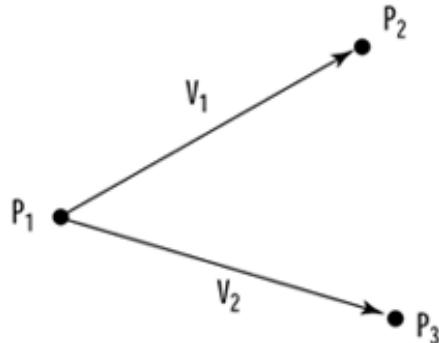
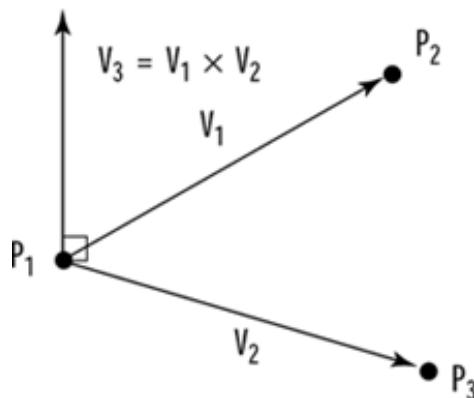


Figure 5.26. A normal vector as the cross product of two vectors.



Again, because this is such a useful and often-used method, the `glTools` library contains a

function that calculates a normal vector based on three points on a polygon:

```
void gltGetNormalVector(GLTVector vP1, GLTVector vP2,
                        GLTVector vP3, GLTVector vNormal);
```

To use this function, pass it three vectors (each just an array of three floats) from your polygon or triangle (specified in counterclockwise winding order) and another vector array that will contain the normal vector on return.

Setting Up a Source

Now that you understand the requirements of setting up your polygons to receive and interact with a light source, it's time to turn on the lights! [Listing 5.3](#) shows the `SetupRC` function from the sample program LITJET. Part of the setup process for this sample program creates a light source and places it to the upper left, slightly behind the viewer. The light source `GL_LIGHT0` has its ambient and diffuse components set to the intensities specified by the arrays `ambientLight[]` and `diffuseLight[]`. This results in a moderate white light source:

```
GLfloat ambientLight[] = { 0.3f, 0.3f, 0.3f, 1.0f };
GLfloat diffuseLight[] = { 0.7f, 0.7f, 0.7f, 1.0f };
...
...
// Set up and enable light 0
glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
```

Finally, the light source `GL_LIGHT0` is enabled:

```
 glEnable(GL_LIGHT0);
```

The light is positioned by this code, located in the `ChangeSize` function:

```
GLfloat lightPos[] = { -50.0f, 50.0f, 100.0f, 1.0f };
...
...
glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
```

Here, `lightPos[]` contains the position of the light. The last value in this array is 1.0, which specifies that the designated coordinates are the position of the light source. If the last value in the array is 0.0, it indicates that the light is an infinite distance away along the vector specified by this array. We touch more on this issue later. Lights are like geometric objects in that they can be moved around by the modelview matrix. By placing the light's position when the viewing transformation is performed, we ensure the light is in the proper location regardless of how we transform the geometry.

Listing 5.3. Light and Rendering Context Setup for LITJET

```
// This function does any needed initialization on the rendering
// context. Here it sets up and initializes the lighting for
// the scene.
void SetupRC()
{
    // Light values and coordinates
    GLfloat ambientLight[] = { 0.3f, 0.3f, 0.3f, 1.0f };
    GLfloat diffuseLight[] = { 0.7f, 0.7f, 0.7f, 1.0f };
    glEnable(GL_DEPTH_TEST);      // Hidden surface removal
    glFrontFace(GL_CCW);        // Counterclockwise polygons face out
    glEnable(GL_CULL_FACE);      // Do not calculate inside of jet
```

```

// Enable lighting
glEnable(GL_LIGHTING);
// Set up and enable light 0
glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
glEnable(GL_LIGHT0);
// Enable color tracking
glEnable(GL_COLOR_MATERIAL);
// Set material properties to follow glColor values
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
// Light blue background
glClearColor(0.0f, 0.0f, 1.0f, 1.0f );
}

```

Setting the Material Properties

Notice in [Listing 5.3](#) that color tracking is enabled, and the properties to be tracked are the ambient and diffuse reflective properties for the front surface of the polygons. This is just as it was defined in the AMBIENT sample program:

```

// Enable color tracking
glEnable(GL_COLOR_MATERIAL);
// Set material properties to follow glColor values
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

```

Specifying the Polygons

The rendering code from the first two JET samples changes considerably now to support the new lighting model. [Listing 5.4](#) is an excerpt taken from the `RenderScene` function from LITJET.

Listing 5.4. Code Sample That Sets Color and Calculates and Specifies Normals and Polygons

```

GLTVector vNormal;      // Storage for calculated surface normal
...
...
// Set material color
glColor3ub(128, 128, 128);
glBegin(GL_TRIANGLES);
    glNormal3f(0.0f, -1.0f, 0.0f);
    glNormal3f(0.0f, -1.0f, 0.0f);
    glVertex3f(0.0f, 0.0f, 60.0f);
    glVertex3f(-15.0f, 0.0f, 30.0f);
    glVertex3f(15.0f, 0.0f, 30.0f);
    // Vertices for this panel
    {
        GLTVector vPoints[3] = {{ 15.0f, 0.0f, 30.0f},
                               { 0.0f, 15.0f, 30.0f},
                               { 0.0f, 0.0f, 60.0f}};
        // Calculate the normal for the plane
        gltGetNormalVector(vPoints[0], vPoints[1], vPoints[2], vNormal);
        glNormal3fv(vNormal);
        glVertex3fv(vPoints[0]);
        glVertex3fv(vPoints[1]);
        glVertex3fv(vPoints[2]);
    }
    {
        GLTVector vPoints[3] = {{ 0.0f, 0.0f, 60.0f },
                               { 0.0f, 15.0f, 30.0f },

```

```

    { -15.0f, 0.0f, 30.0f }};
gltGetNormalVector(vPoints[0], vPoints[1], vPoints[2], vNormal);
glNormal3fv(vNormal);
glVertex3fv(vPoints[0]);
glVertex3fv(vPoints[1]);
glVertex3fv(vPoints[2]);
}

```

Notice that we are calculating the normal vector using the `gltGetNormalVector` function from `glTools`. Also, the material properties are now following the colors set by `glColor`. One other thing you notice is that not every triangle is blocked by `glBegin/glEnd` functions. You can specify once that you are drawing triangles, and every three vertices are used for a new triangle until you specify otherwise with `glEnd`. For very large numbers of polygons, this technique can considerably boost performance by eliminating many unnecessary function calls and primitive batch setup.

[Figure 5.27](#) shows the output from the completed LITJET sample program. The jet is now a single shade of gray instead of multiple colors. We changed the color to make it easier to see the lighting effects on the surface. Even though the plane is one solid "color," you can still see the shape due to the lighting. By rotating the jet around with the arrow keys, you can see the dramatic shading effects as the surface of the jet moves and interacts with the light.

Figure 5.27. Output from the LITJET program.



TIP

The most obvious way to improve the performance of this code is to calculate all the normal vectors ahead of time and store them for use in the `RenderScene` function.

Before you pursue this, read [Chapter 11](#), "It's All About the Pipeline: Faster Geometry Throughput," for the material on display lists and vertex arrays. Display lists and vertex arrays provide a means of storing calculated values not only for the normal vectors, but for the polygon data as well. Remember, these examples are meant to demonstrate the concepts. They are not necessarily the most efficient code possible.

Lighting Effects

The ambient and diffuse lights from the LITJET example are sufficient to provide the illusion of lighting. The surface of the jet appears shaded according to the angle of the incident light. As the jet rotates, these angles change and you can see the lighting effects changing in such a way that you can easily guess where the light is coming from.

We ignored the specular component of the light source, however, as well as the specular reflectivity of the material properties on the jet. Although the lighting effects are pronounced, the surface of the jet is rather flatly colored. Ambient and diffuse lighting and material properties are all you need if you are modeling clay, wood, cardboard, cloth, or some other flatly colored object. But for metallic surfaces such as the skin of an airplane, some shine is often desirable.

Specular Highlights

Specular lighting and material properties add needed gloss to the surface of your objects. This shininess has a whitening effect on an object's color and can produce specular highlights when the angle of incident light is sharp in relation to the viewer. A specular highlight is what occurs when nearly all the light striking the surface of an object is reflected away. The white sparkle on a shiny red ball in the sunlight is a good example of a specular highlight.

Specular Light

You can easily add a specular component to a light source. The following code shows the light source setup for the LITJET program, modified to add a specular component to the light:

```
// Light values and coordinates
GLfloat ambientLight[] = { 0.3f, 0.3f, 0.3f, 1.0f };
GLfloat diffuseLight[] = { 0.7f, 0.7f, 0.7f, 1.0f };
GLfloat specular[] = { 1.0f, 1.0f, 1.0f, 1.0f };
...
...
// Enable lighting
 glEnable(GL_LIGHTING);
// Set up and enable light 0
 glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight);
 glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
 glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
 glEnable(GL_LIGHT0);
```

The `specular[]` array specifies a very bright white light source for the specular component of the light. Our purpose here is to model bright sunlight. The following line simply adds this specular component to the light source `GL_LIGHT0`:

```
glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
```

If this were the only change you made to LITJET, you wouldn't see any difference in the jet's appearance. We haven't yet defined any specular reflectance properties for the material properties.

Specular Reflectance

Adding specular reflectance to material properties is just as easy as adding the specular component to the light source. This next code segment shows the code from LITJET, again modified to add specular reflectance to the material properties:

```
// Light values and coordinates
GLfloat specref[] = { 1.0f, 1.0f, 1.0f, 1.0f };
...
...
```

```
...
// Enable color tracking
glEnable(GL_COLOR_MATERIAL);
// Set material properties to follow glColor values
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
// All materials hereafter have full specular reflectivity
// with a high shine
glMaterialfv(GL_FRONT, GL_SPECULAR,specref);
glMateriali(GL_FRONT,GL_SHININESS,128);
```

As before, we enable color tracking so that the ambient and diffuse reflectance of the materials follow the current color set by the `glColor` functions. (Of course, we don't want the specular reflectance to track `glColor` because we are specifying it separately and it doesn't change.)

Now, we've added the array `specref[]`, which contains the RGBA values for our specular reflectance. This array of all 1s produces a surface that reflects nearly all incident specular light. The following line sets the material properties for all subsequent polygons to have this reflectance:

```
glMaterialfv(GL_FRONT, GL_SPECULAR,specref);
```

Because we do not call `glMaterial` again with the `GL_SPECULAR` property, all materials have this property. We set up the example this way on purpose because we want the entire jet to appear made of metal or very shiny composites.

What we have done here in our setup routine is important: We have specified that the ambient and diffuse reflective material properties of all future polygons (until we say otherwise with another call to `glMaterial` or `glColorMaterial`) change as the current color changes, but that the specular reflective properties remain the same.

Specular Exponent

As stated earlier, high specular light and reflectivity brighten the colors of the object. For this example, the present extremely high specular light (full intensity) and specular reflectivity (full reflectivity) result in a jet that appears almost totally white or gray except where the surface points away from the light source (in which case, it is black and unlit). To temper this effect, we use the next line of code after the specular component is specified:

```
glMateriali(GL_FRONT,GL_SHININESS,128);
```

The `GL_SHININESS` property sets the specular exponent of the material, which specifies how small and focused the specular highlight is. A value of 0 specifies an unfocused specular highlight, which is actually what is producing the brightening of the colors evenly across the entire polygon. If you set this value, you reduce the size and increase the focus of the specular highlight, causing a shiny spot to appear. The larger the value, the more shiny and pronounced the surface. The range of this parameter is 1–128 for all implementations of OpenGL.

[Listing 5.5](#) shows the new `SetupRC` code in the sample program SHINYJET. This is the only code that has changed from LITJET (other than the title of the window) to produce a very shiny and glistening jet. [Figure 5.28](#) shows the output from this program, but to fully appreciate the effect, you should run the program and hold down one of the arrow keys to spin the jet about in the sunlight.

Listing 5.5. Setup from SHINYJET to Produce Specular Highlights on the Jet

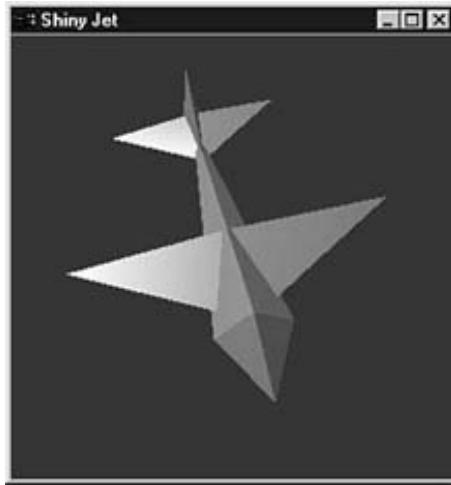
```
// This function does any needed initialization on the rendering
// context. Here it sets up and initializes the lighting for
// the scene.
void SetupRC()
```

```

{
// Light values and coordinates
GLfloat ambientLight[] = { 0.3f, 0.3f, 0.3f, 1.0f };
GLfloat diffuseLight[] = { 0.7f, 0.7f, 0.7f, 1.0f };
GLfloat specular[] = { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat specref[] = { 1.0f, 1.0f, 1.0f, 1.0f };
glEnable(GL_DEPTH_TEST); // Hidden surface removal
glFrontFace(GL_CCW); // Counterclockwise polygons face out
glEnable(GL_CULL_FACE); // Do not calculate inside of jet
// Enable lighting
glEnable(GL_LIGHTING);
// Set up and enable light 0
glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
glLightfv(GL_LIGHT0,GL_SPECULAR,specular);
glEnable(GL_LIGHT0);
// Enable color tracking
glEnable(GL_COLOR_MATERIAL);
// Set material properties to follow glColor values
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
// All materials hereafter have full specular reflectivity
// with a high shine
glMaterialfv(GL_FRONT, GL_SPECULAR,specref);
glMateriali(GL_FRONT,GL_SHININESS,128);
// Light blue background
glClearColor(0.0f, 0.0f, 1.0f, 1.0f );
}

```

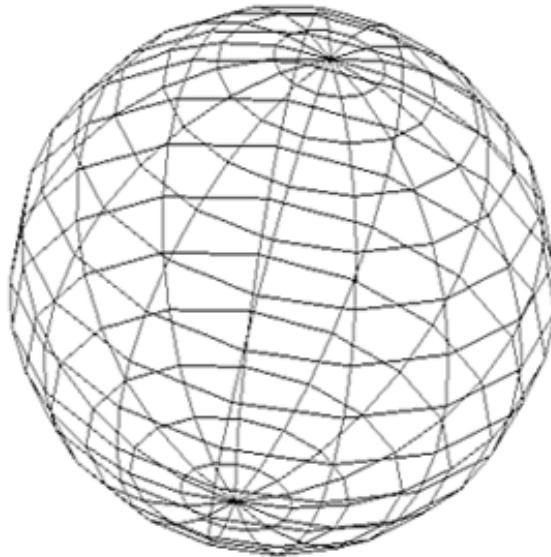
Figure 5.28. Output from the SHINYJET program.



Normal Averaging

Earlier, we mentioned that by "tweaking" your normals, you can produce apparently smooth surfaces with flat polygons. This technique, known as *normal averaging*, produces some interesting optical illusions. Say you have a sphere made up of quads and triangles like the one shown in [Figure 5.29](#).

Figure 5.29. A typical sphere made up of quads and triangles.



If each face of the sphere had a single normal specified, the sphere would look like a large faceted jewel. If you specify the "true" normal for each vertex, however, the lighting calculations at each vertex produce values that OpenGL smoothly interpolates across the face of the polygon. Thus, the flat polygons are shaded as if they were a smooth surface.

What do we mean by "true" normal? The polygonal representation is only an approximation of the true surface. Theoretically, if we used enough polygons, the surface would appear smooth. This is similar to the idea we used in [Chapter 3](#), "Drawing in Space: Geometric Primitives and Buffers," to draw a smooth curve with a series of short line segments. If we consider each vertex to be a point on the true surface, the actual normal value for that surface is the true normal for the surface.

For our case of the sphere, the normal would point directly out from the center of the sphere through each vertex. We show this graphically for a simple 2D case in [Figures 5.30](#) and [5.31](#). In [Figure 5.30](#), each flat segment has a normal pointing perpendicular to its surface. We did this just like we did for our LITJET example previously. [Figure 5.31](#), however, shows how each normal is not perpendicular to the line segment but is perpendicular to the surface of the sphere, or the *tangent line* to the surface.

Figure 5.30. An approximation with normals perpendicular to each face.

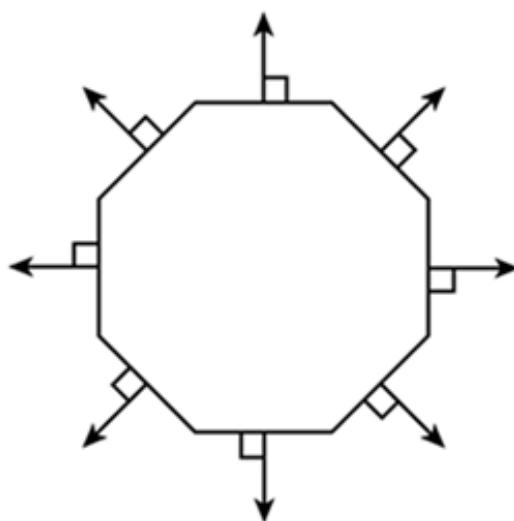
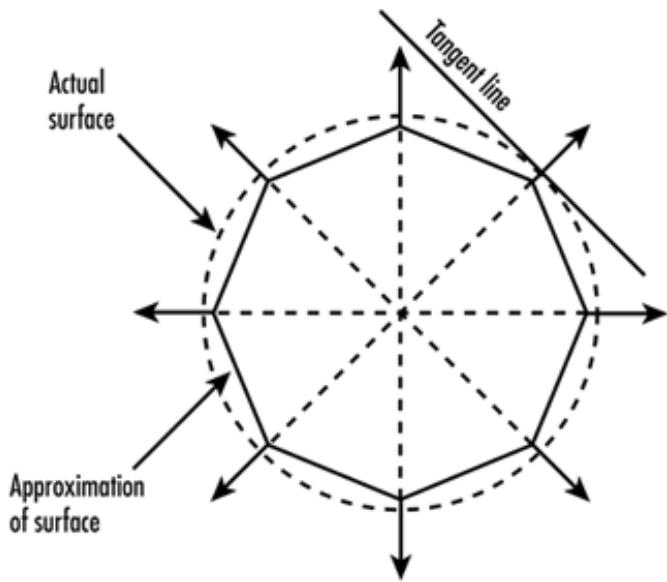


Figure 5.31. Each normal is perpendicular to the surface itself.



The tangent line touches the curve in one place and does not penetrate it. The 3D equivalent is a tangent plane. In [Figure 5.31](#), you can see the outline of the actual surface and that the normal is actually perpendicular to the line tangent to the surface.

For a sphere, calculation of the normal is reasonably simple. (The normal actually has the same values as the vertex relative to the center!) For other nontrivial surfaces, the calculation might not be so easy. In such cases, you calculate the normals for each polygon that shared a vertex. The actual normal you assign to that vertex is the average of these normals. The visual effect is a nice, smooth, regular surface, even though it is actually composed of numerous small, flat segments.

Putting It All Together

Now it's time for a more complex sample program. We demonstrate how to use normals to create a smooth surface appearance, move a light around in a scene, create a spot light, and finally, identify one of the drawbacks of OpenGL lighting.

Our next sample program, SPOT, performs all these tasks. Here, we create a solid sphere in the center of our viewing volume with `glutSolidSphere`. We shine a spotlight on this sphere that we can move around, and we change the "smoothness" of the normals and demonstrate some of the limitations of OpenGL lighting.

So far, we have been specifying a light's position with `glLight` as follows:

```
// Array to specify position
GLfloat lightPos[] = { 0.0f, 150.0f, 150.0f, 1.0f };
...
...
// Set the light position
glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
```

The array `lightPos[]` contains the x, y, and z values that specify either the light's actual position in the scene or the direction from which the light is coming. The last value, 1.0 in this case, indicates that the light is actually present at this location. By default, the light radiates equally in all directions from this location, but you can change this default to make a spotlight effect.

To make a light source an infinite distance away and coming from the direction specified by this vector, you place 0.0 in this last `lightPos[]` array element. A directional light source, as this is called, strikes the surface of your objects evenly. That is, all the light rays are parallel. In a positional light source, on the other hand, the light rays diverge from the light source.

Creating a Spotlight

Creating a spotlight is no different from creating any other positional light source. The code in [Listing 5.6](#) shows the `SetupRC` function from the SPOT sample program. This program places a blue sphere in the center of the window. It also creates a spotlight that you can move vertically with the up- and down-arrow keys and horizontally with the left- and right-arrow keys. As the spotlight moves over the surface of the sphere, a specular highlight follows it on the surface.

Listing 5.6. Lighting Setup for the SPOT Sample Program

```
// Light values and coordinates
GLfloat lightPos[] = { 0.0f, 0.0f, 75.0f, 1.0f };
GLfloat specular[] = { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat specref[] = { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat ambientLight[] = { 0.5f, 0.5f, 0.5f, 1.0f };
GLfloat spotDir[] = { 0.0f, 0.0f, -1.0f };
// This function does any needed initialization on the rendering
// context. Here it sets up and initializes the lighting for
// the scene.
void SetupRC()
{
    glEnable(GL_DEPTH_TEST);           // Hidden surface removal
    glFrontFace(GL_CCW);              // Counterclockwise polygons face out
    glEnable(GL_CULL_FACE);            // Do not try to display the back sides
    // Enable lighting
    glEnable(GL_LIGHTING);
    // Set up and enable light 0
    // Supply a slight ambient light so the objects can be seen
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambientLight);
    // The light is composed of just diffuse and specular components
    glLightfv(GL_LIGHT0, GL_DIFFUSE, ambientLight);
    glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
    glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
    // Specific spot effects
    // Cut off angle is 60 degrees
    glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 60.0f);
    // Enable this light in particular
    glEnable(GL_LIGHT0);
    // Enable color tracking
    glEnable(GL_COLOR_MATERIAL);
    // Set material properties to follow glColor values
    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
    // All materials hereafter have full specular reflectivity
    // with a high shine
    glMaterialfv(GL_FRONT, GL_SPECULAR, specref);
    glMateriali(GL_FRONT, GL_SHININESS, 128);
    // Black background
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f );
}
```

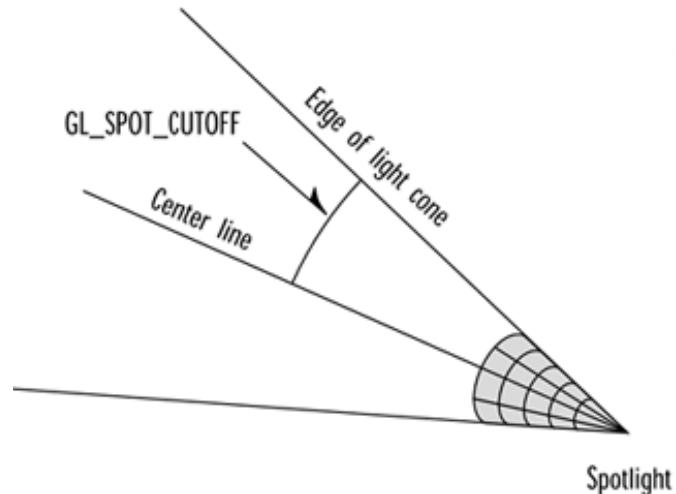
The following line from the listing is actually what makes a positional light source into a spotlight:

```
// Specific spot effects
// Cut off angle is 60 degrees
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 60.0f);
```

The `GL_SPOT_CUTOFF` value specifies the radial angle of the cone of light emanating from the spotlight, from the center line to the edge of the cone. For a normal positional light, this value is 180° so that the light is not confined to a cone. In fact, for spotlights, only values from 0 to 90°

are valid. Spotlights emit a cone of light, and objects outside this cone are not illuminated. [Figure 5.32](#) shows how this angle translates to the cone width.

Figure 5.32. The angle of the spotlight cone.

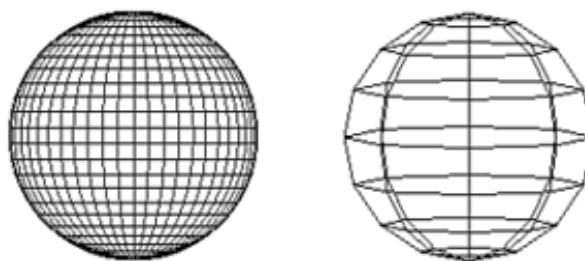


Drawing a Spotlight

When you place a spotlight in a scene, the light must come from somewhere. Just because you have a source of light at some location doesn't mean that you see a bright spot there. For our SPOT sample program, we placed a red cone at the spotlight source to show where the light was coming from. Inside the end of this cone, we placed a bright yellow sphere to simulate a light bulb.

This sample has a pop-up menu that we use to demonstrate several things. The pop-up menu contains items to set flat and smooth shading and to produce a sphere for low, medium, and high tessellation. *Tessellation* means to break a mesh of polygons into a finer mesh of polygons (more vertices). [Figure 5.33](#) shows a wireframe representation of a highly tessellated sphere next to one that has few polygons.

Figure 5.33. On the left is a highly tessellated sphere; on the right, a sphere made up of few polygons.



[Figure 5.34](#) shows our sample in its initial state with the spotlight moved off slightly to one side. (You can use the arrow keys to move the spotlight.) The sphere consists of few polygons, which are flat shaded. In Windows, use the right mouse button to open a pop-up menu (Ctrl-click on the Mac), where you can switch between smooth and flat shading and between very low, medium, and very high tessellation for the sphere. [Listing 5.7](#) shows the complete code to render the scene.

Listing 5.7. The Rendering Function for SPOT, Showing How the Spotlight Is Moved

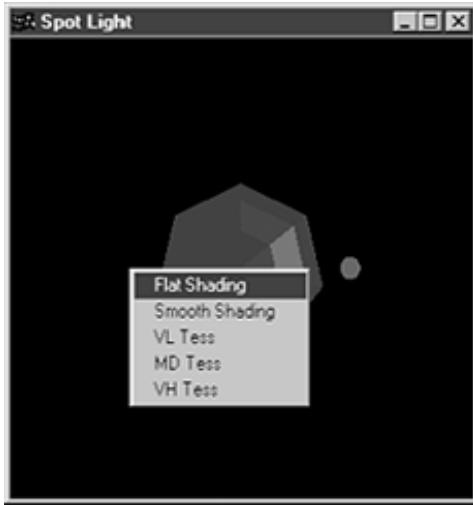
```
// Called to draw scene
```

```

void RenderScene(void)
{
    if(iShade == MODE_FLAT)
        glShadeModel(GL_FLAT);
    else //      iShade = MODE_SMOOTH;
        glShadeModel(GL_SMOOTH);
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // First place the light
    // Save the coordinate transformation
    glPushMatrix();
        // Rotate coordinate system
        glRotatef(yRot, 0.0f, 1.0f, 0.0f);
        glRotatef(xRot, 1.0f, 0.0f, 0.0f);
        // Specify new position and direction in rotated coords
        glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
        glLightfv(GL_LIGHT0,GL_SPOT_DIRECTION,spotDir);
        // Draw a red cone to enclose the light source
        glColor3ub(255,0,0);
        // Translate origin to move the cone out to where the light
        // is positioned.
        glTranslatef(lightPos[0],lightPos[1],lightPos[2]);
        glutSolidCone(4.0f,6.0f,15,15);
        // Draw a smaller displaced sphere to denote the light bulb
        // Save the lighting state variables
        glPushAttrib(GL_LIGHTING_BIT);
            // Turn off lighting and specify a bright yellow sphere
            glDisable(GL_LIGHTING);
            glColor3ub(255,255,0);
            glutSolidSphere(3.0f, 15, 15);
        // Restore lighting state variables
        glPopAttrib();
    // Restore coordinate transformations
    glPopMatrix();
    // Set material color and draw a sphere in the middle
    glColor3ub(0, 0, 255);
    if(iTess == MODE_VERYLOW)
        glutSolidSphere(30.0f, 7, 7);
    else
        if(iTess == MODE_MEDIUM)
            glutSolidSphere(30.0f, 15, 15);
        else //  iTess = MODE_MEDIUM;
            glutSolidSphere(30.0f, 50, 50);
    // Display the results
    glutSwapBuffers();
}

```

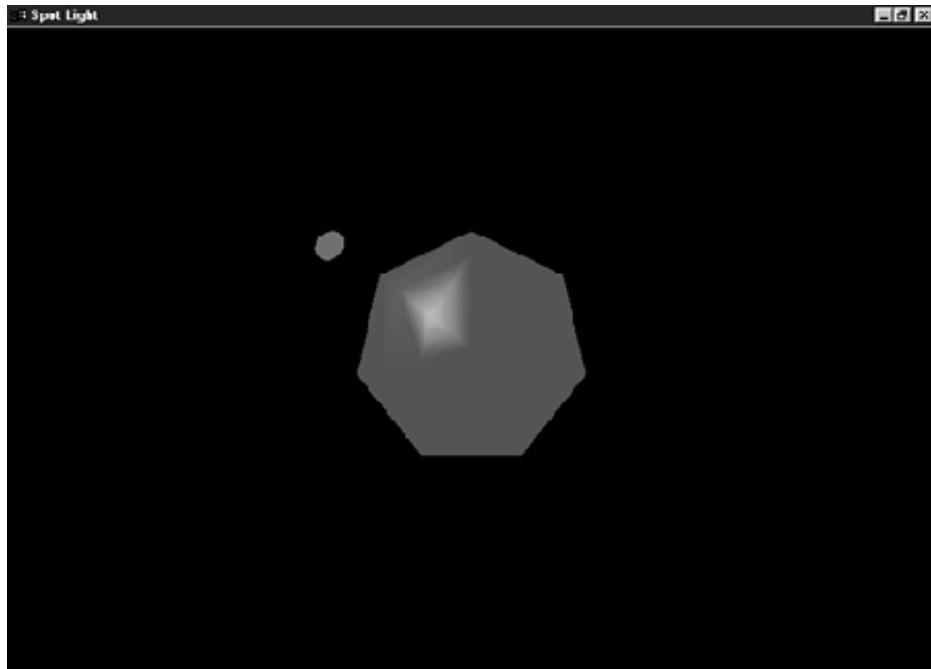
Figure 5.34. The SPOT sample—low tessellation, flat shading.



The variables `iTess` and `iMode` are set by the GLUT menu handler and control how many sections the sphere is broken into and whether flat or smooth shading is employed. Note that the light is positioned before any geometry is rendered. As pointed out in [Chapter 2](#), OpenGL is an immediate-mode API: If you want an object to be illuminated, you have to put the light where you want it before drawing the object.

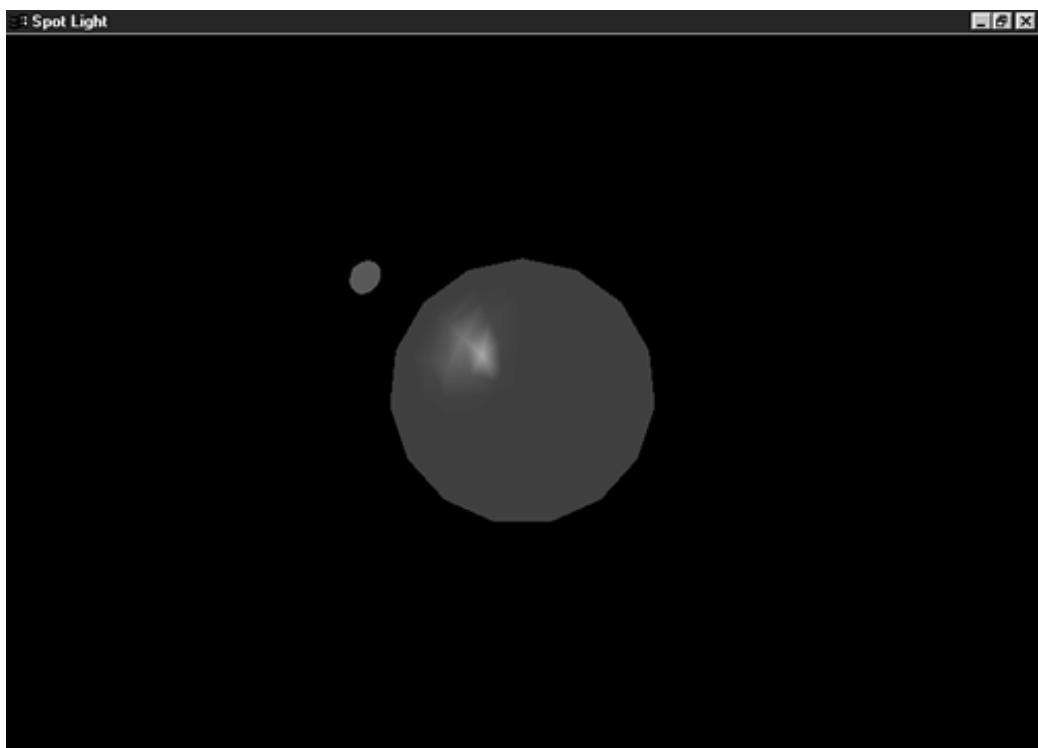
You can see in [Figure 5.34](#) that the sphere is coarsely lit and each flat face is clearly evident. Switching to smooth shading helps a little, as shown in [Figure 5.35](#).

Figure 5.35. Smoothly shaded but inadequate tessellation.



Increasing the tessellation helps, as shown in [Figure 5.36](#), but you still see disturbing artifacts as you move the spotlight around the sphere. These lighting artifacts are one of the drawbacks of OpenGL lighting. A better way to characterize this situation is that these artifacts are a drawback of vertex lighting (not necessarily OpenGL!). By lighting the vertices and then interpolating between them, we get a crude approximation of lighting. This approach is sufficient for many cases, but as you can see in our spot example, it is not sufficient in others. If you switch to very high tessellation and move the spotlight, you see the lighting blemishes all but vanish.

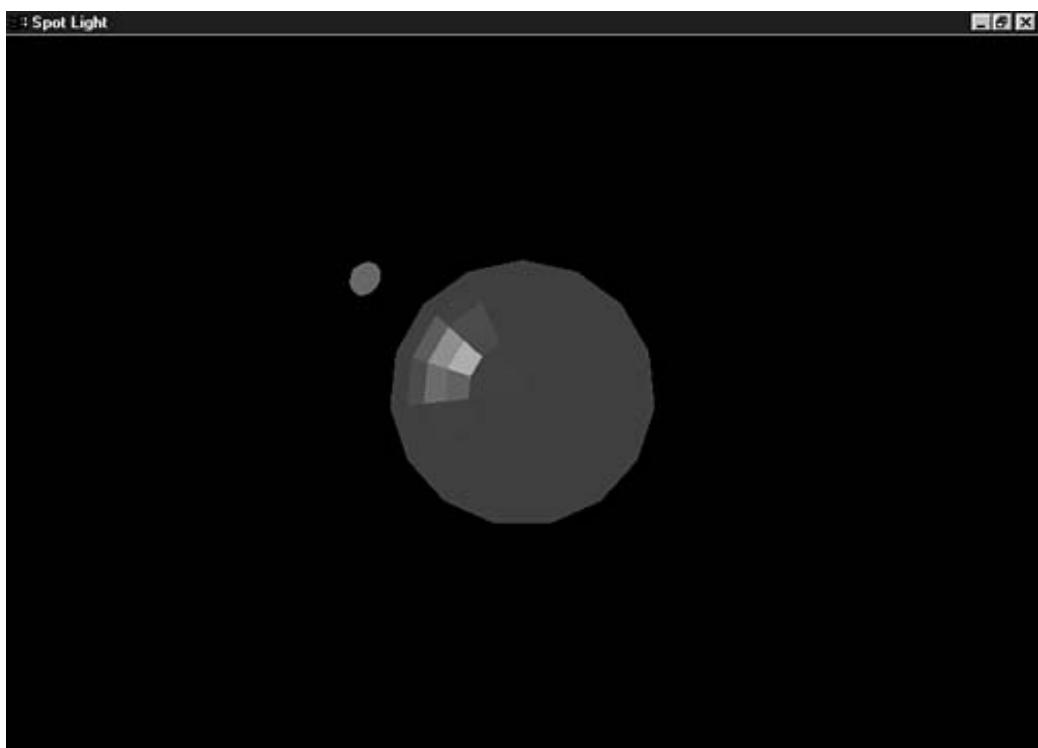
Figure 5.36. Choosing a finer mesh of polygons yields better vertex lighting.



As OpenGL hardware accelerators begin to accelerate transformations and lighting effects, and as CPUs become more powerful, you will be able to more finely tessellate your geometry for better OpenGL-based lighting effects.

The final observation you need to make about the SPOT sample appears when you set the sphere for medium tessellation and flat shading. As shown in [Figure 5.37](#), each face of the sphere is flatly lit. Each vertex is the same color but is modulated by the value of the normal and the light. With flat shading, each polygon is made the color of the last vertex color specified and not smoothly interpolated between each one.

Figure 5.37. A multifaceted sphere.



Shadows

A chapter on color and lighting naturally calls for a discussion of shadows. Adding shadows to your scenes can greatly improve their realism and visual effectiveness. In [Figures 5.38](#) and [5.39](#), you see two views of a lighted cube. Although both are lit, the one with a shadow is more convincing than the one without the shadow.

Figure 5.38. Lighted cube without a shadow.

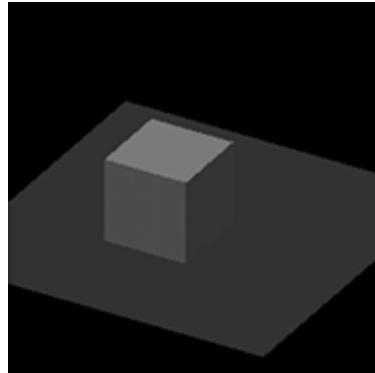
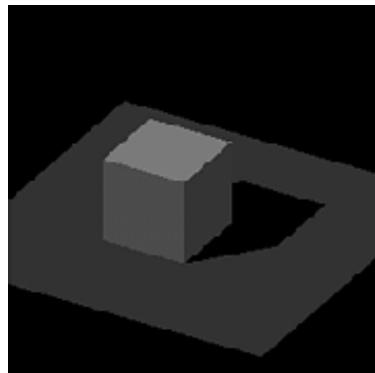


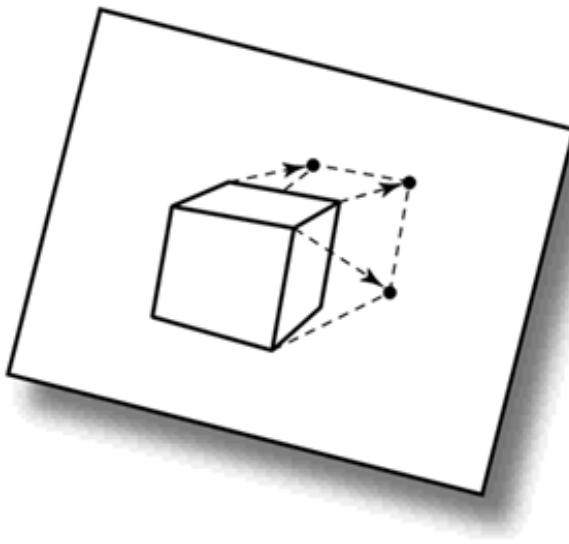
Figure 5.39. Lighted cube with a shadow.



What Is a Shadow?

Conceptually, drawing a shadow is quite simple. A shadow is produced when an object keeps light from a light source from striking some object or surface behind the object casting the shadow. The area on the shadowed object's surface, outlined by the object casting the shadow, appears dark. We can produce a shadow programmatically by flattening the original object into the plane of the surface in which the object lies. The object is then drawn in black or some dark color, perhaps with some translucence. There are many methods and algorithms for drawing shadows, some quite complex. This book's primary focus is on the OpenGL API. It is our hope that, after you've mastered the tool, some of the additional reading suggested in [Appendix A](#) will provide you with a lifetime of learning new applications for this tool. [Chapter 18](#), "Depth Textures and Shadows," covers some new direct support in OpenGL for making shadows; for our purposes in this chapter, we demonstrate one of the simpler methods that works quite well when casting shadows on a flat surface (such as the ground). [Figure 5.40](#) illustrates this flattening.

Figure 5.40. Flattening an object to create a shadow.



We squish an object against another surface by using some of the advanced matrix manipulations we touched on in the preceding chapter. Here, we boil down this process to make it as simple as possible.

Squish Code

We need to flatten the modelview projection matrix so that any and all objects drawn into it are now in this flattened two-dimensional world. No matter how the object is oriented, it is squished into the plane in which the shadow lies. The next two considerations are the distance and direction of the light source. The direction of the light source determines the shape of the shadow and influences the size. If you've ever seen your shadow in the early or late morning hours, you know how long and warped your shadow can appear depending on the position of the sun.

The function `gltMakeShadowMatrix` from the `glTools` library, shown in [Listing 5.8](#), takes three points that lie in the plane in which you want the shadow to appear (these three points cannot be along the same straight line!), the position of the light source, and finally a pointer to a transformation matrix that this function constructs. We won't delve too much into linear algebra, but you do need to know that this function deduces the coefficients of the equation of the plane in which the shadow appears and uses it along with the lighting position to build a transformation matrix. If you multiply this matrix by the current modelview matrix, all further drawing is flattened into this plane.

Listing 5.8. Function to Make a Shadow Transformation Matrix

```

// Creates a shadow projection matrix out of the plane equation
// coefficients and the position of the light. The return value is stored
// in destMat
void gltMakeShadowMatrix(GLTVector3 vPoints[3], GLTVector4 vLightPos,
                        GLTMatrix destMat)
{
    GLTVector4 vPlaneEquation;
    GLfloat dot;
    gltGetPlaneEquation(vPoints[0], vPoints[1], vPoints[2], vPlaneEquation);
    // Dot product of plane and light position
    dot = vPlaneEquation[0]*vLightPos[0] +
          vPlaneEquation[1]*vLightPos[1] +
          vPlaneEquation[2]*vLightPos[2] +
          vPlaneEquation[3]*vLightPos[3];
    // Now do the projection
    // First column
    destMat[0] = dot - vLightPos[0] * vPlaneEquation[0];
    destMat[4] = 0.0f - vLightPos[0] * vPlaneEquation[1];

```

```

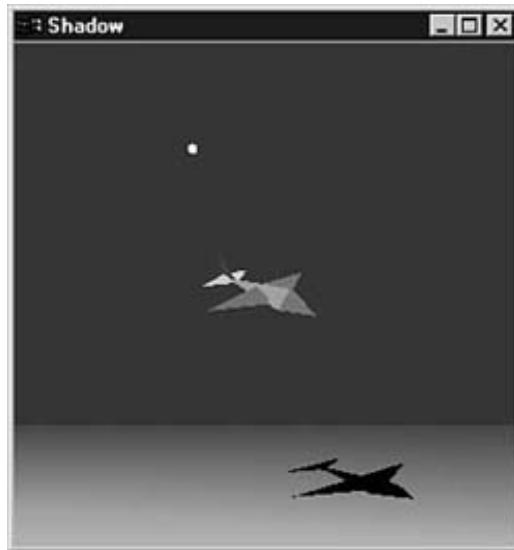
destMat[8] = 0.0f - vLightPos[0] * vPlaneEquation[2];
destMat[12] = 0.0f - vLightPos[0] * vPlaneEquation[3];
// Second column
destMat[1] = 0.0f - vLightPos[1] * vPlaneEquation[0];
destMat[5] = dot - vLightPos[1] * vPlaneEquation[1];
destMat[9] = 0.0f - vLightPos[1] * vPlaneEquation[2];
destMat[13] = 0.0f - vLightPos[1] * vPlaneEquation[3];
// Third Column
destMat[2] = 0.0f - vLightPos[2] * vPlaneEquation[0];
destMat[6] = 0.0f - vLightPos[2] * vPlaneEquation[1];
destMat[10] = dot - vLightPos[2] * vPlaneEquation[2];
destMat[14] = 0.0f - vLightPos[2] * vPlaneEquation[3];
// Fourth Column
destMat[3] = 0.0f - vLightPos[3] * vPlaneEquation[0];
destMat[7] = 0.0f - vLightPos[3] * vPlaneEquation[1];
destMat[11] = 0.0f - vLightPos[3] * vPlaneEquation[2];
destMat[15] = dot - vLightPos[3] * vPlaneEquation[3];
}

```

A Shadow Example

To demonstrate the use of the function in [Listing 5.8](#), we suspend our jet in air high above the ground. We place the light source directly above and a bit to the left of the jet. As you use the arrow keys to spin the jet around, the shadow cast by the jet appears flattened on the ground below. The output from this SHADOW sample program is shown in [Figure 5.41](#).

Figure 5.41. Output from the SHADOW sample program.



The code in [Listing 5.9](#) shows how the shadow projection matrix was created for this example. Note that we create the matrix once in `SetupRC` and save it in a global variable.

Listing 5.9. Setting Up the Shadow Projection Matrix

```

GLfloat lightPos[] = { -75.0f, 150.0f, -50.0f, 0.0f };
...
...
// Transformation matrix to project shadow
GLTMatrix shadowMat;
...
...

```

```

// This function does any needed initialization on the rendering
// context. Here it sets up and initializes the lighting for
// the scene.
void SetupRC()
{
    // Any three points on the ground (counterclockwise order)
    GLTVector3 points[3] = {{ -30.0f, -149.0f, -20.0f },
                           { -30.0f, -149.0f, 20.0f },
                           { 40.0f, -149.0f, 20.0f }};
    glEnable(GL_DEPTH_TEST);      // Hidden surface removal
    glFrontFace(GL_CCW);        // Counterclockwise polygons face out
    glEnable(GL_CULL_FACE);      // Do not calculate inside of jet
    // Enable lighting
    glEnable(GL_LIGHTING);
    ...
    // Code to set up lighting, etc.
    ...
    // Light blue background
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f );
    // Calculate projection matrix to draw shadow on the ground
    gltMakeShadowMatrix(points, lightPos, shadowMat);
}

```

[Listing 5.10](#) shows the rendering code for the shadow example. We first draw the ground. Then we draw the jet as we normally do, restore the modelview matrix, and multiply it by the shadow matrix. This procedure creates our squish matrix. Then we draw the jet again. (We've modified our code to accept a flag telling the `DrawJet` function to render in color or black.) After restoring the modelview matrix once again, we draw a small yellow sphere to approximate the position of the light. Note that we disable depth testing before we draw a plane below the jet to indicate the ground.

This rectangle lies in the same plane in which our shadow is drawn, and we want to make sure the shadow is drawn. We have never before discussed what happens if we draw two objects or planes in the same location. We have discussed depth testing as a means to determine what is drawn in front of what, however. If two objects occupy the same location, usually the last one drawn is shown. Sometimes, however, an effect called *z-fighting* causes fragments from both objects to be intermingled, resulting in a mess!

Listing 5.10. Rendering the Jet and Its Shadow

```

// Called to draw scene
void RenderScene(void)
{
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // Draw the ground; we do manual shading to a darker green
    // in the background to give the illusion of depth
    glBegin(GL_QUADS);
        glColor3ub(0,32,0);
        glVertex3f(400.0f, -150.0f, -200.0f);
        glVertex3f(-400.0f, -150.0f, -200.0f);
        glColor3ub(0,255,0);
        glVertex3f(-400.0f, -150.0f, 200.0f);
        glVertex3f(400.0f, -150.0f, 200.0f);
    glEnd();
    // Save the matrix state and do the rotations
    glPushMatrix();
    // Draw jet at new orientation; put light in correct position
    // before rotating the jet
    glEnable(GL_LIGHTING);

```

```

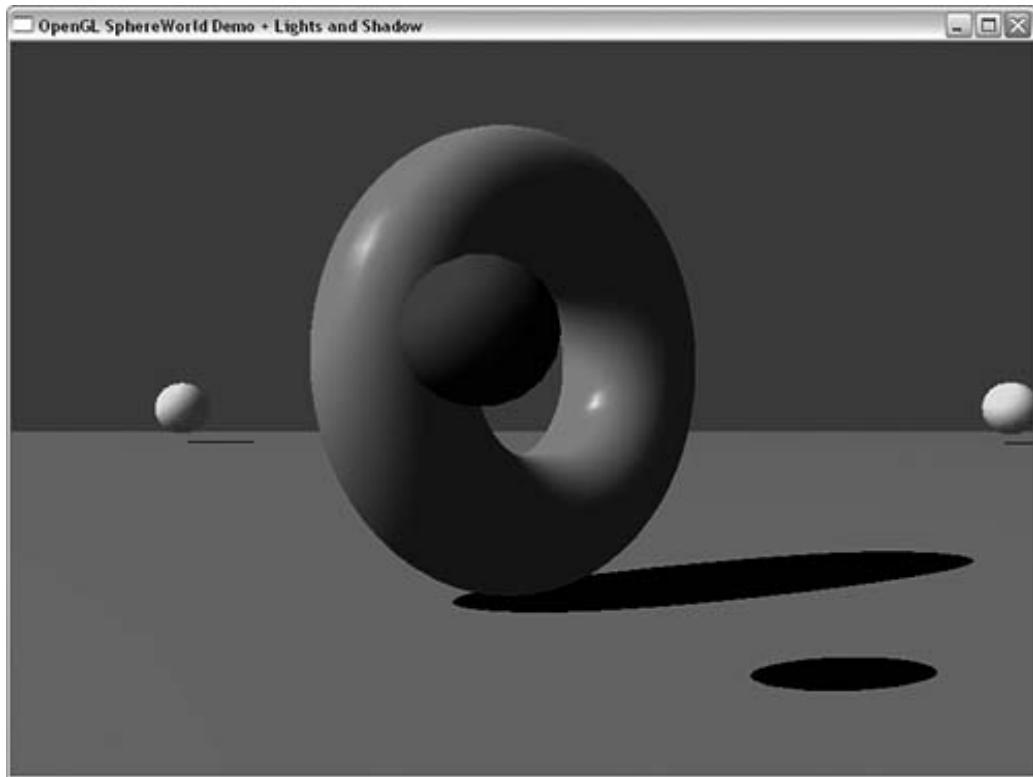
glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
glRotatef(xRot, 1.0f, 0.0f, 0.0f);
glRotatef(yRot, 0.0f, 1.0f, 0.0f);
DrawJet(FALSE);
// Restore original matrix state
glPopMatrix();
// Get ready to draw the shadow and the ground
// First disable lighting and save the projection state
glDisable(GL_DEPTH_TEST);
glDisable(GL_LIGHTING);
glPushMatrix();
// Multiply by shadow projection matrix
glMultMatrixf((GLfloat *)shadowMat);
// Now rotate the jet around in the new flattened space
glRotatef(xRot, 1.0f, 0.0f, 0.0f);
glRotatef(yRot, 0.0f, 1.0f, 0.0f);
// Pass true to indicate drawing shadow
DrawJet(TRUE);
// Restore the projection to normal
glPopMatrix();
// Draw the light source
glPushMatrix();
glTranslatef(lightPos[0], lightPos[1], lightPos[2]);
glColor3ub(255,255,0);
glutSolidSphere(5.0f,10,10);
glPopMatrix();
// Restore lighting state variables
glEnable(GL_DEPTH_TEST);
// Display the results
glutSwapBuffers();
}

```

Sphere World Revisited

Our last example for this chapter is too long to list the source code in its entirety. In the preceding chapter's SPHEREWORLD sample program, we created an immersive 3D world with animation and camera movement. In this chapter, we revisit Sphere World and have added lights and material properties to the torus and sphere inhabitants. Finally, we have also used our planar shadow technique to add a shadow to the ground! We will keep coming back to this example from time to time as we add more and more of our OpenGL functionality to the code. The output of this chapter's version of SPHEREWORLD is shown in [Figure 5.42](#).

Figure 5.42. Fully lit and shadowed Sphere World.



Summary

This chapter introduced some of the more magical and powerful capabilities of OpenGL. We started by adding color to 3D scenes and smooth shading. We then saw how to specify one or more light sources and define their lighting characteristics in terms of ambient, diffuse, and specular components. We explained how the corresponding material properties interact with these light sources and demonstrated some special effects, such as adding specular highlights and softening sharp edges between adjoining triangles.

Also covered were lighting positions and the creation and manipulation of spotlights. The high-level matrix munching function presented here makes shadow generation as easy as it gets for planar shadows.

Reference

glColor

Purpose: Sets the current color when in RGBA color mode.

Include File: `<gl.h>`

Variations:

```
void glColor3b(GLbyte red, GLbyte green, GLbyte blue);
void glColor3d(GLdouble red, GLdouble green,
    GLdouble blue);
void glColor3f(GLfloat red, GLfloat green, GLfloat
    blue);
void glColor3i(GLint red, GLint green, GLint blue);
void glColor3s(GLshort red, GLshort green, GLshort
    blue);
void glColor3ub(GLubyte red, GLubyte green,
    GLubyte blue);
void glColor3ui(GLuint red, GLuint green, GLuint
    blue);
void glColor3us(GLushort red, GLushort green,
    GLushort blue);
void glColor4b(GLbyte red, GLbyte green, GLbyte
    blue, GLbyte alpha);
void glColor4d(GLdouble red, GLdouble green,
    GLdouble blue, GLdouble alpha);
void glColor4f(GLfloat red, GLfloat green, GLfloat
    blue, GLfloat alpha);
void glColor4i(GLint red, GLint green, GLint blue,
    GLint alpha);
void glColor4s(GLshort red, GLshort green, GLshort
    blue, GLshort alpha);
void glColor4ub(GLubyte red, GLubyte green,
    GLubyte blue, GLubyte alpha);
void glColor4ui(GLuint red, GLuint green, GLuint
    blue, GLuint alpha);
void glColor4us(GLushort red, GLushort green,
    GLushort blue, GLushort alpha);
void glColor3bv(const GLbyte *v);
void glColor3dv(const GLdouble *v);
void glColor3fv(const GLfloat *v);
void glColor3iv(const GLint *v);
void glColor3sv(const GLshort *v);
void glColor3ubv(const GLubyte *v);
void glColor3uiv(const GLuint *v);
void glColor3usv(const GLushort *v);
void glColor4bv(const GLbyte *v);
void glColor4dv(const GLdouble *v);
void glColor4fv(const GLfloat *v);
void glColor4iv(const GLint *v);
void glColor4sv(const GLshort *v);
void glColor4ubv(const GLubyte *v);
void glColor4uiv(const GLuint *v);
void glColor4usv(const GLushort *v);
```

Description: This function sets the current color by specifying separate red, green, and blue components of the color. Some functions also accept an alpha component. Each component represents the range of intensity from zero (0.0) to full intensity (1.0). Functions with the *v* suffix take a pointer to an array that specifies the components. Each element in the array must be the same type. When the alpha component is not specified, it is implicitly set to 1.0. When non-floating-point types are specified, the range from zero to the largest value represented by that type is mapped to the floating-point range 0.0 to 1.0.

Parameters:

red Specifies the red component of the color.
green Specifies the green component of the color.
blue Specifies the blue component of the color.
alpha Specifies the alpha component of the color. Used only in variations that take four arguments.
**v* A pointer to an array of red, green, blue, and possibly alpha values.

Returns: None.

See Also: [glColorMaterial](#), [glMaterial](#)

glColorMask

Purpose: Enables or disables modification of color components in the color buffers.

Include File: `<gl.h>`

Syntax:

```
void glColorMask(GLboolean bRed, GLboolean bGreen,
  ↪ GLboolean bBlue,
GLboolean bAlpha);
```

Description: This function allows changes to individual color components in the color buffer to be disabled or enabled. (All are enabled by default.) For example, setting the *bAlpha* argument to `GL_FALSE` disallows changes to the alpha color components.

Parameters:

bRed `GLboolean`: Specifies whether the red component can be modified.
bGreen `GLboolean`: Specifies whether the green component can be modified.
bBlue `GLboolean`: Specifies whether the blue component can be modified.
bAlpha `GLboolean`: Specifies whether the alpha component can be modified.

Returns: None.

See Also: [glColor](#)

glColorMaterial

Purpose: Allows material colors to track the current color as set by `glColor`.

Include File: `<gl.h>`

Syntax:

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

Description: This function allows material properties to be set without having to call `glMaterial` directly. By using this function, you can set certain material properties to follow the current color as specified by `glColor`. By default, color tracking is disabled; to enable it, you must also call `glEnable(GL_COLOR_MATERIAL)`. To disable color tracking again, you call `glDisable(GL_COLOR_MATERIAL)`.

Parameters:

`face` `GLenum`: Specifies whether the front (`GL_FRONT`), back (`GL_BACK`), or both (`GL_FRONT_AND_BACK`) should follow the current color.

`mode` `GLenum`: Specifies which material property should be following the current color. This can be `GL_EMISSION`, `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`, or `GL_AMBIENT_AND_DIFFUSE`.

Returns: None.

See Also: `glColor`, `glMaterial`, `glLight`, `glLightModel`

glGetLight

Purpose: Returns information about the current light source settings.

Include File: `<gl.h>`

Variations:

```
void glGetLightfv(GLenum light, GLenum pname,
  ↵ GLfloat *params);
void glGetLightiv(GLenum light, GLenum pname,
  ↵ GLint *params);
```

Description: You use this function to query the current settings for one of the eight supported light sources. The return values are stored at the address pointed to by `params`. For most properties, this is an array of four values containing the RGBA components of the properties specified.

Parameters:

`light` `GLenum`: Specifies the light source for which information is being requested. This ranges from 0 to `GL_MAX_LIGHTS` (a minimum of 8 is required by specification). Constant light values are enumerated from `GL_LIGHT0` to `GL_LIGHT7`.

<i>pname</i>	<i>GLenum</i> : Specifies which property of the light source is being queried. Any of the following values are valid: <i>GL_AMBIENT</i> , <i>GL_DIFFUSE</i> , <i>GL_SPECULAR</i> , <i>GL_POSITION</i> , <i>GL_SPOT_DIRECTION</i> , <i>GL_SPOT_EXPONENT</i> , <i>GL_SPOT_CUTOFF</i> , <i>GL_CONSTANT_ATTENUATION</i> , <i>GL_LINEAR_ATTENUATION</i> , and <i>GL_QUADRATIC_ATTENUATION</i> .
<i>params</i>	<i>GLfloat*</i> or <i>GLint*</i> : Specifies an array of integer or floating-point values representing the return values. These return values are in the form of an array of four or three or a single value. Table 5.2 shows the return value meanings for each property.

Table 5.2. Valid Lighting Parameters for *glGetLight*

Property	Meaning of Return Values
<i>GL_AMBIENT</i>	Four RGBA components.
<i>GL_DIFFUSE</i>	Four RGBA components.
<i>GL_SPECULAR</i>	Four RGBA components.
<i>GL_POSITION</i>	Four elements that specify the position of the light source. The first three elements specify the position of the light. The fourth, if 1.0, specifies that the light is at this position. Otherwise, the light source is directional and all rays are parallel.
<i>GL_SPOT_DIRECTION</i>	Three elements specifying the direction of the spotlight. This vector is not normalized and is in eye coordinates.
<i>GL_SPOT_EXPONENT</i>	A single value representing the spot exponent.
<i>GL_SPOT_CUTOFF</i>	A single value representing the cutoff angle of the spot source.
<i>GL_CONSTANT_ATTENUATION</i>	A single value representing the constant attenuation of the light.
<i>GL_LINEAR_ATTENUATION</i>	A single value representing the linear attenuation of the light.
<i>GL_QUADRATIC_ATTENUATION</i>	A single value representing the quadratic attenuation of the light.

Returns: None.**See Also:** [glLight](#)**glGetMaterial****Purpose:** Returns the current material property settings.**Include File:** [<gl.h>](#)**Variations:**

```
void glGetMaterialfv(GLenum face, GLenum pname,
    ➔ GLfloat *params);
void glGetMaterialiv(GLenum face, GLenum pname,
    ➔ GLint *params);
```

Description: You use this function to query the current front or back material properties. The return values are stored at the address pointed to by *params*. For most properties, this is an array of four values containing the RGBA components of the property specified.

Parameters:

face GLenum: Specifies whether the front (GL_FRONT) or back (GL_BACK) material properties are being sought.

pname GLenum: Specifies which material property is being queried. Valid values are GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION, GL_SHININESS, and GL_COLOR_INDEXES.

params GLint* or GLfloat*: Specifies an array of integer or floating-point values representing the return values. For GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, and GL_EMISSION, this is a four-element array containing the RGBA values of the property specified. For GL_SHININESS, a single value representing the specular exponent of the material is returned. GL_COLOR_INDEXES returns an array of three elements containing the ambient, diffuse, and specular components in the form of color indexes. GL_COLOR_INDEXES is used only for color index lighting.

Returns: None.

See Also: [glMaterial](#)

glLight

Purpose: Sets light source parameters for one of the available light sources.

Include File: <gl.h>

Variations:

```
void glLightf(GLenum light, GLenum pname, GLfloat
    ➔ param);
void glLighti(GLenum light, GLenum pname, GLint
    ➔ param);
void glLightfv(GLenum light, GLenum pname, const
    ➔ GLfloat*params);
void glLightiv(GLenum light, GLenum pname, const
    ➔ GLint *params);
```

Description: You use this function to set the lighting parameters for one of the eight supported light sources. The first two variations of this function require only a single parameter value to set one of the following properties: `GL_SPOT_EXPONENT`, `GL_SPOT_CUTOFF`, `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUATION`, and `GL_QUADRATIC_ATTENUATION`. The second two variations are used for lighting parameters that require an array of multiple values. They include `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`, `GL_POSITION`, and `GL_SPOT_DIRECTION`. You can also use these variations with single-valued parameters by specifying a single element array for `*params`.

Parameters:

`light` `GLenum`: Specifies which light source is being modified. This ranges from 0 to `GL_MAX_LIGHTS` (8 minimum). Constant light values are enumerated from `GL_LIGHT0` to `GL_LIGHT7`.

`pname` `GLenum`: Specifies which lighting parameter is being set by this function call. See [Table 5.2](#) for a complete list and the meaning of these parameters.

`param` `GLfloat` or `GLint`: Specifies the value for parameters that are specified by a single value. These parameters are `GL_SPOT_EXPONENT`, `GL_SPOT_CUTOFF`, `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUATION`, and `GL_QUADRATIC_ATTENUATION`. These parameters have meaning only for spotlights.

`params` `GLfloat*` or `GLint*`: Specifies an array of values that fully describe the parameters being set. See [Table 5.2](#) for a list and the meaning of these parameters.

Returns: None.

See Also: [glGetLight](#)

glLightModel

Purpose: Sets the lighting model parameters used by OpenGL.

Include File: `<gl.h>`

Variations:

```
void glLightModelf(GLenum pname, GLfloat param)
void glLightModelfi(GLenum pname, GLint param);
void glLightModelfv(GLenum pname, const GLfloat
  *params);
void glLightModeliv(GLenum pname, const GLint
  *params);
```

Description: You use this function to set the lighting model parameters used by OpenGL. You can set any or all of three lighting model parameters. `GL_LIGHT_MODEL_AMBIENT` is used to set the default ambient illumination for a scene. By default, this light has an RGBA value of (0.2, 0.2, 0.2, 1.0). Only the last two variations can be used to set this lighting model because they take pointers to an array that can contain the RGBA values.

The `GL_LIGHT_MODEL_TWO_SIDE` parameter is specified to indicate whether both sides of polygons are illuminated. By default, only the front (defined by winding) of polygons is illuminated, using the front material properties as specified by `glMaterial`. Specifying a lighting model parameter of `GL_LIGHT_MODEL_LOCAL_VIEWER` modifies the calculation of specular reflection angles, whether the view is down along the negative z-axis or from the origin of the eye coordinate system. Finally, `GL_LIGHT_MODEL_COLOR` control can be used to specify whether specular lighting produces a second color (textures get specular light) or whether all three lighting components are combined with `GL_SINGLE_COLOR`.

Parameters:

pname `GLenum`: Specifies a lighting model parameter. `GL_LIGHT_MODEL_AMBIENT`, `GL_LIGHT_MODEL_LOCAL_VIEWER`, `GL_LIGHT_MODEL_TWO_SIDE`, and `GL_LIGHT_MODEL_COLOR_CONTROL` are accepted.

param `GLfloat` or `GLint`: For `GL_LIGHT_MODEL_LOCAL_VIEWER`, a value of 0.0 indicates that specular lighting angles take the view direction to be parallel to and in the direction of the negative z-axis. Any other value indicates that the view is from the origin of eye coordinate system. For `GL_LIGHT_MODEL_TWO_SIDE`, a value of 0.0 indicates that only the fronts of polygons are to be included in illumination calculations. Any other value indicates that both the front and back are included. This parameter has no effect on points, lines, or bitmaps. For `GL_LIGHT_MODEL_COLOR_CONTROL`, the valid values are `GL_SEPARATE_SPECULAR_COLOR` or `GL_SINGLE_COLOR`.

params `GLfloat*` or `GLint*`: For `GL_LIGHT_MODEL_AMBIENT` or `GL_LIGHT_MODEL_LOCAL_VIEWER`, this points to an array of integers or floating-point values, only the first element of which is used to set the parameter value. For `GL_LIGHT_MODEL_AMBIENT`, this array points to four values that indicate the RGBA components of the ambient light.

Returns: None.

See Also: `glLight`, `glMaterial`

glMaterial

Purpose: Sets material parameters for use by the lighting model.

Include File: `<gl.h>`

Variations:

```

void glMaterialf(GLenum face, GLenum pname,
  ↪ GLfloat param);
void glMateriali(GLenum face, GLenum pname, GLint
  ↪ param);
void glMaterialfv(GLenum face, GLenum pname, const
  ↪ GLfloat params)
void glMaterialiv(GLenum face, GLenum pname, const
  ↪ GLint params);

```

Description: You use this function to set the material reflectance properties of polygons. The `GL_AMBIENT`, `GL_DIFFUSE`, and `GL_SPECULAR` properties affect how these components of incident light are reflected. `GL_EMISSION` is used for materials that appear to give off their own light. `GL_SHININESS` can vary from 0 to 128, with the higher values producing a larger specular highlight on the material surface. You use `GL_COLOR_INDEXES` for material reflectance properties in color index mode.

Parameters:

`face` `GLenum`: Specifies whether the front, back, or both material properties of the polygons are being set by this function. May be either `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`.

`pname` `GLenum`: Specifies the single-valued material parameter being set for the first two variations. Currently, the only single-valued material parameter is `GL_SHININESS`. The second two variations, which take arrays for their parameters, can set the following material properties: `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`, `GL_EMISSION`, `GL_SHININESS`, `GL_AMBIENT_AND_DIFFUSE`, or `GL_COLOR_INDEXES`.

`param` `GLfloat` or `GLint`: Specifies the value to which the parameter specified by `pname` (`GL_SHININESS`) is set.

`params` `GLfloat*` or `GLint*`: Specifies an array of floats or integers that contain the components of the property being set.

Returns: None.

See Also: `glGetMaterial`, `glColorMaterial`, `glLight`, `glLightModel`

glNormal

Purpose: Defines a surface normal for the next vertex or set of vertices specified.

Include File: `<gl.h>`

Variations:

```

void glNormal3b(GLbyte nx, GLbyte ny, GLbyte nz);
void glNormal3d(GLdouble nx, GLdouble ny, GLdouble
    ↪ nz);
void glNormal3f(GLfloat nx, GLfloat ny, GLfloat nz);
void glNormal3i(GLint nx, GLint ny, GLint nz);
void glNormal3s(GLshort nx, GLshort ny, GLshort nz);
void glNormal3bv(const GLbyte *v);
void glNormal3dv(const GLdouble *v);
void glNormal3fv(const GLfloat *v);
void glNormal3iv(const GLint *v);
void glNormal3sv(const GLshort *v);

```

Description: The normal vector specifies which direction is up and perpendicular to the surface of the polygon. This function is used for lighting and shading calculations. Specifying a unit vector of length 1 improves rendering speed. OpenGL automatically converts your normals to unit normals if you enable this with `glEnable(GL_NORMALIZE);`.

Parameters:

`nx` Specifies the x magnitude of the normal vector.
`ny` Specifies the y magnitude of the normal vector.
`nz` Specifies the z magnitude of the normal vector.
`v` Specifies an array of three elements containing the x, y, and z magnitudes of the normal vector.

Returns: None.

See Also: `glTexCoord`, `glVertex`

glShadeModel

Purpose: Sets the default shading to flat or smooth.

Include File: `<gl.h>`

Syntax:

```
void glShadeModel(GLenum mode);
```

Description: OpenGL primitives are always shaded, but the shading model can be flat (`GL_FLAT`) or smooth (`GL_SMOOTH`). In the simplest of scenarios, one color is set with `glColor` before a primitive is drawn. This primitive is solid and flat (does not vary) throughout, regardless of the shading. If a different color is specified for each vertex, the resulting image varies with the shading model. With smooth shading, the color of the polygon's interior points are interpolated from the colors specified at the vertices. This means the color varies from one color to the next between two vertices. The color variation follows a line through the color cube between the two colors. If lighting is enabled, OpenGL performs other calculations to determine the correct value for each vertex. In flat shading, the color specified for the last vertex is used throughout the region of the primitive. The only exception is for `GL_POLYGON`, in which case, the color used throughout the region is the one specified for the first vertex.

Parameters:

mode GLenum: Specifies the shading model to use, either `GL_FLAT` or `GL_SMOOTH`. The default is `GL_SMOOTH`.

Returns: None.

See Also: `glColor`, `glLight`, `glLightModel`

Chapter 6. More on Colors and Materials

by Richard S. Wright, Jr.

WHAT YOU'LL LEARN IN THIS CHAPTER:

How To	Functions You'll Use
Blend colors and objects together	<code>glBlendFunc</code> , <code>glBlendFuncSeparate</code> , <code>glBlendEquation</code> , <code>glBlendColor</code>
Use alpha testing to eliminate fragments	<code>glAlphaFunc</code>
Add depth cues with fog	<code>glFog</code>
Render motion-blurred animation	<code>glAccum</code>

In the preceding chapter, you learned that there is more to making a ball appear red than just setting the drawing color to red. Material properties and lighting parameters can go a long way toward adding realism to your graphics, but modeling the real world has a few other challenges that we will address in this chapter. For starters, many effects are accomplished by means of blending colors together. Transparent objects such as stained glass windows or plastic bottles allow you to see through them, but the light from the objects behind them is blended with the color of the transparent object you are seeing through. This type of transparency is achieved in OpenGL by drawing the background objects first and then blending the foreground object in front with the colors that are already present in the color buffer. A good part of making this technique work requires that we now consider the fourth color component that until now we have been ignoring, *alpha*.

Blending

You have already learned that OpenGL rendering places color values in the color buffer under normal circumstances. You have also learned that depth values for each fragment are also placed in the depth buffer. When depth testing is turned off (disabled), new color values simply overwrite any other values already present in the color buffer. When depth testing is turned on (enabled), new color fragments replace an existing fragment only if they are deemed closer to the near clipping plane than the values already there. This is, of course, under *normal* circumstances. These rules suddenly no longer apply the moment you turn on OpenGL blending:

```
glEnable(GL_BLENDING);
```

When blending is enabled, the incoming color is combined with the color value already present in the color buffer. How these colors are combined leads to a great many and varied special effects.

Combining Colors

First, we must introduce a more official terminology for the color values coming in and already in the color buffer. The color value already stored in the color buffer is called the *destination* color, and this color value contains the three individual red, green, and blue components, and optionally a stored alpha value as well. A color value that is coming in as a result of more rendering commands that may or may not interact with the destination color is called the *source* color. The source color also contains either three or four color components (red, green, blue, and optionally alpha).

How the source and destination colors are combined when blending is enabled is controlled by the blending equation. By default, the blending equation looks like this:

$$C_f = (C_s * S) + (C_d * D)$$

Here, C_f is the final computed color, C_s is the source color, C_d is the destination color, and S and D are the source and destination blending factors. These blending factors are set with the following function:

```
glBlendFunc(GLenum S, GLenum D);
```

As you can see, S and D are enumerants and not physical values that you specify directly. [Table 6.1](#) lists the possible values for the blending function. The subscripts stand for source, destination, and color (for blend color, to be discussed shortly). R , G , B , and A stand for Red, Green, Blue, and Alpha, respectively.

Table 6.1. OpenGL Blending Factors

Function	RGB Blend Factors	Alpha Blend Factor
GL_ZERO	(0,0,0)	0
GL_ONE	(1,1,1)	1
GL_SRC_COLOR	(R_s, G_s, B_s)	A_s
GL_ONE_MINUS_SRC_COLOR	$(1,1,1) - (R_s, G_s, B_s)$	$1 - A_s$
GL_DST_COLOR	(R_d, G_d, B_d)	A_d
GL_ONE_MINUS_DST_COLOR	$(1,1,1) - (R_d, G_d, B_d)$	$1 - A_d$
GL_SRC_ALPHA	(A_s, A_s, A_s)	A_s
GL_ONE_MINUS_SRC_ALPHA	$(1,1,1) - (A_s, A_s, A_s)$	$1 - A_s$
GL_DST_ALPHA	(A_d, A_d, A_d)	A_d
GL_ONE_MINUS_DST_ALPHA	$(1,1,1) - (A_d, A_d, A_d)$	$1 - A_d$
GL_CONSTANT_COLOR	(R_c, G_c, B_c)	A_c
GL_ONE_MINUS_CONSTANT_COLOR	$(1,1,1) - (R_c, G_c, B_c)$	$1 - A_c$
GL_CONSTANT_ALPHA	(A_c, A_c, A_c)	A_c
GL_ONE_MINUS_CONSTANT_ALPHA	$(1,1,1) - (A_c, A_c, A_c)$	$1 - A_c$

GL_SRC_ALPHA_SATURATE

 (f, f, f) ^[*]

1

[*] Where $f = \min(A_s, 1 - A_d)$.

Remember that colors are represented by floating-point numbers, so adding them, subtracting them, and even multiplying them are all perfectly valid operations. [Table 6.1](#) may seem a bit bewildering, so let's look at a common blending function combination:

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

This function tells OpenGL to take the source (incoming) color and multiply the color (the RGB values) by the alpha value. Add this to the result of multiplying the destination color by one minus the alpha value from the source. Say, for example, that you have the color Red (1.0f, 0.0f, 0.0f, 0.0f) already in the color buffer. This is the destination color, or C_d . If something is drawn over this with the color blue and an alpha of 0.5 (0.0f, 0.0f, 1.0f, 0.5f), you would compute the final color as follows:

$$C_d = \text{destination color} = (1.0f, 0.0f, 0.0f, 0.0f)$$

$$C_s = \text{source color} = (0.0f, 0.0f, 1.0f, 0.5f)$$

$$S = \text{source alpha} = 0.5$$

$$D = \text{one minus source alpha} = 1.0 - 0.5 = 0.5$$

Now, the equation

$$C_f = (C_s * S) + (C_d * D)$$

evaluates to

$$C_f = (\text{Blue} * 0.5) + (\text{Red} * 0.5)$$

The final color is a scaled combination of the original red value with the incoming blue value. The higher the incoming alpha value, the more of the incoming color that is added and the less of the original color is retained.

This blending function is often used to achieve the effect of drawing a transparent object in front of some other opaque object. This technique does require, however, that you draw the background object or objects first and then draw the transparent object blended over the top. The effect can be quite dramatic. For example, in the [REFLECTION](#) sample program, we will use transparency to achieve the illusion of a reflection in a mirrored surface. We begin with a rotating torus with a sphere revolving around it, similar to the view in the preceding chapter's [Sphere World](#) example. Beneath the torus and sphere, we will place a reflective tiled floor. The output from this program is shown in [Figure 6.1](#), and the drawing code is shown in [Listing 6.1](#).

Listing 6.1. Rendering Function for the [REFLECTION](#) Program

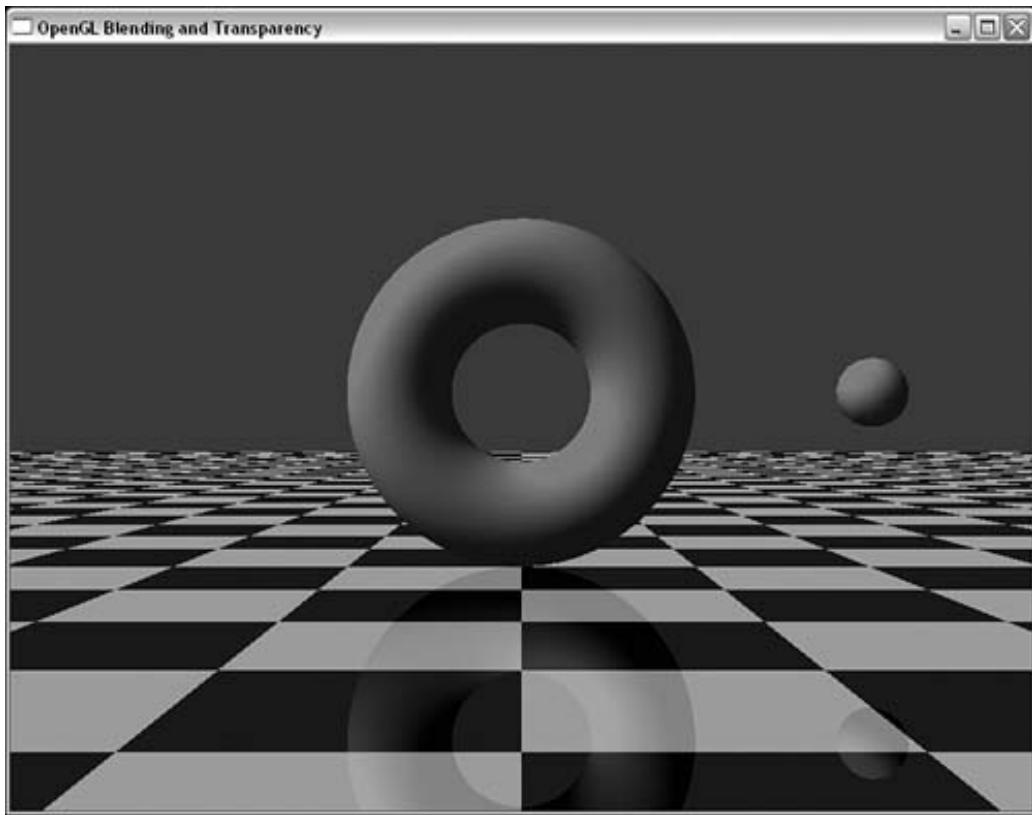
```
////////////////////////////////////////////////////////////////
// Called to draw scene
void RenderScene(void)
{
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
```

```

// Move light under floor to light the "reflected" world
glLightfv(GL_LIGHT0, GL_POSITION, fLightPosMirror);
glPushMatrix();
    glFrontFace(GL_CW);           // geometry is mirrored,
                                   // swap orientation
    glScalef(1.0f, -1.0f, 1.0f);
    DrawWorld();
    glFrontFace(GL_CCW);
glPopMatrix();
// Draw the ground transparently over the reflection
glDisable(GL_LIGHTING);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
DrawGround();
glDisable(GL_BLEND);
glEnable(GL_LIGHTING);
// Restore correct lighting and draw the world correctly
glLightfv(GL_LIGHT0, GL_POSITION, fLightPos);
DrawWorld();
glPopMatrix();
// Do the buffer Swap
glutSwapBuffers();
}

```

Figure 6.1. Using blending to create a fake reflection effect.



The basic algorithm for this effect is to draw the scene upside down first. We use one function to draw the scene, `DrawWorld()`, but to draw it upside down, we scale by -1 to invert the y-axis, reverse our polygon winding, and place the light down beneath us. After drawing the upside-down world, we draw the ground, but we use blending to create a transparent floor over the top of the inverted world. Finally, we turn off blending, put the light back overhead, and draw the world right side up.

Changing the Blending Equation

The blending equation we showed you earlier

$$C_f = (C_s * S) + (C_d * D)$$

is the *default* blending equation. You can actually choose from five different blending equations, each given in [Table 6.2](#) and selected with the following function:

```
void glBlendEquation(GLenum mode);
```

Table 6.2. Available Blend Equation Modes

Mode	Function
<code>GL_FUNC_ADD</code> (default)	$C_f = (C_s * S) + (C_d * D)$
<code>GL_FUNC_SUBTRACT</code>	$C_f = (C_s * S) - (C_d * D)$
<code>GL_FUNC_REVERSE_SUBTRACT</code>	$C_f = (C_d * D) - (C_s * S)$
<code>GL_MIN</code>	$C_f = \min(C_s, C_d)$
<code>GL_MAX</code>	$C_f = \max(C_s, C_d)$

In addition to `glBlendFunc`, you have even more flexibility with this function:

```
void glBlendFuncSeparate(GLenum srcRGB, GLenum dstRGB, GLenum srcAlpha,
                        GLenum dstAlpha);
```

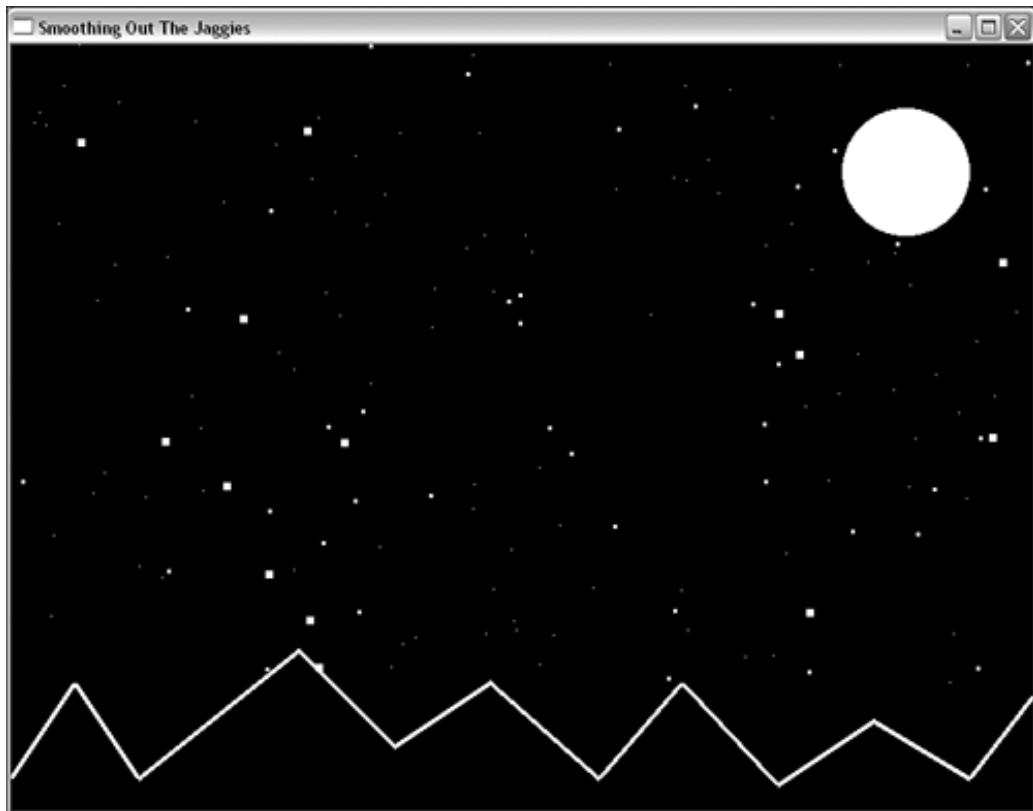
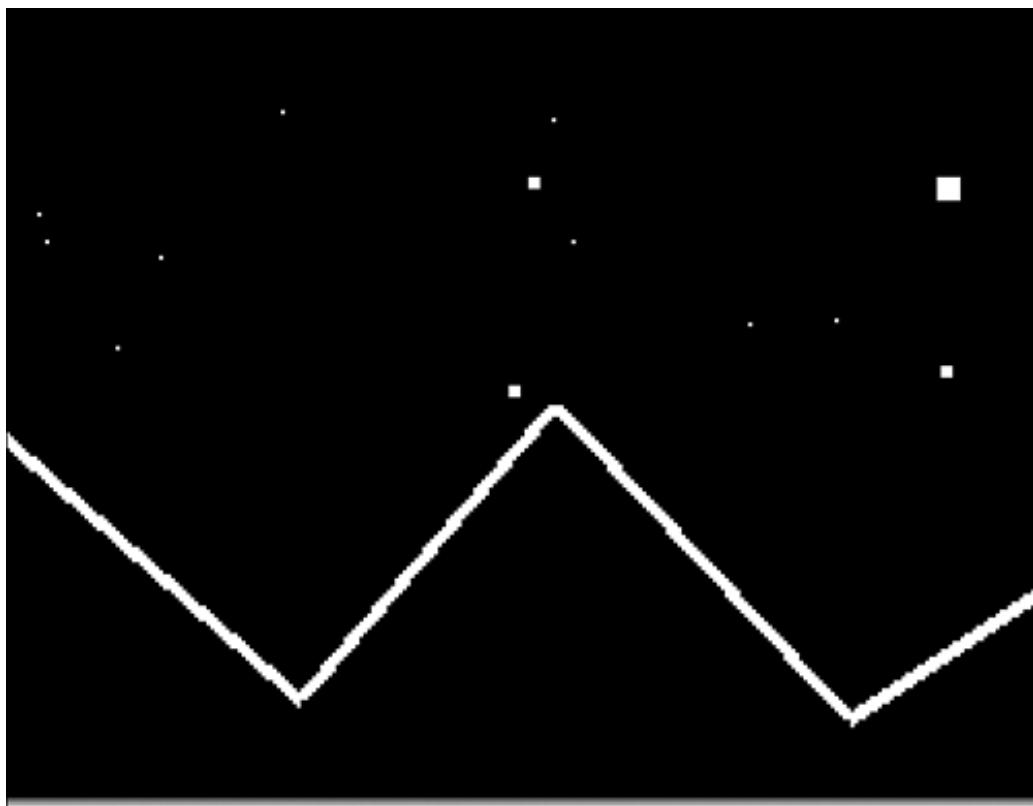
Whereas `glBlendFunc` specifies the blend functions for source and destination RGBA values, `glBlendFuncSeparate` allows you to specify blending functions for the RGB and alpha components separately.

Finally, as shown in [Table 6.1](#), the `GL_CONSTANT_COLOR`, `GL_ONE_MINUS_CONSTANT_COLOR`, `GL_CONSTANT_ALPHA`, and `GL_ONE_MINUS_CONSTANT_ALPHA` values all allow a constant blending color to be introduced to the blending equation. This constant blending color is initially black (0.0f, 0.0f, 0.0f), but can be changed with this function:

```
void glBlendColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);
```

Antialiasing

Another use for OpenGL's blending capabilities is antialiasing. Under most circumstances, individual rendered fragments are mapped to individual pixels on a computer screen. These pixels are square (or squarish), and usually you can spot the division between two colors quite clearly. These *jaggies*, as they are often called, catch the eye's attention and can destroy the illusion that the image is natural. These jaggies are a dead give-away that a computer-generated image is computer generated! For many rendering tasks, it is desirable to achieve as much realism as possible, particularly in games, simulations, or artistic endeavors. [Figure 6.2](#) shows the output for the sample program SMOOTHER. In [Figure 6.3](#), we have zoomed in on a line segment and some points to show the jagged edges.

Figure 6.2. Output from the program SMOOTH.**Figure 6.3. A closer look at some jaggies.**

To get rid of the jagged edges between primitives, OpenGL uses blending to blend the color of the fragment with the destination color of the pixel and its surrounding pixels. In essence, pixel colors are smeared slightly to neighboring pixels along the edges of any primitives.

Turning on antialiasing is simple. First, you must enable blending and set the blending function to be the same as you used in the preceding section for transparency:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

You also need to make sure the blend equation is set to `GL_ADD`, but because this is the default and most common blending equation, we don't show it here (changing the blending equation is also not supported on all OpenGL implementations). After blending is enabled and the proper blending function and equation are selected, you can choose to antialias points, lines, and/or polygons (any solid primitive) by calling `glEnable`:

```
glEnable(GL_POINT_SMOOTH);      // Smooth out points
glEnable(GL_LINE_SMOOTH);      // Smooth out lines
glEnable(GL_POLYGON_SMOOTH);   // Smooth out polygon edges
```

You should note, however, that `GL_POLYGON_SMOOTH` is not supported on all OpenGL implementations. [Listing 6.2](#) shows the code from the SMOOTHER program that responds to a pop-up menu that allows the user to switch between antialiased and non-antialiased rendering modes. When this program is run with antialiasing enabled, the points and lines appear smoother (fuzzier). In [Figure 6.4](#), a zoomed-in section shows the same area as [Figure 6.3](#), but now with the jagged edges somewhat reduced.

Listing 6.2. Switching Between Antialiased and Normal Rendering

```
///////////////////////////////
// Reset flags as appropriate in response to menu selections
void ProcessMenu(int value)
{
    switch(value)
    {
        case 1:
            // Turn on antialiasing, and give hint to do the best
            // job possible.
            glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
            glEnable(GL_BLEND);
            glEnable(GL_POINT_SMOOTH);
            glHint(GL_POINT_SMOOTH_HINT, GL_NICEST);
            glEnable(GL_LINE_SMOOTH);
            glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);
            glEnable(GL_POLYGON_SMOOTH);
            glHint(GL_POLYGON_SMOOTH_HINT, GL_NICEST);
            break;
        case 2:
            // Turn off blending and all smoothing
            glDisable(GL_BLEND);
            glDisable(GL_LINE_SMOOTH);
            glDisable(GL_POINT_SMOOTH);
            glDisable(GL_POLYGON_SMOOTH);
            break;
        default:
            break;
    }
    // Trigger a redraw
    glutPostRedisplay();
}
```

Figure 6.4. No more jaggies!



Note especially here the calls to the `glHint` function that was discussed in [Chapter 2](#), "Using OpenGL." There are many algorithms and approaches to achieve antialiased primitives. Any specific OpenGL implementation may choose any one of those approaches, and perhaps even support two! You can ask OpenGL if it does support multiple antialiasing algorithms to choose one that is very fast (`GL_FASTEST`) or the one with the most accuracy in appearance (`GL_NICEST`).

Multisample

One of the biggest advantages to antialiasing is that it smoothes out the edges of primitives and can lend a more natural and realistic appearance to renderings. Point and line smoothing is widely supported, but unfortunately polygon smoothing is not available on all platforms. Even when `GL_POLYGON_SMOOTH` is available, it is not as convenient a means of having your whole scene antialiased as you might think. Because it is based on the blending operation, you would need to sort all your primitives from *front to back!* Yuck.

A more recent addition to OpenGL to address this shortcoming is *multisampling*. When this feature is supported (it is an OpenGL 1.3 feature), an additional buffer is added to the framebuffer that includes the color, depth, and stencil values. All primitives are sampled multiple times per pixel, and the results are stored in this buffer. These samples are resolved to a single value each time the pixel is updated, so from the programmer's standpoint, it appears automatic and happens "behind the scenes." Naturally, this extra memory and processing that must take place are not without their performance penalties, and some implementations may not support multisampling for multiple rendering contexts.

To get multisampling, you must first obtain a rendering context that has support for a multisampled framebuffer. This varies from platform to platform, but GLUT exposes a bit field (`GLUT_MULTISAMPLE`) that allows you to request this until you reach the operating system-specific chapters later. For example, to request a multisampled, full-color, double-buffered frame buffer with depth, you would call

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH | GLUT_MULTISAMPLE);
```

You can turn multisampling on and off using the `glEnable/glDisable` combination and the `GL_MULTISAMPLE` token:

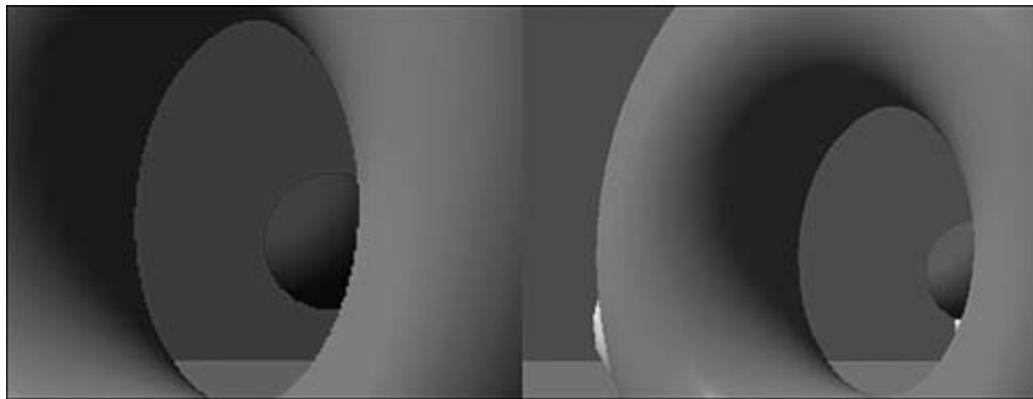
```
glEnable(GL_MULTISAMPLE);
```

or

```
glDisable(GL_MULTISAMPLE);
```

The sample program MULTISAMPLE is simply the Sphere World sample from the preceding chapter with multisampling selected and enabled. [Figure 6.5](#) shows the difference between two zoomed-in sections from each program. You can see that multisampling really helps smooth out the geometry's edges on the image to the right, lending to a much more pleasing appearance to the rendered output.

Figure 6.5. Zoomed-in view contrasting normal and multisampled rendering.



Another important note about multisampling is that when it is enabled, the point, line, and polygon smoothing features are ignored if enabled. This means you cannot use point and line smoothing at the same time as multisampling. On a given OpenGL implementation, points and lines may look better with smoothing turned on instead of multisampling. To accommodate this, you might turn off multisampling before drawing points and lines and then turn on multisampling for other solid geometry. The following pseudocode shows a rough outline of how to do this:

```
glDisable(GL_MULTISAMPLE);
glEnable(GL_POINT_SMOOTH);
// Draw some smooth points
// ...
glDisable(GL_POINT_SMOOTH);
glEnable(GL_MULTISAMPLE);
```

STATE SORTING

Turning different OpenGL features on and off changes the internal state of the driver. These state changes can be costly in terms of rendering performance. Frequently, performance-sensitive programmers will go to great lengths to sort all the drawing commands so that geometry needing the same state will be drawn together. This state sorting is one of the more common techniques to improve rendering speed in games.

The multisample buffers use the RGB values of fragments by default and do not include the alpha component of the colors. You can change this by calling `glEnable` with one of the following three values:

- `GL_SAMPLE_ALPHA_TO_COVERAGE`— Use the alpha value.
- `GL_SAMPLE_ALPHA_TO_ONE`— Set alpha to 1 and use it.

- **GL_SAMPLE_COVERAGE**— Use the value set with `glSampleCoverage`.

When `GL_SAMPLE_COVERAGE` is enabled, the `glSampleCoverage` function allows you to specify a specific value that is ANDed (bitwise) with the fragment coverage value:

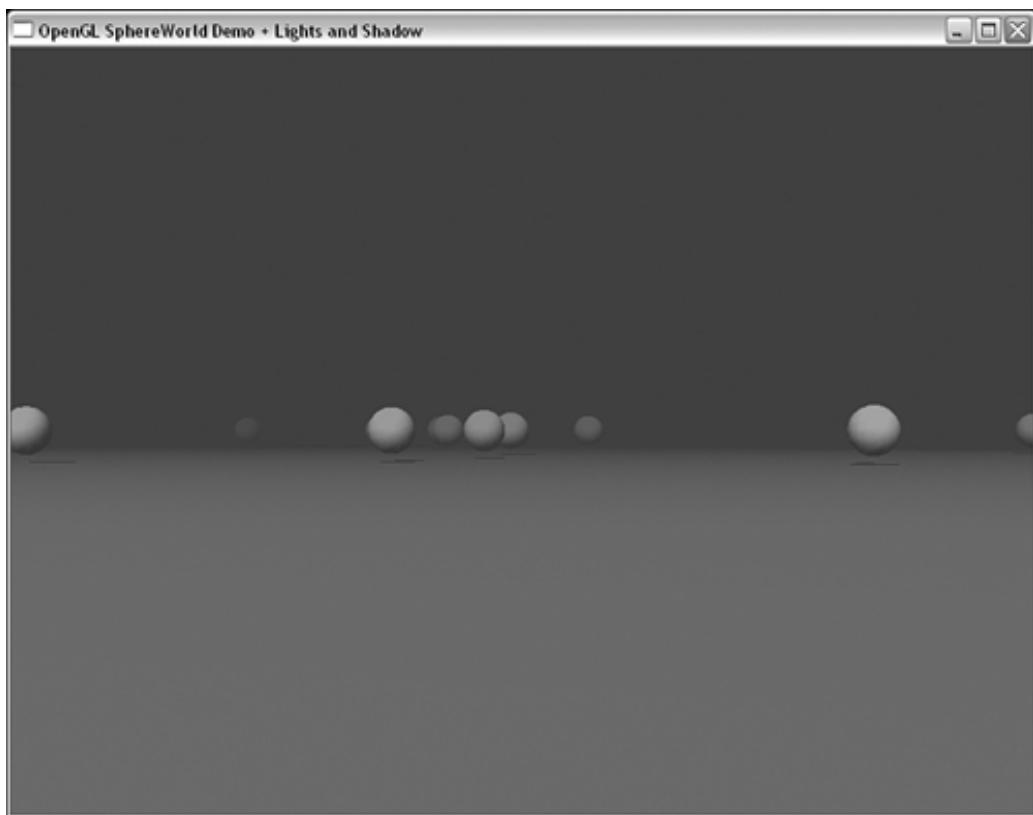
```
void glSampleCoverage(GLclampf value, GLboolean invert);
```

This fine-tuning of how the multisample operation works is not strictly specified by the specification, and the exact results may vary from implementation to implementation.

Fog

Another easy-to-use special effect that OpenGL supports is fog. With fog, OpenGL blends a fog color that you specify with geometry after all other color computations have been completed. The amount of the fog color mixed with the geometry varies with the distance of the geometry from the camera origin. The result is a 3D scene that appears to contain fog. Fog can be useful for slowly obscuring objects as they "disappear" into the background fog, or a slight amount of fog will produce a hazy effect on distant objects, providing a powerful and realistic depth cue. [Figure 6.6](#) shows output from the sample program FOGGED. As you can see, this is nothing more than the ubiquitous Sphere World example with fog turned on.

Figure 6.6. Sphere World with fog.



[Listing 6.3](#) shows the few lines of code added to the `SetupRC` function to produce this effect.

Listing 6.3. Setting Up Fog for Our Sphere World

```
// Grayish background
glClearColor(fLowLight[0], fLowLight[1], fLowLight[2], fLowLight[3]);
// Setup Fog parameters
glEnable(GL_FOG); // Turn Fog on
glFogfv(GL_FOG_COLOR, fLowLight); // Set fog color to match background
```

```
glFogf(GL_FOG_START, 5.0f);           // How far away does the fog start
glFogf(GL_FOG_END, 30.0f);           // How far away does the fog stop
glFogi(GL_FOG_MODE, GL_LINEAR);      // Which fog equation do I use?
```

Turning fog on and off is as easy as using the following functions:

```
glEnable/glDisable(GL_FOG);
```

The means of changing fog parameters (how the fog behaves) is to use the `glFog` function. There are several variations on `glFog`:

```
void glFogi(GLenum pname, GLint param);
void glFogf(GLenum pname, GLfloat param);
void glFogiv(GLenum pname, GLint* params);
void glFogfv(GLenum pname, GLfloat* params);
```

The first use of `glFog` shown here is

```
glFogfv(GL_FOG_COLOR, fLowLight); // Set fog color to match background
```

When used with the `GL_FOG_COLOR` parameter, this function expects a pointer to an array of floating-point values that specifies what color the fog should be. Here, we used the same color for the fog as the background clear color. If the fog color does not match the background (there is no strict requirement for this!), as objects become fogged, they will become a fog-colored silhouette against the background.

The next two lines allow us to specify how far away an object must be before fog is applied and how far away the object must be for the fog to be fully applied (object is fog color):

```
glFogf(GL_FOG_START, 5.0f);           // How far away does the fog start
glFogf(GL_FOG_END, 30.0f);           // How far away does the fog stop
```

The parameter `GL_FOG_START` specifies how far away from the eye fogging begins to take effect, and `GL_FOG_END` is the distance from the eye where the fog color completely overpowers the color of the object. The transition from start to end is controlled by the fog equation, which we set to `GL_LINEAR` here:

```
glFogi(GL_FOG_MODE, GL_LINEAR);       // Which fog equation do I use?
```

The fog equation calculates a fog factor that varies from 0 to 1 as the distance of the fragment moves between the start and end distances. OpenGL supports three fog equations: `GL_LINEAR`, `GL_EXP`, and `GL_EXP2`. These equations are shown in [Table 6.3](#).

Table 6.3. Three OpenGL Supported Fog Equations

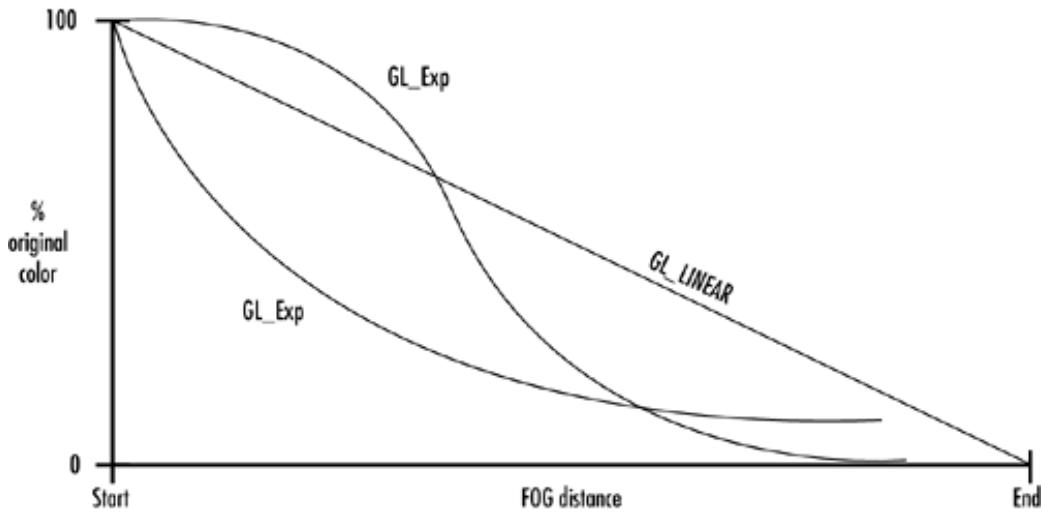
Fog Mode	Fog Equation
<code>GL_LINEAR</code>	$f = (\text{end} - c) / (\text{end} - \text{start})$
<code>GL_EXP</code>	$f = \exp(-d * c)$
<code>GL_EXP2</code>	$f = \exp(-(d * c)^2)$

In these equations, c is the distance of the fragment from the eye point, end is the `GL_FOG_END` distance, and start is the `GL_FOG_START` distance. The value d is the fog density. Fog density is typically set with `glFogf`:

```
glFogf(GL_FOG_DENSITY, 0.5f);
```

Figure 6.7 shows graphically how the fog equation and fog density parameters affect the transition from the original fragment color to the fog color. `GL_LINEAR` is a straight linear progression, whereas the `GL_EXP` and `GL_EXP2` equations show two characteristic curves for their transitions. The fog density value has no effect with linear fog (`GL_LINEAR`), but the other two curves you see here are generally pulled downward with increasing density values. These graphs, for example, show approximately a density value of 0.5.

Figure 6.7. Fog density equations.



The distance to a fragment from the eye point can be calculated in one of two ways. The first, `GL_FRAGMENT_DEPTH`, uses the depth value of the fragment itself and can be turned on with `glFog`, as shown here:

```
glFogi(GL_FOG_COORD_SRC, GL_FRAGMENT_DEPTH);
```

The second method interpolates the fog depth between vertices and is the default fog depth calculation method. This method can be a little faster but can result in a lower quality image. Like vertex-based lighting, the more geometry, the better the results will look. Once again, `glFog` allows you to set this mode specifically:

```
glFogi(GL_FOG_COORD_SRC, GL_FOG_COORD);
```

Accumulation Buffer

In addition to the color, stencil, and depth buffers, OpenGL supports what is called the *accumulation buffer*. This buffer allows you to render to the color buffer, and then instead of displaying the results in the window, copy the contents of the color buffer to the accumulation buffer. Several supported copy operations allow you to repeatedly blend in different ways the color buffer contents with the accumulated contents in the accumulation buffer (thus its name!). When you have finished accumulating an image, you can then copy the accumulation buffer back to the color buffer and display the results with a buffer swap.

The behavior of the accumulation buffer is controlled by one function:

```
void glAccum(GLenum op, GLfloat value);
```

The first parameter specifies which accumulation operation you want to use, and the second is a floating-point value that is used to scale the operation. [Table 6.4](#) lists the accumulation operations supported.

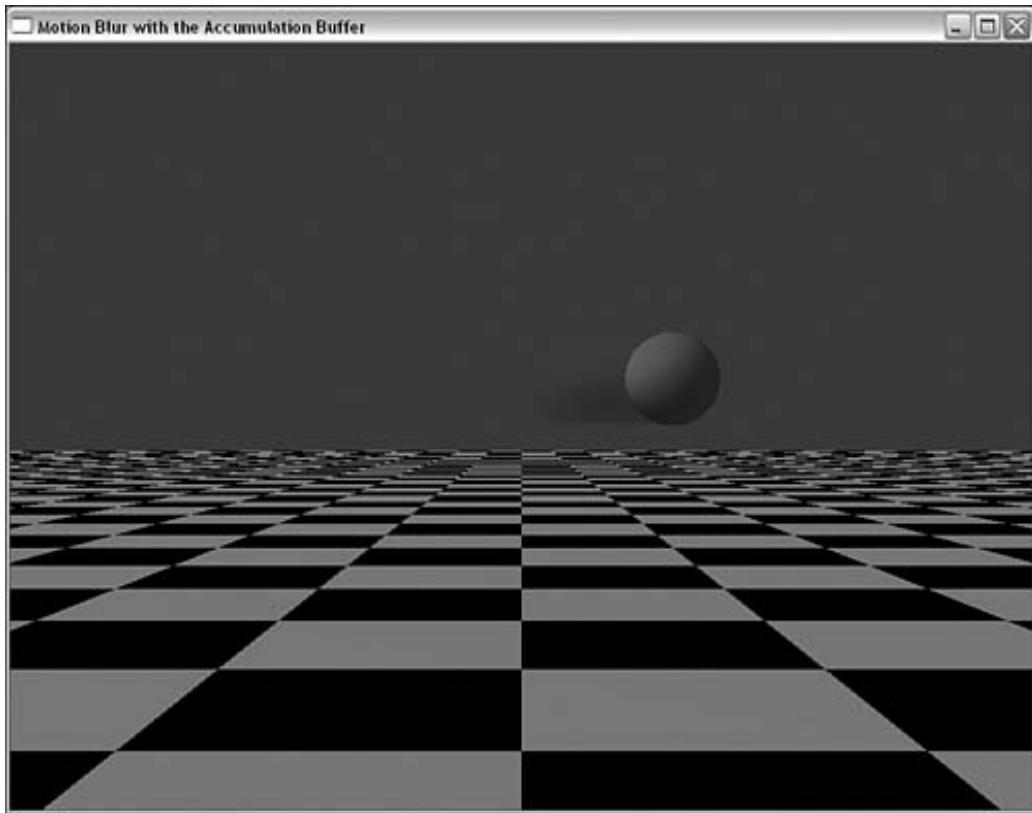
Table 6.4. OpenGL Accumulation Operations

Operation	Description
<code>GL_ACCUM</code>	Scales color buffer values by value and adds them to current contents of the accumulation buffer.
<code>GL_LOAD</code>	Scales color buffer values by value and replaces the current contents of the accumulation buffer.
<code>GL_RETURN</code>	Scales the color values from the accumulation buffer by value and then copies the values to the color buffer.
<code>GL_MULT</code>	Scales the color values in the accumulation buffer by value and stores the result in the accumulation buffer.
<code>GL_ADD</code>	Scales the color values in the accumulation buffer by value and adds the result to the current accumulation buffer contents.

Because of the large amount of memory that must be copied and processed for accumulation buffer operations, few real-time applications make use of this facility. For non-real-time rendering, OpenGL can produce some astonishing effects that you might not expect from a real-time API. For example, you can render a scene multiple times and move the point of view around by a fraction of a pixel each time. Accumulating these multiple rendering passes blurs the sharp edges and can produce an entire scene fully antialiased with a quality that surpasses anything that can be done with multisampling. You can also use this blurring effect to blur the background or foreground of an image and then render the object of focus clearly afterward, simulating some depth of field camera effects.

In our sample program MOTIONBLUR, we will demonstrate yet another use of the accumulation buffer to create what appears to be a motion blur effect. A moving sphere is drawn repeatedly in different positions. Each time it is drawn, it is accumulated to the accumulation buffer, with a smaller weight on subsequent passes. The result is a brighter red sphere with a trailing ghost-like image of itself following along behind. The output from this program is shown in [Figure 6.8](#).

Figure 6.8. Motion-blurred flying sphere.



Listing 6.4 shows the `DrawGeometry` function, which draws all the geometry of the scene. The `RenderScene` function then repeatedly calls this function and accumulates the results into the accumulation buffer. When finished, the lines

```
glAccum(GL_RETURN, 1.0f);
glutSwapBuffers();
```

copy the accumulation buffer back to the color buffer and perform the buffer swap.

Listing 6.4. Using the Accumulation Buffer for Motion Blur

```
///////////
// Draw the ground and the revolving sphere
void DrawGeometry(void)
{
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    DrawGround();
    // Place the moving sphere
    glColor3f(1.0f, 0.0f, 0.0f);
    glTranslatef(0.0f, 0.5f, -3.5f);
    glRotatef(-(yRot * 2.0f), 0.0f, 1.0f, 0.0f);
    glTranslatef(1.0f, 0.0f, 0.0f);
    glutSolidSphere(0.1f, 17, 9);
    glPopMatrix();
}
///////////
// Called to draw scene. The world is drawn multiple times with each
// frame blended with the last. The current rotation is advanced each
// time to create the illusion of motion blur.
void RenderScene(void)
{
    GLfloat fPass;
```

```

GLfloat fPasses = 10.0f;
// Set the current rotation back a few degrees
yRot = 35.0f;
for(fPass = 0.0f; fPass < fPasses; fPass += 1.0f)
{
    yRot += .75f; //1.0f / (fPass+1.0f);
    // Draw sphere
    DrawGeometry();
    // Accumulate to back buffer
    if(fPass == 0.0f)
        glAccum(GL_LOAD, 0.5f);
    else
        glAccum(GL_ACCUM, 0.5f * (1.0f / fPasses));
}
// copy accumulation buffer to color buffer and
// do the buffer Swap
glAccum(GL_RETURN, 1.0f);
glutSwapBuffers();
}

```

Finally, you must remember to ask for an accumulation buffer when you set up your OpenGL rendering context (see the OS-specific chapters for how to perform this task on your platform). GLUT also provides support for the accumulation buffer by passing the token `GLUT_ACCUM` to the `glutInitDisplayMode` function, as shown here:

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH | GLUT_ACCUM);
```

Other Color Operations

Blending is a powerful OpenGL feature that enables a myriad of special effects algorithms. Aside from direct support for blending, fog, and an accumulation buffer, OpenGL also supports some other means of tweaking color values and fragments as they are written to the color buffer.

Color Masking

After a final color is computed and is about to be written to the color buffer, OpenGL allows you to mask out one or more of the color channels with the `glColorMask` function:

```
void glColorMask(GLboolean red, GLboolean green, GLboolean blue,
                 GLboolean alpha);
```

The parameters are for the red, green, blue, and alpha channels, respectively. Passing `GL_TRUE` allows writing of this channel, and `GL_FALSE` prevents writing to this channel.

Color Logical Operations

Many 2D graphics APIs allow binary logical operations to be performed between the source and destination colors. OpenGL also supports these types of 2D operations with the `glLogicOp` function:

```
void glLogicOp(GLenum op);
```

The logical operation modes are listed in [Table 6.5](#). The logical operation is not enabled by default and is controlled, as most states are, with `glEnable/glDisable` using the value `GL_COLOR_LOGIC_OP`. For example, to turn on the logical operations, you use the following:

```
glEnable(GL_COLOR_LOGIC_OP);
```

Table 6.5. Bitwise Color Logical Operations

Argument Value	Operation
GL_CLEAR	0
GL_AND	$s \ \& \ d$
GL_AND_REVERSE	$s \ \& \ \sim d$
GL_COPY	s
GL_AND_INVERTED	$\sim s \ \& \ d$
NOOP	d
XOR	$s \ \text{xor} \ d$
OR	$s \ \ d$
NOR	$\sim(s \ \ d)$
GL_EQUIV	$\sim(s \ \text{xor} \ d)$
GL_INVERT	$\sim d$
GL_OR_REVERSE	$s \ \ \sim d$
GL_COPY_INVERTED	$\sim s$
GL_OR_INVERTED	$\sim s \ \ d$
GL_NAND	$\sim(s \ \& \ d)$
SET	all 1s

Alpha Testing

Alpha testing allows you to tell OpenGL to discard fragments whose alpha value fails the alpha comparison test. Discarded fragments are not written to the color, depth, stencil, or accumulation buffers. This feature allows you to improve performance by dropping values that otherwise might be written to the buffers and to eliminate geometry from the depth buffer that may not be visible in the color buffer (because of very low alpha values). The alpha test value and comparison function are specified with the `glAlphaFunc` function:

```
void glAlphaFunc(GLenum func, GLclampf ref);
```

The reference value is clamped to the range 0.0 to 1.0, and the comparison function may be specified by any of the constants in [Table 6.6](#). You can turn alpha testing on and off with `glEnable/glDisable` using the constant `GL_ALPHA_TEST`. The behavior of this function is similar to the `glDepthFunc` function covered in [Chapter 3](#).

Table 6.6. Alpha Test Comparison Functions

Constant	Comparison Function
<code>GL_NEVER</code>	Never passes
<code>GL_ALWAYS</code>	Always passes
<code>GL_LESS</code>	Passes if the fragment is less than the reference value
<code>GL_EQUAL</code>	Passes if the fragment is equal to the reference value
<code>GL_GREATER</code>	Passes if the fragment is greater than or equal to the reference value
<code>GL_NOTEQUAL</code>	Passes if the fragment is not equal to the reference value

Dithering

Dithering is a simple operation (in principle) that allows a display system with a small number of discrete colors to simulate displaying a much wider range of colors. For example, the color gray can be simulated by displaying a mix of white and black dots on the screen. More white than black dots make for a lighter gray, whereas more black dots make a darker gray. When your eye is far enough from the display, you cannot see the individual dots, and the blending effect creates the illusion of the color mix. This technique is useful for display systems that support only 8 or 16 bits of color information. Each OpenGL implementation is free to implement its own dithering algorithm, but the effect can be dramatically improved image quality on lower-end color systems. By default, dithering is turned on, and can be controlled with `glEnable/glDisable` and the constant `GL_DITHER`:

```
glEnable(GL_DITHER);      // Initially enabled
```

On higher-end display systems with greater color resolution, the implementation may not need dithering, and dithering may not be employed at a potentially considerable performance savings.

Summary

In this chapter, we took color beyond simple shading and lighting effects. You saw how to use blending to create transparent and reflective surfaces and create antialiased points, lines, and polygons with the blending and multisampling features of OpenGL. You also were introduced to the accumulation buffer and saw at least one common special effect that it is normally used for. Finally, you saw how OpenGL supports other color manipulation features such as color masks, bitwise color operations, and dithering, and how to use the alpha test to discard fragments altogether. Now we progress further in the next chapter from colors, shading, and blending to operations that incorporate real image data.

Included in the sample directory on the CD-ROM for this chapter, you'll find an update of the Sphere World example from [Chapter 5](#). You can study the source code to see how we have incorporated many of the techniques from this chapter to add some additional depth queuing to the world with fog, partially transparent shadows on the ground, and fully antialiased rendering of all geometry.

Reference

glAccum

Purpose: Operates on the accumulation buffer to establish pixel values.

Include File: `<GL/gl.h>`

Syntax:

```
void glAccum(GLenum op, GLfloat value);
```

Description: This function operates on the accumulation buffer. Except for `GL_RETURN`, color values are scaled by the `value` parameter and added or stored into the accumulation buffer. For `GL_RETURN`, the accumulation buffer's color values are scaled by the `value` parameter and stored in the current color buffer.

Parameters:

`op` `GLenum`: The accumulation function to apply. These functions are listed in [Table 6.4](#).

`value` `GLfloat`: The fractional amount of accumulation to perform.

Returns: None.

See Also: `glClearAccum`

glBlendColor

Purpose: Sets the constant blending color that is optionally used by the blending equation.

Include File: `<gl/gl.h>`

Syntax:

```
void glBlendcolor(GLclampf red, GLclampf green,
  GLclampf blue, GLclampf alpha);
```

Description: This function sets the blending color to be used when the blending equation is set to use the constant blending color for either the source or destination factors. These blending factors are `GL_CONSTANT_COLOR`, `GL_ONE_MINUS_CONSTANT_COLOR`, `GL_CONSTANT_ALPHA`, and `GL_ONE_MINUS_CONSTANT_ALPHA`.

Parameters:

`red` `GLclampf`: The constant blending color's red intensity.

`green` `GLclampf`: The constant blending color's green intensity.

`blue` `GLclampf`: The constant blending color's blue intensity.

`alpha` `GLclampf`: The constant blending color's alpha intensity.

Returns: None.

See Also: [glBlendEquation](#), [glBlendFunc](#), [glBlendFuncSeparate](#)

glBlendEquation

Purpose: Sets the blending equation to be used for color blending operations.

Include File: `<gl/gl.h>`

Syntax:

```
void glBlendEquation(GLenum mode);
```

Description: When blending is enabled, the source and destination colors are combined. The [glBlendFunc](#) function determines the weighting factors for these two colors, but this function selects which equation will be used to derive the new color value. The valid equations and their meanings are given in [Table 6.2](#). The default blending equation is `GL_FUNC_ADD`.

Parameters:

mode `GLenum`: One of the values specified in [Table 6.2](#).

Returns: None.

See Also: [glBlendColor](#), [glBlendFunc](#), [glBlendFuncSeparate](#)

glBlendFunc

Purpose: Sets color blending function's source and destination factors.

Include File: `<gl/gl.h>`

Syntax:

```
void glBlendFunc(GLenum sfactor, GLenum dfactor);
```

Description: This function sets the source and destination blending factors for color blending. You must call [glEnable\(GL_BLEND\)](#) to enable color blending. The default settings for blending are `glBlendFunc(GL_ONE, GL_ZERO)`. The list of valid blending factors is given in [Table 6.1](#).

Parameters:

sfactor `GLenum`: The source color's blending function.

dfactor `GLenum`: The destination color's blending function.

Returns: None.

See Also: [glBlendColor](#), [glBlendEquation](#), [glBlendFuncSeparate](#)

glBlendFuncSeparate

Purpose: Allows separate blending factors to be applied to the RGB color and alpha value.

Include File: `<gl/gl.h>`

Syntax:

```
void glBlendFuncSeparate(GLenum srcRGB, GLenum
➥ dstRGB, GLenum srcAlpha, GLenum dstAlpha);
```

Description: This function allows a separate weighting factor to be applied to the color (RGB) portion of a fragment and its alpha component. This applies to both source and destination color values. The list of valid blending factors is given in [Table 6.1](#).

Parameters:

`srcRGB` **GLenum:** The source's RGB blending factor.

`dstRGB` **GLenum:** The destination's RGB blending factor.

`srcAlpha` **GLenum:** The source's alpha blending factor.

`dstAlpha` **GLenum:** The destination's alpha blending factor.

Returns: None.

See Also: `glBlendColor`, `glBlendEquation`, `glBlendFunc`

glClearAccum

Purpose: Specifies the color values used to clear the accumulation buffer.

Include File: `<gl/gl.h>`

Syntax:

```
void glClearAccum(GLfloat red, GLfloat green,
➥ GLfloat blue, GLfloat alpha);
```

Description: This function specifies the color values to be used when clearing the accumulation buffer. The accumulation buffer is cleared by passing `GL_ACCUM_BUFFER_BIT` to the `glClear` function.

Parameters:

`red` **GLfloat:** Value of the red color component.

`green` **GLfloat:** Value of the green color component.

`blue` **GLfloat:** Value of the blue color component.

`alpha` **GLfloat:** Value of the alpha color component.

Returns: None.

See Also: `glAccum`, `glClear`

glColorMask**Purpose:** Masks or enables writing to the color buffer.**Include File:** `<gl/gl.h>`**Syntax:**

```
void glColorMask(GLboolean red, GLboolean green,
    GLboolean blue, GLboolean alpha);
```

Description: Writing to the color buffer may be masked out by setting the color mask with this function. Pass `GL_TRUE` for any color channel to allow writing, and `GL_FALSE` to prevent changes or writes.**Parameters:**

`red` `GLboolean`: Enable or disable writes to the red component of the color buffer.
`green` `GLboolean`: Enable or disable writes to the green component of the color buffer.
`blue` `GLboolean`: Enable or disable writes to the blue component of the color buffer.
`alpha` `GLboolean`: Enable or disable writes to the alpha component of the color buffer.

Returns: None.**See Also:** `glDepthMask`, `glLogicOp`, `glStencilMask`**glFog****Purpose:** Controls the behavior of fog.**Include File:** `<gl/gl.h>`**Syntax:**

```
void glFogf(GLenum pname, GLfloat param);
void glFogfv(GLenum pname, GLfloat *params);
void glFogi(GLenum pname, GLint param);
void glFogiv(GLenum pname, GLint *params);
```

Description: The `glFog` functions set the various fog parameters. To render using fog, you must enable fog with `glEnable(GL_FOG)`.**Parameters:**

pname **GLenum**: The parameter to set. Valid values are as follows:

GL_FOG_COLOR: Specifies the fog color as an array of four floats (RGBA).

GL_FOG_COORD_SRC: Determines how the fog coordinates are calculated. Must be either **GL_FOG_COORD** (vertex-interpolated fog) or **GL_FOG_FRAGMENT_DEPTH** (using the fragment's depth value).

GL_FOG_DENSITY: Specifies the fog density.

GL_FOG_END: Specifies the maximum distance at which fog is applied.

GL_FOG_MODE: Specifies the fog mode. Must be either **GL_LINEAR**, **GL_EXP**, or **GL_EXP2**.

GL_FOG_START: Specifies the distance at which fog begins to be applied.

param **GLfloat**, **GLint**: The parameter value.

params **GLfloat ***, **GLint ***: A pointer to the parameter array as either floats or ints.

Returns: None.

See Also: [glEnable](#)

glLogicOp

Purpose: Selects a logical operation to be performed on color writes.

Include File: `<gl/gl.h>`

Syntax:

```
void glLogicOp(GLenum op);
```

Description: This function sets a bitwise logical operation to be performed between an incoming color value (source) and the existing (destination) color in the color buffer. By default, the logic operation is disabled and must be turned on with [glEnable\(GL_COLOR_LOGIC_OP\)](#).

Parameters:

op **GLenum**: Specifies the logical operation to be performed. Any constant from [Table 6.5](#) may be used.

Returns: None.

See Also: [glColorMask](#)

glSampleCoverage

Purpose: Sets how alpha values are interpreted when computing multisampling coverage.

Include File: <gl/gl.h>**Syntax:**

```
void glSampleCoverage(GLclampf value, GLboolean
→ invert);
```

Description: Normally, the multisampling operation uses only RGB values to calculate fragment coverage. However, if you enable either `GL_SAMPLE_ALPHA_TO_ONE`, `GL_SAMPLE_ALPHA_TO_COVERAGE`, or `GL_SAMPLE_COVERAGE`, OpenGL will use the alpha component for coverage calculations. This function is used to set a value that is ANDed (or inverted and ANDed) with the fragment coverage value when either `GL_SAMPLE_ALPHA_TO_COVERAGE` or `GL_SAMPLE_COVERAGE` is enabled.

Parameters:

`value` `GLclampf`: Temporary coverage value used if `GL_ALPHA_TO_COVERAGE` or `GL_SAMPLE_COVERAGE` has been enabled.

`invert` `GLboolean`: Set to true to indicate the temporary coverage value should be bitwise inverted before it is used.

Returns: None.**See Also:** `glutInitDisplayMode`

Chapter 7. Imaging with OpenGL

by Richard S. Wright, Jr.

WHAT YOU'LL LEARN IN THIS CHAPTER:

How To	Functions You'll Use
Set the raster position	<code>glRasterPos</code> , <code>glWindowPos</code>
Draw bitmaps	<code>glBitmap</code>
Read and write color images	<code>glReadPixels</code> , <code>glDrawPixels</code>
Magnify, shrink, and flip images	<code>glPixelZoom</code>
Set up operations on colors	<code>glPixelTransfer</code> , <code>glPixelMap</code>
Perform color substitutions	<code>glColorTable</code>
Perform advanced image filtering	<code>glConvolutionFilter2D</code>
Collect statistics on images	<code>glHistogram</code> , <code>glGetHistogram</code>

In the preceding chapters, you learned the basics of OpenGL's acclaimed 3D graphics capabilities. Until now, all output has been the result of three-dimensional primitives being transformed and projected to 2D space and finally rasterized into the color buffer. However, OpenGL also supports reading and writing directly from and to the color buffer. This means image data can be read directly from the color buffer into your own memory buffer where it can be manipulated or written to a file. This also means you can derive or read image data from a file and place it directly into the color buffer yourself. OpenGL goes beyond merely reading and writing 2D images and has

support for a number of imaging operations that can be applied automatically during reading and writing operations. This chapter is all about OpenGL's rich but sometimes overlooked 2D capabilities.

Bitmaps

In the beginning, there were bitmaps. And they were...good enough. The original electronic computer displays were monochrome (one color), typically green or amber, and every pixel on the screen had one of two states: on or off. Computer graphics were simple in the early days, and image data was represented by *bitmaps*—a series of ones and zeros representing on and off pixel values. In a bitmap, each bit in a block of memory corresponds to exactly one pixel's state on the screen. We introduced this idea in the "[Filling Polygons, or Stippling Revisited](#)" section in [Chapter 3](#), "Drawing in Space: Geometric Primitives and Buffers." Bitmaps can be used for masks (polygon stippling), fonts and character shapes, and even two-color dithered images. [Figure 7.1](#) shows an image of a horse represented as a bitmap. Even though only two colors are used (black and white dots), the representation of a horse is still apparent. Compare this image with the one in [Figure 7.2](#), which shows a grayscale image of the same horse. In this *pixelmap*, each pixel has one of 256 different intensities of gray. We discuss pixelmaps further in the next section. The term *bitmap* is often applied to images that contain grayscale or full color data. This description is especially common on the Windows platform, and many would argue that, strictly speaking, this is a misapplication of the term. In this book, we use the term *bitmap* to mean a true binary map of on and off values, and we use the term *pixelmap* (or frequently *pixmap* for short) for image data that contains color or intensity values for each pixel.

Figure 7.1. A bitmapped image of a horse.

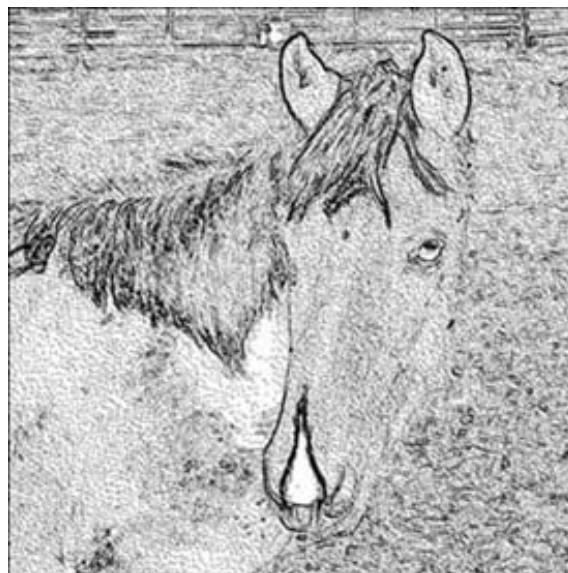


Figure 7.2. A pixmap image of a horse.



A Bitmapped Sample

The sample program BITMAPS is shown in [Listing 7.1](#). This program uses the same bitmap data used in [Chapter 3](#) for the polygon stippling sample that represents the shape of a small campfire arranged as a pattern of bits measuring 32x32. Remember that bitmaps are built from the bottom up, which means the first row of data actually represents the bottom row of the bitmapped image. This program creates a 512x512 window and fills the window with 16 rows and columns of the campfire bitmap. The output is shown in [Figure 7.3](#). Note that the `ChangeSize` function sets an orthographic projection matching the window's width and height in pixels.

Listing 7.1. The BITMAPS Sample Program

```
#include "../../Common/OpenGLSB.h"      // System and OpenGL Stuff
#include "../../Common/GLTools.h"        // OpenGL toolkit
// Bitmap of camp fire
GLubyte fire[128] = { 0x00, 0x00, 0x00, 0x00,
                      0x00, 0x00, 0x00, 0xc0,
                      0x00, 0x00, 0x01, 0xf0,
                      0x00, 0x00, 0x07, 0xf0,
                      0x0f, 0x00, 0x1f, 0xe0,
                      0x1f, 0x80, 0x1f, 0xc0,
                      0x0f, 0xc0, 0x3f, 0x80,
                      0x07, 0xe0, 0x7e, 0x00,
                      0x03, 0xf0, 0xff, 0x80,
                      0x03, 0xf5, 0xff, 0xe0,
                      0x07, 0xfd, 0xff, 0xf8,
                      0x1f, 0xfc, 0xff, 0xe8,
                      0xff, 0xe3, 0xbf, 0x70,
                      0xde, 0x80, 0xb7, 0x00,
                      0x71, 0x10, 0x4a, 0x80,
                      0x03, 0x10, 0x4e, 0x40,
                      0x02, 0x88, 0x8c, 0x20,
                      0x05, 0x05, 0x04, 0x40,
                      0x02, 0x82, 0x14, 0x40,
                      0x02, 0x40, 0x10, 0x80,
                      0x02, 0x64, 0x1a, 0x80,
```

```

0x00, 0x92, 0x29, 0x00,
0x00, 0xb0, 0x48, 0x00,
0x00, 0xc8, 0x90, 0x00,
0x00, 0x85, 0x10, 0x00,
0x00, 0x03, 0x00, 0x00,
0x00, 0x00, 0x10, 0x00 };

///////////////////////////////
// This function does any needed initialization on the rendering
// context.
void SetupRC()
{
    //
    // Black background
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
}

///////////////////////////////
// Set coordinate system to match window coordinates
void ChangeSize(int w, int h)
{
    //
    // Prevent a divide by zero, when window is too short
    // (you can't make a window of zero width).
    if(h == 0)
        h = 1;
    glViewport(0, 0, w, h);
    // Reset the coordinate system before modifying
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    // Pseudo window coordinates
    gluOrtho2D(0.0, (GLfloat) w, 0.0f, (GLfloat) h);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

///////////////////////////////
// Called to draw scene
void RenderScene(void)
{
    int x, y;
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT);
    // Set color to white
    glColor3f(1.0f, 1.0f, 1.0f);
    // Loop through 16 rows and columns
    for(y = 0; y < 16; y++)
    {
        //
        // Set raster position for this "square"
        glRasterPos2i(0, y * 32);
        for(x = 0; x < 16; x++)
            // Draw the "fire" bitmap, advance raster position
            glBitmap(32, 32, 0.0, 0.0, 32.0, 0.0, fire);
    }
    // Do the buffer Swap
    glutSwapBuffers();
}

///////////////////////////////
// Main program entrypoint
int main(int argc, char* argv[])
{
    //
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
    glutInitWindowSize(512, 512);
    glutCreateWindow("OpenGL Bitmaps");
}

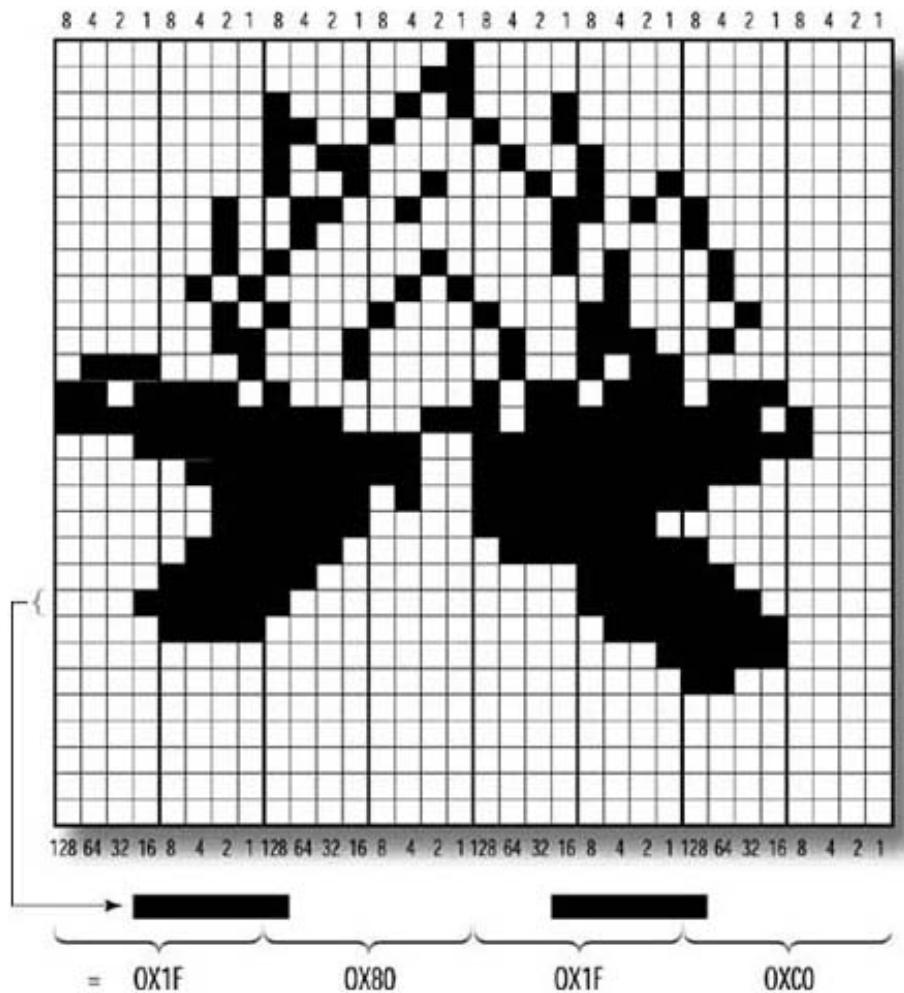
```

```

glutReshapeFunc(ChangeSize);
glutDisplayFunc(RenderScene);
SetupRC();
glutMainLoop();
return 0;
}

```

Figure 7.3. The 16 rows and columns of the campfire bitmap.



Setting the Raster Position

The real meat of the BITMAPS sample program occurs in the `RenderScene` function where a set of nested loops draws 16 rows of 16 columns of the campfire bitmap:

```

// Loop through 16 rows and columns
for(y = 0; y < 16; y++)
{
    // Set raster position for this "square"
    glRasterPos2i(0, y * 32);
    for(x = 0; x < 16; x++)
        // Draw the "fire" bitmap, advance raster position
        glBitmap(32, 32, 0.0, 0.0, 32.0, 0.0, fire);
}

```

The first loop (`y` variable) steps the row from 0 to 16. The following function call sets the raster position to the place where you want the bitmap drawn:

```
glRasterPos2i(0, y * 32);
```

The raster position is interpreted much like a call to `glVertex` in that the coordinates are transformed by the current modelview and projection matrices. The resulting window position becomes the current raster position. All rasterizing operations (bitmaps and pixmaps) occur with the current raster position specifying the image's lower-left corner. If the current raster position falls outside the window's viewport, it is invalid, and any OpenGL operations that require the raster position will fail.

In this example, we deliberately set the OpenGL projection to match the window dimensions so that we could use window coordinates to place the bitmaps. However, this technique may not always be convenient, so OpenGL provides an alternative function that allows you to set the raster position in window coordinates without regard to the current transformation matrix or projection:

```
void glWindowPos2i(GLint x, GLint y);
```

The `glWindowPos` function comes in two-and three-argument flavors and accepts integers, floats, doubles, and short arguments much like `glVertex`. See the reference section for a complete breakdown.

One important note about the raster position is that the color of the bitmap is set when either `glRasterPos` or `glWindowPos` is called. This means that the current color *previously* set with `glColor` is bound to subsequent bitmap operations. Calls to `glColor` made after the raster position is set will have no effect on the bitmap color.

Drawing the Bitmap

Finally, we get to the command that actually draws the bitmap into the color buffer:

```
glBitmap(32, 32, 0.0, 0.0, 32.0, 0.0, fire);
```

The `glBitmap` function copies the supplied bitmap to the color buffer at the current raster position and optionally advances the raster position all in one operation. This function has the following syntax:

```
void glBitmap(GLsize width, GLsize height, GLfloat xorig, GLfloat yorig,
             GLfloat xmove, GLfloat ymove, GLubyte *bitmap);
```

The first two parameters, `width` and `height`, specify the width and height of the bitmap (in bits). The next two parameters, `xorig` and `yorig`, specify the floating-point origin of the bitmap. To begin at the lower-left corner of the bitmap, specify 0.0 for both of these arguments. Then `xmove` and `ymove` specify an offset in pixels to move the raster position in the x and y directions after the bitmap is rendered. Note that these four parameters are all in floating-point units. The final argument, `bitmap`, is simply a pointer to the bitmap data. Note that when a bitmap is drawn, only the 1s in the image create fragments in the color buffer; 0s have no effect on anything already present.

Pixel Packing

Bitmaps and pixmaps are rarely packed tightly into memory. On many hardware platforms, each row of a bitmap or pixmap should begin on some particular byte-aligned address for performance reasons. Most compilers automatically put variables and buffers at an address alignment optimal for that architecture. OpenGL, by default, assumes a 4-byte alignment, which is appropriate for many systems in use today. The campfire bitmap used in the preceding example was tightly packed, but it didn't cause problems because the bitmap *just happened* to also be 4-byte aligned. Recall that the bitmap was 32 bits wide, exactly 4 bytes. If we had used a 34-bit wide bitmap (only two more bits), we would have had to pad each row with an extra 30 bits of unused storage, for a total of 64 bits (8 bytes is evenly divisible by 4). Although this may seem like a waste of memory, this arrangement allows most CPUs to more efficiently grab blocks of data (such as a row of bits for a bitmap).

You can change how pixels for bitmaps or pixmaps are stored and retrieved by using the following functions:

```
void glPixelStorei(GLenum pname, GLint param);
void glPixelStoref(GLenum pname, GLfloat param);
```

If you want to change to tightly packed pixel data, for example, you make the following function call:

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

`GL_UNPACK_ALIGNMENT` specifies how image data OpenGL will unpack from the data buffer. Likewise, you can use `GL_PACK_ALIGNMENT` to tell OpenGL how to pack data being read from the color buffer and placed in a user-specified memory buffer. The complete list of pixel storage modes available through this function is given in [Table 7.1](#) and explained in more detail in the reference section.

Table 7.1. `glPixelStore` Parameters

Parameter Name	Type	Initial Value
<code>GL_PACK_SWAP_BYTES</code>	GLboolean	<code>GL_FALSE</code>
<code>GL_UNPACK_SWAP_BYTES</code>	GLboolean	<code>GL_FALSE</code>
<code>GL_PACK_LSB_FIRST</code>	GLboolean	<code>GL_FALSE</code>
<code>GL_UNPACK_LSB_FIRST</code>	GLboolean	<code>GL_FALSE</code>
<code>GL_PACK_ROW_LENGTH</code>	GLint	0
<code>GL_UNPACK_ROW_LENGTH</code>	GLint	0
<code>GL_PACK_SKIP_ROWS</code>	GLint	0
<code>GL_UNPACK_SKIP_ROWS</code>	GLint	0
<code>GL_PACK_SKIP_PIXELS</code>	GLint	0
<code>GL_UNPACK_SKIP_PIXELS</code>	GLint	0
<code>GL_PACK_ALIGNMENT</code>	GLint	4
<code>GL_UNPACK_ALIGNMENT</code>	GLint	4
<code>GL_PACK_IMAGE_HEIGHT</code>	GLint	0

<code>GL_UNPACK_IMAGE_HEIGHT</code>	GLint	0
<code>GL_PACK_SKIP_IMAGES</code>	GLint	0
<code>GL_UNPACK_SKIP_IMAGES</code>	Glint	0

Pixmaps

Of more interest and somewhat greater utility on today's full-color computer systems are pixmaps. A pixmap is similar in memory layout to a bitmap; however, each pixel may be represented by more than one bit of storage. Extra bits of storage for each pixel allow either intensity (sometimes referred to as *luminance* values) or color component values to be stored. You draw pixmaps at the current raster position just like bitmaps, but you draw them using a new function:

```
void glDrawPixels(GLsizei width, GLsizei height, GLenum format,
                  GLenum type, const void *pixels);
```

The first two arguments specify the width and height of the image in pixels. The third argument specifies the format of the image data, followed by the data type of the data and finally a pointer to the data itself. Unlike `glBitmap`, this function does not update the raster position and is considerably more flexible in the way you can specify image data.

Each pixel is represented by one or more data elements contained at the `*pixels` pointer. The color layout of these data elements is specified by the `format` parameter using one of the constants listed in [Table 7.2](#).

Table 7.2. OpenGL Pixel Formats

Constant	Description
<code>GL_RGB</code>	Colors are in red, green, blue order.
<code>GL_RGBA</code>	Colors are in red, green, blue, alpha order.
<code>GL_BGR/GL_BGR_EXT</code>	Colors are in blue, green, red order.
<code>GL_BGRA/GL_BGRA_EXT</code>	Colors are in blue, green, red, alpha order.
<code>GL_RED</code>	Each pixel contains a single red component.
<code>GL_GREEN</code>	Each pixel contains a single green component.
<code>GL_BLUE</code>	Each pixel contains a single blue component.
<code>GL_ALPHA</code>	Each pixel contains a single alpha component.
<code>GL_LUMINANCE</code>	Each pixel contains a single luminance (intensity) component.
<code>GL_LUMINANCE_ALPHA</code>	Each pixel contains a luminance followed by an alpha component.
<code>GL_STENCIL_INDEX</code>	Each pixel contains a single stencil value.
<code>GL_DEPTH_COMPONENT</code>	Each pixel contains a single depth value.

Two of the formats, `GL_STENCIL_INDEX` and `GL_DEPTH_COMPONENT`, are used for reading and writing directly to the stencil and depth buffers. The `type` parameter interprets the data pointed to by the `*pixels` parameter. It tells OpenGL what data type within the buffer is used to store the

color components. The recognized values are specified in [Table 7.3](#).

Table 7.3. Data Types for Pixel Data

Constant	Description
<code>GL_UNSIGNED_BYTE</code>	Each color component is an 8-bit unsigned integer
<code>GL_BYTE</code>	Signed 8-bit integer
<code>GL_BITMAP</code>	Single bits, no color data; same as <code>glBitmap</code>
<code>GL_UNSIGNED_SHORT</code>	Unsigned 16-bit integer
<code>GL_SHORT</code>	Signed 16-bit integer
<code>GL_UNSIGNED_INT</code>	Unsigned 32-bit integer
<code>GL_INT</code>	Signed 32-bit integer
<code>GL_FLOAT</code>	Single precision float
<code>GL_UNSIGNED_BYTE_3_2_2</code>	Packed RGB values
<code>GL_UNSIGNED_BYTE_2_3_3_REV</code>	Packed RGB values
<code>GL_UNSIGNED_SHORT_5_6_5</code>	Packed RGB values
<code>GL_UNSIGNED_SHORT_5_6_5_REV</code>	Packed RGB values
<code>GL_UNSIGNED_SHORT_4_4_4_4</code>	Packed RGBA values
<code>GL_UNSIGNED_SHORT_4_4_4_4_REV</code>	Packed RGBA values
<code>GL_UNSIGNED_SHORT_5_5_5_1</code>	Packed RGBA values
<code>GL_UNSIGNED_SHORT_1_5_5_5_REV</code>	Packed RGBA values
<code>GL_UNSIGNED_INT_8_8_8</code>	Packed RGBA values
<code>GL_UNSIGNED_INT_8_8_8_8_REV</code>	Packed RGBA values
<code>GL_UNSIGNED_INT_10_10_10_2</code>	Packed RGBA values
<code>GL_UNSIGNED_INT_2_10_10_10_REV</code>	Packed RGBA values

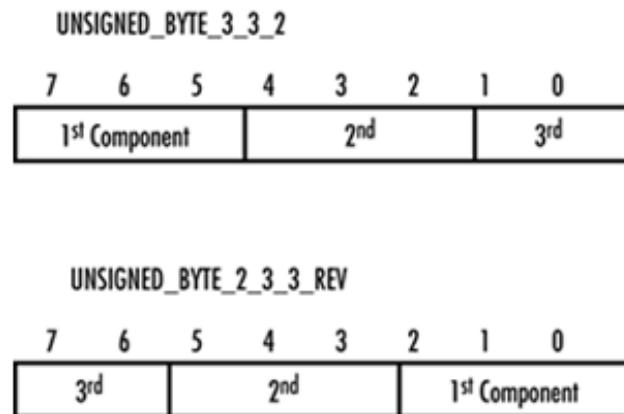
Packed Pixel Formats

The packed formats listed in [Table 7.3](#) were introduced in OpenGL 1.2 as a means of allowing image data to be stored in a more compressed form that matched a range of color graphics hardware. Display hardware designs could save memory or operate faster on a smaller set of packed pixel data. These packed pixel formats are still found on some PC hardware and may continue to be useful for future hardware platforms.

The packed pixel formats compress color data into as few bits as possible, with the number of bits per color channel shown in the constant. For example, the `GL_UNSIGNED_BYTE_3_3_2` format stores three bits of the first component, three bits of the second component, and two bits of the third component. Remember, the specific components (red, green, blue, and alpha) are still ordered according to the `format` parameter of `glDrawPixels`. The components are ordered from the highest bits (most significant bit or MSB) to the lowest (least significant bit or LSB). `GL_UNSIGNED_BYTE_2_3_3_REV` reverses this order and places the last component in the top two bits, and so on. [Figure 7.4](#) shows graphically the bitwise layout for these two arrangements. All

the other packed formats are interpreted in the same manner.

Figure 7.4. Sample layout for two packed pixel formats.



A More Colorful Example

Now it's time to put your new pixel knowledge to work with a more colorful and realistic rendition of a campfire. [Figure 7.5](#) shows the output of the next sample program, IMAGELOAD. This program loads an image, `fire.tga`, and uses `glDrawPixels` to place the image directly into the color buffer. This program is almost identical to the BITMAPS sample program with the exception that the color image data is read from a targa image file (note the `.tga` file extension) using the `glTools` function `gltLoadTGA` and then drawn with a call to `glDrawPixels` instead of `glBitmap`. The function that loads the file and displays it is shown in [Listing 7.2](#).

Listing 7.2. The `RenderScene` Function to Load and Display the Image File

```
// Called to draw scene
void RenderScene(void)
{
    GLubyte *pImage = NULL;
    GLint iWidth, iHeight, iComponents;
    GLenum eFormat;
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT);
    // Targas are 1 byte aligned
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    // Load the TGA file, get width, height, and component/format information
    pImage = gltLoadTGA("fire.tga", &iWidth, &iHeight, &iComponents, &eFormat);
    // Use Window coordinates to set raster position
    glRasterPos2i(0, 0);
    // Draw the pixmap
    if(pImage != NULL)
        glDrawPixels(iWidth, iHeight, eFormat, GL_UNSIGNED_BYTE, pImage);
    // Don't need the image data anymore
    free(pImage);
    // Do the buffer Swap
    glutSwapBuffers();
}
```

Figure 7.5. A campfire image loaded from a file.



Note the call that reads the targa file:

```
// Load the TGA file, get width, height, and component/format information
pImage = gltLoadTGA("fire.tga", &iWidth, &iHeight, &iComponents, &eFormat);
```

We use this function frequently in other sample programs when the need arises to load image data from a file. The first argument is the filename (with the path if necessary) of the targa file to load. The targa image format is a well-supported and common image file format. Unlike JPEG files, targa files (usually) store an image in its uncompressed form. The `gltLoadTGA` function opens the file and then reads in and parses the header to determine the width, height, and data format of the file. The number of components can be one, three, or four for luminance, RGB, or RGBA images, respectively. The final parameter is a pointer to a `GLenum` that receives the corresponding OpenGL image format for the file. If the function call is successful, it returns a newly allocated pointer to the image data read directly from the file. If the file is not found, or some other error occurs, the function returns `NULL`. The complete listing for the `gltLoadTGA` function is given in [Listing 7.3](#).

Listing 7.3. The `gltLoadTGA` Function to Load Targa Files for Use in OpenGL

```
///////////////////////////////
// Allocate memory and load targa bits. Returns pointer to new buffer,
// height, and width of texture, and the OpenGL format of data.
// Call free() on buffer when finished!
// This only works on pretty vanilla targas... 8, 24, or 32 bit color
// only, no palettes, no RLE encoding.
GLbyte *gltLoadTGA(const char *szFileName,
                    GLint *iWidth, GLint *iHeight,
                    GLint *iComponents, GLenum *eFormat)
{
    FILE *pFile;                      // File pointer
    TGAHEADER tgaHeader;              // TGA file header
    unsigned long lImageSize;         // Size in bytes of image
    short sDepth;                    // Pixel depth;
    GLbyte *pBits = NULL;            // Pointer to bits
    // Default/Failed values
    *iWidth = 0;
    *iHeight = 0;
    *eFormat = GL_BGR_EXT;
```

```

*iComponents = GL_RGB8;
// Attempt to open the file
pFile = fopen(szFileName, "rb");
if(pFile == NULL)
    return NULL;
// Read in header (binary)
fread(&tgaHeader, 18/* sizeof(TGAHEADER)*/, 1, pFile);
// Do byte swap for big vs little endian
#ifndef __APPLE__
    BYTE_SWAP(tgaHeader.colorMapStart);
    BYTE_SWAP(tgaHeader.colorMapLength);
    BYTE_SWAP(tgaHeader.xstart);
    BYTE_SWAP(tgaHeader.ystart);
    BYTE_SWAP(tgaHeader.width);
    BYTE_SWAP(tgaHeader.height);
#endif
// Get width, height, and depth of texture
*iWidth = tgaHeader.width;
*iHeight = tgaHeader.height;
sDepth = tgaHeader.bits / 8;
// Put some validity checks here. Very simply, I only understand
// or care about 8, 24, or 32 bit targas.
if(tgaHeader.bits != 8 && tgaHeader.bits != 24 && tgaHeader.bits != 32)
    return NULL;
// Calculate size of image buffer
lImageSize = tgaHeader.width * tgaHeader.height * sDepth;
// Allocate memory and check for success
pBits = malloc(lImageSize * sizeof(GLbyte));
if(pBits == NULL)
    return NULL;
// Read in the bits
// Check for read error. This should catch RLE or other
// weird formats that I don't want to recognize
if(fread(pBits, lImageSize, 1, pFile) != 1)
{
    free(pBits);
    return NULL;
}
// Set OpenGL format expected
switch(sDepth)
{
    case 3:      // Most likely case
        *eFormat = GL_BGR_EXT;
        *iComponents = GL_RGB8;
        break;
    case 4:
        *eFormat = GL_BGRA_EXT;
        *iComponents = GL_RGBA8;
        break;
    case 1:
        *eFormat = GL_LUMINANCE;
        *iComponents = GL_LUMINANCE8;
        break;
}
// Done with File
fclose(pFile);
// Return pointer to image data
return pBits;
}

```

You may notice that the number of components is not set to the integers 1, 3, or 4, but `GL_LUMINANCE8`, `GL_RGB8`, and `GL_RGBA8`. OpenGL recognizes these special constants as a request to maintain full image precision internally when it manipulates the image data. For example, for performance reasons, some OpenGL implementations may down-sample a 24-bit color image to 16 bits internally. This is especially common for texture loads (see [Chapter 8](#), "Texture Mapping: The Basics") on many implementations where the display output color resolution is only 16 bits, but a higher bit depth image is loaded. These constants are requests to the implementation to store and use the image data as supplied at their full 8-bit per channel color depth.

Moving Pixels Around

Writing pixel data to the color buffer can be very useful in and of itself, but you can also read pixel data from the color buffer and even copy data from one part of the color buffer to another. The function to read pixel data works just like `glDrawPixels`, but in reverse:

```
void glReadPixels(GLint x, GLint y, GLsizei width, GLsizei height,
                  GLenum format, GLenum type, const void *pixels);
```

You specify the `x` and `y` in window coordinates of the lower-left corner of the rectangle to read followed by the `width` and `height` of the rectangle in pixels. The `format` and `type` parameters are the format and type you want the data to have. If the color buffer stores data differently than what you have requested, OpenGL will take care of the necessary conversions. This capability can be very useful, especially after you learn a couple of magic tricks that you can do during this process using the `glPixelTransfer` function (coming up in the "[Pixel Transfer](#)" section). The pointer to the image data, `*pixels`, must be valid and must contain enough storage to contain the image data after conversion, or you will likely get a nasty memory exception at runtime.

Copying pixels from one part of the color buffer to another is also easy, and you don't have to allocate any temporary storage during the operation. First, set the raster position using `glRasterPos` or `glWindowPos` to the destination corner (remember, the lower-left corner) where you want the image data copied. Then use the following function to perform the copy operation:

```
void glCopyPixels(GLint x, GLint y, GLsizei width,
                  GLsizei height, GLenum type);
```

The `x` and `y` parameters specify the lower-left corner of the rectangle to copy, followed by the `width` and `height` in pixels. The `type` parameter should be `GL_COLOR` to copy color data. You can also use `GL_DEPTH` and `GL_STENCIL` here, and the copy will be performed in the depth or stencil buffer instead. Moving depth and stencil values around can also be useful for some rendering algorithms and special effects.

By default, all these pixel operations operate on the back buffer for double-buffered rendering contexts, and the front buffer for single-buffered rendering contexts. You can change the source or destination of these pixel operations by using these two functions:

```
void glDrawBuffer(GLenum mode);
void glReadBuffer(GLenum mode);
```

The `glDrawBuffer` function affects where pixels are drawn by either `glDrawPixels` or `glCopyPixels` operations. You can use any of the valid buffer constants discussed in [Chapter 3](#): `GL_NONE`, `GL_FRONT`, `GL_BACK`, `GL_FRONT_AND_BACK`, `GL_FRONT_LEFT`, `GL_FRONT_RIGHT`, and so on.

The `glReadBuffer` function accepts the same constants and sets the target color buffer for read operations performed by `glReadPixels` or `glCopyPixels`.

Saving Pixels

You now know enough about how to move pixels around to write another useful function for the `glTools` library. A counterpart to the targa loading function, `gltLoadTGA`, is `gltWriteTGA`. This function reads the color data from the front color buffer and saves it to an image file in the targa file format. You use this function in the next section when you start playing with some interesting OpenGL pixel operations. The complete listing for the `gltWriteTGA` function is shown in [Listing 7.4](#).

Listing 7.4. The `gltWriteTGA` Function to Save the Screen as a Targa File

```

// Capture the current viewport and save it as a targa file.
// Be sure and call SwapBuffers for double buffered contexts or
// glFinish for single buffered contexts before calling this function.
// Returns 0 if an error occurs, or 1 on success.

GLint gltWriteTGA(const char *szFileName)
{
    FILE *pFile;           // File pointer
    TGAHEADER tgaHeader;   // TGA file header
    unsigned long lImageSize; // Size in bytes of image
    GLbyte *pBits = NULL;  // Pointer to bits
    GLint iViewport[4];    // Viewport in pixels
    GLenum lastBuffer;    // Storage for the current read buffer setting

    // Get the viewport dimensions
    glGetIntegerv(GL_VIEWPORT, iViewport);
    // How big is the image going to be (targas are tightly packed)
    lImageSize = iViewport[2] * 3 * iViewport[3];
    // Allocate block. If this doesn't work, go home
    pBits = (GLbyte *)malloc(lImageSize);
    if(pBits == NULL)
        return 0;
    // Read bits from color buffer
    glPixelStorei(GL_PACK_ALIGNMENT, 1);
    glPixelStorei(GL_PACK_ROW_LENGTH, 0);
    glPixelStorei(GL_PACK_SKIP_ROWS, 0);
    glPixelStorei(GL_PACK_SKIP_PIXELS, 0);

    // Get the current read buffer setting and save it. Switch to
    // the front buffer and do the read operation. Finally, restore
    // the read buffer state
    glGetIntegerv(GL_READ_BUFFER, &lastBuffer);
    glReadBuffer(GL_FRONT);
    glReadPixels(0, 0, iViewport[2], iViewport[3], GL_BGR,
                GL_UNSIGNED_BYTE, pBits);
    glReadBuffer(lastBuffer);
    // Initialize the Targa header
    tgaHeader.identsize = 0;
    tgaHeader.colorMapType = 0;
    tgaHeader.imageType = 2;
    tgaHeader.colorMapStart = 0;
    tgaHeader.colorMapLength = 0;
    tgaHeader.colorMapBits = 0;
    tgaHeader.xstart = 0;
    tgaHeader.ystart = 0;
    tgaHeader.width = iViewport[2];
    tgaHeader.height = iViewport[3];
    tgaHeader.bits = 24;
    tgaHeader.descriptor = 0;
    // Do byte swap for big vs little endian
#endif __APPLE__
    BYTE_SWAP(tgaHeader.colorMapStart);
    BYTE_SWAP(tgaHeader.colorMapLength);

```

```

BYTE_SWAP(tgaHeader.xstart);
BYTE_SWAP(tgaHeader.ystart);
BYTE_SWAP(tgaHeader.width);
BYTE_SWAP(tgaHeader.height);
#endif
// Attempt to open the file
pFile = fopen(szFileName, "wb");
if(pFile == NULL)
{
    free(pBits); // Free buffer and return error
    return 0;
}
// Write the header
fwrite(&tgaHeader, sizeof(TGAHEADER), 1, pFile);
// Write the image data
fwrite(pBits, lImageSize, 1, pFile);
// Free temporary buffer and close the file
free(pBits);
fclose(pFile);
// Success!
return 1;
}

```

More Fun with Pixels

In this section, we discuss OpenGL's support for magnifying and reducing images, flipping images, and performing some special operations during the transfer of pixel data to and from the color buffer. Rather than have a different sample program for every special effect discussed, we have provided one sample program named OPERATIONS. This sample program ordinarily displays a simple color image loaded from a targa file. A right mouse click is attached to the GLUT menu system, allowing you to select from one of eight drawing modes or to save the modified image to a disk file named `screenshot.tga`. Listing 7.5 provides the program in its entirety. We dissect this program and explain it piece by piece in the coming sections.

Listing 7.5. Source Code for the OPERATIONS Sample Program

```

// Operations.c
// OpenGL SuperBible
// Demonstrates Imaging Operations
// Program by Richard S. Wright Jr.
#include "../../../Common/OpenGLSB.h" // System and OpenGL Stuff
#include "../../../Common/GLTools.h" // OpenGL toolkit
#include <math.h>
///////////////////////////////
// Module globals to save source image data
static GLubyte *pImage = NULL;
static GLint iWidth, iHeight, iComponents;
static GLenum eFormat;
// Global variable to store desired drawing mode
static GLint iRenderMode = 1;
///////////////////////////////
// This function does any needed initialization on the rendering
// context.
void SetupRC(void)
{
    // Black background
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    // Load the horse image
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
}

```

```

pImage = gltLoadTGA( "horse.tga", &iWidth, &iHeight,
                      &iComponents, &eFormat );
}

void ShutdownRC(void)
{
    // Free the original image data
    free(pImage);
}

// Reset flags as appropriate in response to menu selections
void ProcessMenu(int value)
{
    if(value == 0)
        // Save image
        gltWriteTGA( "ScreenShot.tga" );
    else
        // Change render mode index to match menu entry index
        iRenderMode = value;

    // Trigger Redraw
    glutPostRedisplay();
}

// Called to draw scene
void RenderScene(void)
{
    GLint iViewport[4];
    GLbyte *pModifiedBytes = NULL;
    GLfloat invertMap[256];
    GLint i;
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT);
    // Current Raster Position always at bottom left hand corner
    glRasterPos2i(0, 0);
    // Do image operation, depending on rendermode index
    switch(iRenderMode)
    {
        case 2:      // Flip the pixels
            glPixelZoom(-1.0f, -1.0f);
            glRasterPos2i(iWidth, iHeight);
            break;
        case 3:      // Zoom pixels to fill window
            glGetIntegerv(GL_VIEWPORT, iViewport);
            glPixelZoom((GLfloat) iViewport[2] / (GLfloat)iWidth,
                        (GLfloat) iViewport[3] / (GLfloat)iHeight);
            break;

        case 4:      // Just Red
            glPixelTransferf(GL_RED_SCALE, 1.0f);
            glPixelTransferf(GL_GREEN_SCALE, 0.0f);
            glPixelTransferf(GL_BLUE_SCALE, 0.0f);
            break;
        case 5:      // Just Green
            glPixelTransferf(GL_RED_SCALE, 0.0f);
            glPixelTransferf(GL_GREEN_SCALE, 1.0f);
            glPixelTransferf(GL_BLUE_SCALE, 0.0f);
            break;
        case 6:      // Just Blue
            glPixelTransferf(GL_RED_SCALE, 0.0f);
            glPixelTransferf(GL_GREEN_SCALE, 0.0f);
            glPixelTransferf(GL_BLUE_SCALE, 1.0f);
            break;
    }
}

```

```

case 7:      // Black & White, more tricky
    // First draw image into color buffer
    glDrawPixels(iWidth, iHeight, eFormat,
                  GL_UNSIGNED_BYTE, pImage);
    // Allocate space for the luminance map
    pModifiedBytes = (GLbyte *)malloc(iWidth * iHeight);
    // Scale colors according to NSTC standard
    glPixelTransferf(GL_RED_SCALE, 0.3f);
    glPixelTransferf(GL_GREEN_SCALE, 0.59f);
    glPixelTransferf(GL_BLUE_SCALE, 0.11f);
    // Read pixels into buffer (scale above will be applied)
    glReadPixels(0,0,iWidth, iHeight, GL_LUMINANCE,
                  GL_UNSIGNED_BYTE, pModifiedBytes);

    // Return color scaling to normal
    glPixelTransferf(GL_RED_SCALE, 1.0f);
    glPixelTransferf(GL_GREEN_SCALE, 1.0f);
    glPixelTransferf(GL_BLUE_SCALE, 1.0f);
    break;
case 8:      // Invert colors
    invertMap[0] = 1.0f;
    for(i = 1; i < 256; i++)
        invertMap[i] = 1.0f - (1.0f / 255.0f * (GLfloat)i);
    glPixelMapfv(GL_PIXEL_MAP_R_TO_R, 255, invertMap);
    glPixelMapfv(GL_PIXEL_MAP_G_TO_G, 255, invertMap);
    glPixelMapfv(GL_PIXEL_MAP_B_TO_B, 255, invertMap);
    glPixelTransferi(GL_MAP_COLOR, GL_TRUE);
    break;
case 1:      // Just do a plain old image copy
default:
    // This line intentionally left blank
    break;
}
// Do the pixel draw
if(pModifiedBytes == NULL)
    glDrawPixels(iWidth, iHeight, eFormat, GL_UNSIGNED_BYTE,
                  pImage);
else
{
    glDrawPixels(iWidth, iHeight, GL_LUMINANCE, GL_UNSIGNED_BYTE,
                  pModifiedBytes);
    free(pModifiedBytes);
}
// Reset everything to default
glPixelTransferi(GL_MAP_COLOR, GL_FALSE);
glPixelTransferf(GL_RED_SCALE, 1.0f);
glPixelTransferf(GL_GREEN_SCALE, 1.0f);
glPixelTransferf(GL_BLUE_SCALE, 1.0f);
glPixelZoom(1.0f, 1.0f);                                // No Pixel Zooming
// Do the buffer Swap
glutSwapBuffers();
}
void ChangeSize(int w, int h)
{
    // Prevent a divide by zero, when window is too short
    // (you can't make a window of zero width).
    if(h == 0)
        h = 1;
    glViewport(0, 0, w, h);
    // Reset the coordinate system before modifying
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
}

```

```

// Set the clipping volume
gluOrtho2D(0.0f, (GLfloat) w, 0.0, (GLfloat) h);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}

///////////////
// Main program entrypoint
int main(int argc, char* argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GL_DOUBLE);
    glutInitWindowSize(800, 600);
    glutCreateWindow("OpenGL Image Operations");
    glutReshapeFunc(ChangeSize);
    glutDisplayFunc(RenderScene);
    // Create the Menu and add choices
    glutCreateMenu(ProcessMenu);
    glutAddMenuEntry("Save Image", 0);
    glutAddMenuEntry("DrawPixels", 1);
    glutAddMenuEntry("FlipPixels", 2);
    glutAddMenuEntry("ZoomPixels", 3);
    glutAddMenuEntry("Just Red Channel", 4);
    glutAddMenuEntry("Just Green Channel", 5);
    glutAddMenuEntry("Just Blue Channel", 6);
    glutAddMenuEntry("Black and White", 7);
    glutAddMenuEntry("Invert Colors", 8);
    glutAttachMenu(GLUT_RIGHT_BUTTON);
    SetupRC();           // Do setup
    glutMainLoop();      // Main program loop
    ShutdownRC();        // Do shutdown
    return 0;
}

```

The basic framework of this program is simple. Unlike the previous example, IMAGELOAD, here the image is loaded and kept in memory for the duration of the program so that reloading the image is not necessary every time the screen must be redrawn. The information about the image and a pointer to the bytes are kept as module global variables, as shown here:

```

static GLubyte *pImage = NULL;
static GLint iWidth, iHeight, iComponents;
static GLenum eFormat;

```

The `SetupRC` function then does little other than load the image and initialize the global variables containing the image format, width, and height:

```

// Load the horse image
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
pImage = gltLoadTGA("horse.tga", &iWidth, &iHeight,
                     &iComponents, &eFormat);

```

When the program terminates, you make sure to free the memory allocated by the `gltLoadTGA` function in `ShutdownRC`:

```
free(pImage);
```

In the main function, you create a menu and add entries and values for the different operations

you want to accomplish:

```
// Create the Menu and add choices
glutCreateMenu(ProcessMenu);
glutAddMenuEntry("Save Image",0);
glutAddMenuEntry("Draw Pixels",1);
glutAddMenuEntry("Flip Pixels",2);
glutAddMenuEntry("Zoom Pixels",3);
glutAddMenuEntry("Just Red Channel",4);
glutAddMenuEntry("Just Green Channel",5);
glutAddMenuEntry("Just Blue Channel",6);
glutAddMenuEntry("Black and White", 7);
glutAddMenuEntry("Invert Colors", 8);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

These menu selections then set the variable `iRenderMode` to the desired value or, if the value is 0, save the image as it is currently displayed:

```
void ProcessMenu(int value)
{
    if(value == 0)
        // Save image
        gltWriteTGA("ScreenShot.tga");
    else
        // Change render mode index to match menu entry index
        iRenderMode = value;
    // Trigger Redraw
    glutPostRedisplay();
}
```

Finally, the image is actually drawn into the color buffer in the `RenderScene` function. This function contains a switch statement that uses the `iRenderMode` variable to select from one of eight different drawing modes. The default case is simply to perform an unaltered `glDrawPixels` function, placing the image in the lower-left corner of the window, as shown in Figure 7.6. The other cases, however, are now the subject of our discussion.

Figure 7.6. The default output of the OPERATIONS sample program.



Pixel Zoom

Another simple yet common operation that you may want to perform on pixel data is stretching or shrinking the image. OpenGL calls this *pixel zoom* and provides a function that performs this operation:

```
void glPixelZoom(GLfloat xfactor, GLfloat yfactor);
```

The two arguments, *xfactor* and *yfactor*, specify the amount of zoom to occur in the x and y directions. Zoom can shrink, expand, or even reverse an image. For example, a zoom factor of 2 causes the image to be written at twice its size along the axis specified, whereas a factor of 0.5 shrinks it by half. As an example, the menu selection *Zoom Pixels* in the OPERATIONS sample program sets the render mode to 3. The following code lines are then executed before the call to *glDrawPixels*, causing the x and y zoom factors to stretch the image to fill the entire window:

```
case 3:    // Zoom pixels to fill window
    glGetIntegerv(GL_VIEWPORT, iViewport);
    glPixelZoom((GLfloat) iViewport[2] / (GLfloat)iWidth,
                (GLfloat) iViewport[3] / (GLfloat)iHeight);
    break;
```

The output is shown in Figure 7.7 .

Figure 7.7. Using pixel zoom to stretch an image to match the window size.



A negative zoom factor, on the other hand, has the effect of flipping the image along the direction of the zoom. Using such a zoom factor not only reverses the order of the pixels in the image, but it also reverses the direction onscreen that the pixels are drawn with respect to the raster position. For example, normally an image is drawn with the lower-left corner being placed at the current raster position. If both zoom factors are negative, the raster position becomes the upper-right corner of the resulting image.

In the OPERATIONS sample program, selecting Flip Pixels inverts the image both horizontally and vertically. As shown in the following code snippet, the pixel zoom factors are both set to -1.0 , and the raster position is changed from the lower-left corner of the window to a position that represents the upper-right corner of the image to be drawn (the image's width and height):

```
case 2:    // Flip the pixels
    glPixelZoom(-1.0f, -1.0f);
    glRasterPos2i(iWidth, iHeight);
    break;
```

Figure 7.8 shows the inverted image when this option is selected.

Figure 7.8. Image displayed with x and y dimensions inverted.



Pixel Transfer

In addition to zooming pixels, OpenGL supports a set of simple mathematical operations that can be performed on image data as it is transferred either to or from the color buffer. These pixel transfer modes are set with one of the following functions and the pixel transfer parameters listed in Table 7.4 :

```
void glPixelTransferi(GLenum pname, GLint param);
void glPixelTransferf(GLenum pname, GLfloat param);
```

GL_MAP_COLOR

GLboolean

GL_FALSE

GL_MAP_STENCIL

GLboolean

GL_FALSE

GL_RED_SCALE

GLfloat

1.0

GL_GREEN_SCALE

`GLfloat`

1.0

`GL_BLUE_SCALE`

`GLfloat`

1.0

`GL_ALPHA_SCALE`

`GLfloat`

1.0

`GL_DEPTH_SCALE`

`GLfloat`

1.0

`GL_RED_BIAS`

`GLfloat`

0.0

`GL_GREEN_BIAS`

`GLfloat`

0.0

`GL_BLUE_BIAS`

`GLfloat`

0.0

`GL_ALPHA_BIAS`

`GLfloat`

0.0

`GL_DEPTH_BIAS`

`GLfloat`

0.0

GL_POST_CONVOLUTION_RED_SCALE

GLfloat

1.0

GL_POST_CONVOLUTION_GREEN_SCALE

GLfloat

1.0

GL_POST_CONVOLUTION_BLUE_SCALE

GLfloat

1.0

GL_POST_CONVOLUTION_ALPHA_SCALE

GLfloat

1.0

GL_POST_CONVOLUTION_RED_BIAS

GLfloat

0.0

GL_POST_CONVOLUTION_GREEN_BIAS

GLfloat

0.0

GL_POST_CONVOLUTION_BLUE_BIAS

GLfloat

0.0

GL_POST_CONVOLUTION_ALPHA_BIAS

GLfloat

0.0

GL_POST_COLOR_MATRIX_RED_SCALE

GLfloat

1.0

`GL_POST_COLOR_MATRIX_GREEN_SCALE`

`GLfloat`

1.0

`GL_POST_COLOR_MATRIX_BLUE_SCALE`

`GLfloat`

1.0

`GL_POST_COLOR_MATRIX_ALPHA_SCALE`

`GLfloat`

1.0

`GL_POST_COLOR_MATRIX_RED_BIAS`

`GLfloat`

0.0

`GL_POST_COLOR_MATRIX_GREEN_BIAS`

`GLfloat`

0.0

`GL_POST_COLOR_MATRIX_BLUE_BIAS`

`GLfloat`

0.0

`GL_POST_COLOR_MATRIX_ALPHA_BIAS`

`GLfloat`

0.0

Table 7.4. Pixel Transfer Parameters

Constant	Type	Default Value

The scale and bias parameters allow you to scale and bias individual color channels. A scaling factor is multiplied by the component value, and a bias value is added to that component value. A scale and bias operation is common in computer graphics for adjusting color channel values. The equation is simple:

$$\text{New Value} = (\text{Old Value} * \text{Scale Value}) + \text{Bias Value}$$

By default, the scale values are 1.0, and the bias values are 0.0. They essentially have no effect on the component values. Say you want to display a color image's red component values only. To do this, you set the blue and green scale factors to 0.0 before drawing and back to 1.0 afterward:

```
glPixelTransferf(GL_GREEN_SCALE, 0.0f);
glPixelTransfer(GL_BLUE_SCALE, 0.0f);
```

The OPERATIONS sample program includes the menu selections Just Red, Just Green, and Just Blue, which demonstrate this particular example. Each selection turns off all but one color channel to show the image's red, green, or blue color values only:

```
case 4: // Just Red
    glPixelTransferf(GL_RED_SCALE, 1.0f);
    glPixelTransferf(GL_GREEN_SCALE, 0.0f);
    glPixelTransferf(GL_BLUE_SCALE, 0.0f);
    break;
case 5: // Just Green
    glPixelTransferf(GL_RED_SCALE, 0.0f);
    glPixelTransferf(GL_GREEN_SCALE, 1.0f);
    glPixelTransferf(GL_BLUE_SCALE, 0.0f);
    break;
case 6: // Just Blue
    glPixelTransferf(GL_RED_SCALE, 0.0f);
    glPixelTransferf(GL_GREEN_SCALE, 0.0f);
    glPixelTransferf(GL_BLUE_SCALE, 1.0f);
    break;
```

After drawing, the pixel transfer for the color channels resets the scale values to 1.0:

```
glPixelTransferf(GL_RED_SCALE, 1.0f);
glPixelTransferf(GL_GREEN_SCALE, 1.0f);
glPixelTransferf(GL_BLUE_SCALE, 1.0f);
```

The post-convolution and post-color matrix scale and bias parameters perform the same operation but wait until after the convolution or color matrix operations have been performed. These operations are available in the imaging subset, which is discussed shortly.

A more interesting example of the pixel transfer operations is to display a color image in black and white. The OPERATIONS sample does this when you choose the Black and White menu selection. First, the full color image is drawn to the color buffer:

```
glDrawPixels(iWidth, iHeight, eFormat, GL_UNSIGNED_BYTE, pImage);
```

Next, a buffer large enough to hold just the luminance values for each pixel is allocated:

```
pModifiedBytes = (GLbyte *)malloc(iWidth * iHeight);
```

Remember, a luminance image has only one color channel, and here you allocate 1 byte (8 bits)

per pixel. OpenGL automatically converts the image in the color buffer to luminance for use when you call `glReadPixels` but request the data be in the `GL_LUMINANCE` format:

```
glReadPixels(0,0,iWidth, iHeight, GL_LUMINANCE, GL_UNSIGNED_BYTE, pModifiedBytes);
```

The luminance image can then be written back into the color buffer, and you would see the converted black-and-white image:

```
glDrawPixels(iWidth, iHeight, GL_LUMINANCE, GL_UNSIGNED_BYTE,
             pModifiedBytes);
```

Using this approach sounds like a good plan, and it almost works. The problem is that when OpenGL converts a color image to luminance, it simply adds the color channels together. If the three color channels add up to a value greater than 1.0, it is simply clamped to 1.0. This has the effect of oversaturating many areas of the image. This effect is shown in Figure 7.9 .

Figure 7.9. Oversaturation due to OpenGL's default color-to-luminance operation.



To solve this problem, you must set the pixel transfer mode to scale the color value appropriately when OpenGL does the transfer from color to luminance colorspace. According to the National Television Standards Committee (NTSC) standard, the conversion from RGB colorspace to black and white (grayscale) is

$$\text{Luminance} = (0.3 * \text{Red}) + (0.59 * \text{Green}) + (0.11 * \text{Blue})$$

You can easily set up this conversion in OpenGL by calling these functions just before `glReadPixels` :

```
// Scale colors according to NTSC standard
glPixelTransferf(GL_RED_SCALE, 0.3f);
glPixelTransferf(GL_GREEN_SCALE, 0.59f);
```

```
glPixelTransferf(GL_BLUE_SCALE, 0.11f);
```

After reading pixels, you return the pixel transfer mode to normal:

```
// Return color scaling to normal
glPixelTransferf(GL_RED_SCALE, 1.0f);
glPixelTransferf(GL_GREEN_SCALE, 1.0f);
glPixelTransferf(GL_BLUE_SCALE, 1.0f);
```

The output is now a nice grayscale representation of the image. Because the figures in this book are not in color, but grayscale, the output onscreen looks exactly like the image in Figure 7.6 .

Pixel Mapping

In addition to scaling and bias operations, the pixel transfer operation also supports color mapping. A color map is a table used as a lookup to convert one color value (used as an index into the table) to another color value (the color value stored at that index). Color mapping has many applications, such as performing color corrections, making gamma adjustments, or converting to and from different color representations.

You'll notice an interesting example in the OPERATIONS sample program when you select Invert Colors. In this case, a color map is set up to flip all the color values during a pixel transfer. This means all three channels are mapped from the range 0.0 to 1.0 to the range 1.0 to 0.0. The result is an image that looks like a photographic negative.

You enable pixel mapping by calling `glPixelTransfer` with the `GL_MAP_COLOR` parameter set to `GL_TRUE` :

```
glPixelTransferi(GL_MAP_COLOR, GL_TRUE);
```

To set up a pixel map, you must call another function, `glPixelMap` , and supply the map in one of three formats:

```
glPixelMapuiv(GLenum map, GLint mapsize, GLuint *values);
glPixelMapusv(GLenum map, GLint mapsize, GLushort *values);
glPixelMapfv(GLenum map, GLint mapsize, GLfloat *values);
```

The valid map values are listed in Table 7.5 .

`GL_PIXEL_MAP_R_TO_R`

`GL_PIXEL_MAP_G_TO_G`

`GL_PIXEL_MAP_B_TO_B`

`GL_PIXEL_MAP_A_TO_A`

Table 7.5.
Pixelmap
Parameters

Map
Name

For the example, you set up a map of 256 floating-point values and fill the map with intermediate values from 1.0 to 0.0:

```
GLfloat invertMap[256];
...
...
invertMap[0] = 1.0f;
    for(i = 1; i < 256; i++)
        invertMap[i] = 1.0f - (1.0f / 255.0f * (GLfloat)i);
```

Then you set the red, green, and blue maps to this inversion map and turn on color mapping:

```
glPixelMapfv(GL_PIXEL_MAP_R_TO_R, 255, invertMap);
glPixelMapfv(GL_PIXEL_MAP_G_TO_G, 255, invertMap);
glPixelMapfv(GL_PIXEL_MAP_B_TO_B, 255, invertMap);
glPixelTransferi(GL_MAP_COLOR, GL_TRUE);
```

When `glDrawPixels` is called, the color components are remapped using the inversion table, essentially creating a color negative image. Figure 7.10 shows the output.

Figure 7.10. Using a color map to create a color negative image.



The Imaging "Subset"

All the OpenGL functions covered so far in this chapter for image manipulation have been a part of the core OpenGL API since version 1.0. The only exception is the `glWindowPos` function, which was added in OpenGL 1.4 to make it easier to set the raster position. These features provide OpenGL with adequate support for most image manipulation needs. For more advanced imaging operations, OpenGL may also include, as of version 1.2, an *imaging subset*. The imaging subset is optional, which means vendors may choose not to include this functionality in their implementation. However, if the imaging subset is supported, it is an all-or-nothing commitment to support the entire functionality of these features.

Your application can determine at runtime whether the imaging subset is supported by searching the extension string for the token `"GL_ARB_imaging"`. For example, when you use the `glTools` library, your code might look something like this:

```
if(gltIsExtSupported( "GL_ARB_imaging" ) == 0)
{
    // Error, imaging not supported
    ...
}
else
{
    // Do some imaging stuff
    ...
    ...
}
```

You access the imaging subset through the OpenGL extension mechanism, which means you will likely need to use the `glext.h` header file and obtain function pointers for the functions you need to use. Some OpenGL implementations, depending on your platform's development tools, may already have these functions and constants included in the `gl.h` OpenGL header file (for example, in the Apple XCode headers, they are already defined). For compiles on the Macintosh, we use the built-in support for the imaging subset; for the PC, we use the extension mechanism to obtain function pointers to the imaging functions.

The IMAGING sample program is modeled much like the previous OPERATIONS sample program in that a single sample program demonstrates different operations via the context menu. When the program starts, it checks for the availability of the imaging subset and aborts if it is not found:

```
// Check for imaging subset, must be done after window
// is create or there won't be an OpenGL context to query
if(gltIsExtSupported("GL_ARB_imaging") == 0)
{
    printf("Imaging subset not supported\r\n");
    return 0;
}
```

The entire `RenderScene` function is presented in [Listing 7.6](#). We discuss the various pieces of this function throughout this section.

Listing 7.6. The `RenderScene` Function from the Sample Program IMAGING

```
///////////////////////////////
// Called to draw scene
void RenderScene(void)
{
    GLint i;                                // Looping variable
    GLint iViewport[4];                      // Viewport
    GLint iLargest;                          // Largest histogram value
```

```

static GLubyte invertTable[256][3];// Inverted color table
// Do a black and white scaling
static GLfloat lumMat[16] = { 0.30f, 0.30f, 0.30f, 0.0f,
                             0.59f, 0.59f, 0.59f, 0.0f,
                             0.11f, 0.11f, 0.11f, 0.0f,
                             0.0f, 0.0f, 0.0f, 1.0f };
static GLfloat mSharpen[3][3] = { // Sharpen convolution kernel
{0.0f, -1.0f, 0.0f},
{-1.0f, 5.0f, -1.0f },
{0.0f, -1.0f, 0.0f }};
static GLfloat mEmboss[3][3] = { // Emboss convolution kernel
{ 2.0f, 0.0f, 0.0f },
{ 0.0f, -1.0f, 0.0f },
{ 0.0f, 0.0f, -1.0f }};
static GLint histoGram[256]; // Storage for histogram statistics
// Clear the window with current clearing color
glClear(GL_COLOR_BUFFER_BIT);
// Current Raster Position always at bottom left hand corner of window
glRasterPos2i(0, 0);
glGetIntegerv(GL_VIEWPORT, iViewport);
glPixelZoom((GLfloat) iViewport[2] / (GLfloat)iWidth,
             (GLfloat) iViewport[3] / (GLfloat)iHeight);
if(bHistogram == GL_TRUE) // Collect Histogram data
{
    // We are collecting luminance data, use our conversion formula
    // instead of OpenGL's (which just adds color components together)
    glMatrixMode(GL_COLOR);
    glLoadMatrixf(lumMat);
    glMatrixMode(GL_MODELVIEW);
    // Start collecting histogram data, 256 luminance values
    glHistogram(GL_HISTOGRAM, 256, GL_LUMINANCE, GL_FALSE);
    glEnable(GL_HISTOGRAM);
}
// Do image operation, depending on rendermode index
switch(iRenderMode)
{
    case 5: // Sharpen image
        glConvolutionFilter2D(GL_CONVOLUTION_2D, GL_RGB, 3, 3,
                             GL_LUMINANCE, GL_FLOAT, mSharpen);
        glEnable(GL_CONVOLUTION_2D);
        break;
    case 4: // Emboss image
        glConvolutionFilter2D(GL_CONVOLUTION_2D, GL_RGB, 3, 3,
                             GL_LUMINANCE, GL_FLOAT, mEmboss);
        glEnable(GL_CONVOLUTION_2D);
        glMatrixMode(GL_COLOR);
        glLoadMatrixf(lumMat);
        glMatrixMode(GL_MODELVIEW);
        break;
    case 3: // Invert Image
        for(i = 0; i < 255; i++)
        {
            invertTable[i][0] = (GLubyte)(255 - i);
            invertTable[i][1] = (GLubyte)(255 - i);
            invertTable[i][2] = (GLubyte)(255 - i);
        }
        glColorTable(GL_COLOR_TABLE, GL_RGB, 256, GL_RGB,
                    GL_UNSIGNED_BYTE, invertTable);
        glEnable(GL_COLOR_TABLE);
        break;
    case 2: // Brighten Image
        glMatrixMode(GL_COLOR);

```

```

glScalef(1.25f, 1.25f, 1.25f);
glMatrixMode(GL_MODELVIEW);
break;
case 1: // Just do a plain old image copy
default:
    // This line intentionally left blank
    break;
}
// Do the pixel draw
glDrawPixels(iWidth, iHeight, eFormat, GL_UNSIGNED_BYTE, pImage);
// Fetch and draw histogram?
if(bHistogram == GL_TRUE)
{
    // Read histogram data into buffer
    glGetHistogram(GL_HISTOGRAM, GL_TRUE, GL_LUMINANCE, GL_INT, histoGram);
    // Find largest value for scaling graph down
    iLargest = 0;
    for(i = 0; i < 255; i++)
        if(iLargest < histoGram[i])
            iLargest = histoGram[i];
    // White lines
    glColor3f(1.0f, 1.0f, 1.0f);
    glBegin(GL_LINE_STRIP);
    for(i = 0; i < 255; i++)
        glVertex2f(GLfloat)i,
            (GLfloat)histoGram[i] / (GLfloat) iLargest * 128.0f);
    glEnd();
    bHistogram = GL_FALSE;
    glDisable(GL_HISTOGRAM);
}
// Reset everything to default
glMatrixMode(GL_COLOR);
glLoadIdentity();
glMatrixMode(GL_MODELVIEW);
glDisable(GL_CONVOLUTION_2D);
glDisable(GL_COLOR_TABLE);
// Show our hard work...
glutSwapBuffers();
}

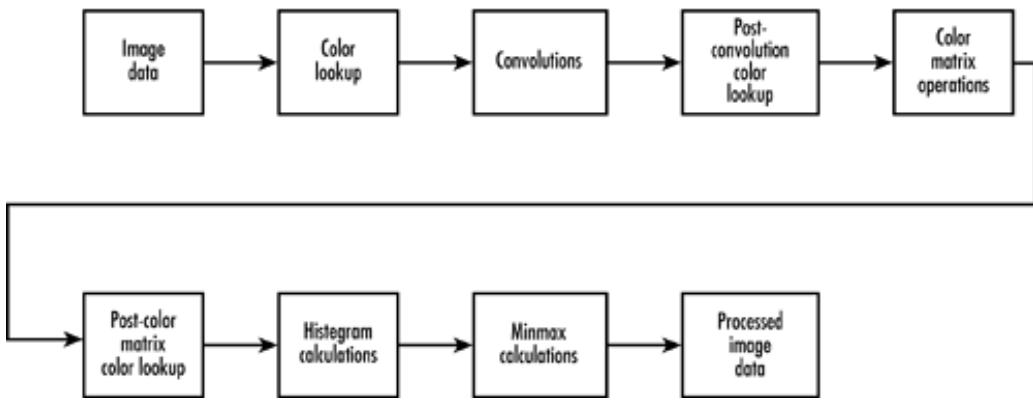
```

The image-processing subset can be broken down into three major areas of new functionality: the color matrix and color table, convolutions, and histograms. Bear in mind that image processing is a broad and complex topic all by itself and could easily warrant an entire book on this subject alone. What follows is an overview of this functionality with some simple examples of their use. For a more in-depth discussion on image processing, see the list of suggested references in [Appendix A](#), "Further Reading."

Imaging Pipeline

OpenGL imaging operations are processed in a specific order along what is called the *imaging pipeline*. In the same way that geometry is processed by the transformation pipeline, image data goes through the imaging operations in a fixed manner. [Figure 7.11](#) breaks down the imaging pipeline operation by operation. The sections that follow describe these operations in more detail.

Figure 7.11. The OpenGL imaging pipeline.



Color Matrix

The simplest piece of new functionality added with the imaging subset is the color matrix. You can think of color values as coordinates in colorspace—RGB being akin to XYZ on the color axis of the color cube (described in [Chapter 5](#), "Color, Materials, and Lighting: The Basics"). You could think of the alpha color component as the W component of a vector, and it would be transformed appropriately by a 4x4 color matrix. The color matrix is a matrix stack that works just like the other OpenGL matrix stacks (`GL_MODELVIEW`, `GL_PROJECTION`, `GL_TEXTURE`). You can make the color matrix stack the current stack by calling `glMatrixMode` with the argument `GL_COLOR`:

```
glMatrixMode(GL_COLOR);
```

All the matrix manipulation routines (`glLoadIdentity`, `glLoadMatrix`, and so on) are available for the color matrix. The color matrix stack can be pushed and popped as well, but implementations are required to support only a color stack two elements deep.

A menu item named Increase Contrast in the IMAGING sample program sets the render mode to 2, which causes the `RenderScene` function to use the color matrix to set a positive scaling factor to the color values, increasing the contrast of the image:

```
case 2:    // Brighten Image
    glMatrixMode(GL_COLOR);
    glScalef(1.25f, 1.25f, 1.25f);
    glMatrixMode(GL_MODELVIEW);
    break;
```

The effect is subtle yet clearly visible when the change occurs onscreen. After rendering, the color matrix is restored to identity:

```
// Reset everything to default
glMatrixMode(GL_COLOR);
glLoadIdentity();
glMatrixMode(GL_MODELVIEW);
```

Color Lookup

With color tables, you can specify a table of color values used to replace a pixel's current color. This functionality is similar to pixel mapping but has some added flexibility in the way the color table is composed and applied. The following function is used to set up a color table:

```
void glColorTable(GLenum target, GLenum internalFormat, GLsizei width,
                  GLenum format, GLenum type,
                  const GLvoid *table);
```

The `target` parameter specifies where in the imaging pipeline the color table is to be applied. This parameter may be one of the size values listed in [Table 7.6](#).

Table 7.6. The Place to Apply the Color Lookup Table

Target	Location
<code>GL_COLOR_TABLE</code>	Applied at the beginning of the imaging pipeline
<code>GL_POST_CONVOLUTION_COLOR_TABLE</code>	Applied after the convolution operation
<code>GL_POST_COLOR_MATRIX_COLOR_TABLE</code>	Applied after the color matrix operation
<code>GL_PROXY_COLOR_TABLE</code>	Verify this color table will fit
<code>GL_PROXY_POST_CONVOLUTION_COLOR_TABLE</code>	Verify this color table will fit
<code>GL_PROXY_POST_COLOR_MATRIX_COLOR_TABLE</code>	Verify this color table will fit

You use the `GL_PROXY` prefixed targets to verify that the supplied color table can be loaded (will fit into memory).

The `internalFormat` parameter specifies the internal OpenGL representation of the color table pointed to by `table`. It can be any of the following symbolic constants: `GL_ALPHA`, `GL_ALPHA4`, `GL_ALPHA8`, `GL_ALPHA12`, `GL_ALPHA16`, `GL_LUMINANCE`, `GL_LUMINANCE4`, `GL_LUMINANCE8`, `GL_LUMINANCE12`, `GL_LUMINANCE16`, `GL_LUMINANCE_ALPHA`, `GL_LUMINANCE4_ALPHA4`, `GL_LUMINANCE6_ALPHA2`, `GL_LUMINANCE8_ALPHA8`, `GL_LUMINANCE12_ALPHA4`, `GL_LUMINANCE12_ALPHA12`, `GL_LUMINANCE16_ALPHA16`, `GL_INTENSITY`, `GL_INTENSITY4`, `GL_INTENSITY8`, `GL_INTENSTIY12`, `GL_INTENSITY16`, `GL_RGB`, `GL_R3_G3_B2`, `GL_RGB4`, `GL_RGB5`, `GL_RGB8`, `GL_RGB10`, `GL_RGB12`, `GL_RGB16`, `GL_RGBA`, `GL_RGBA2`, `GL_RGBA4`, `GL_RGB5_A1`, `GL_RGBA8`, `GL_RGB10_A2`, `GL_RGBA12`, `GL_RGBA16`. The color component name in this list should be fairly obvious to you by now, and the numerical suffix simply represents the bit count of that component's representation.

The `format` and `type` parameters describe the format of the color table being supplied in the `table` pointer. The values for these parameters all correspond to the same arguments used in `glDrawPixels`, and are listed in [Tables 7.2](#) and [7.3](#).

The following example demonstrates a color table in action. It duplicates the color inversion effect from the OPERATIONS sample program but uses a color table instead of pixel mapping. When you choose the Invert Color menu selection, the render mode is set to 3, and the following segment of the `RenderScene` function is executed:

```
case 3: // Invert Image
    for(i = 0; i < 255; i++)
    {
        invertTable[i][0] = 255 - i;
        invertTable[i][1] = 255 - i;
        invertTable[i][2] = 255 - i;
    }
    glColorTable(GL_COLOR_TABLE, GL_RGB, 256, GL_RGB,
                GL_UNSIGNED_BYTE, invertTable);
    glEnable(GL_COLOR_TABLE);
```

For a loaded color table to be used, you must also enable the color table with a call to `glEnable`

with the `GL_COLOR_TABLE` parameter. After the pixels are drawn, the color table is disabled:

```
glDisable(GL_COLOR_TABLE);
```

The output from this example matches exactly the image from [Figure 7.10](#).

Proxies

An OpenGL implementation's support for color tables may be limited by system resources. Large color tables, for example, may not be loaded if they require too much memory. You can use the proxy color table targets listed in [Table 7.6](#) to determine whether a given color table fits into memory and can be used. These targets are used in conjunction with `glGetColorTableParameter` to see whether a color table will fit. The `glGetColorTableParameter` function enables you to query OpenGL about the various settings of the color tables; it is discussed in greater detail in the reference section. Here, you can use this function to see whether the width of the color table matches the width requested with the proxy color table call:

```
GLint width;
...
...
glColorTable(GL_PROXY_COLOR_TABLE, GL_RGB, 256, GL_RGB,
             GL_UNSIGNED_BYTE, NULL);
glGetColorTableParameteriv(GL_PROXY_COLOR_TABLE, GL_COLOR_TABLE_WIDTH, &width);
if(width == 0) {
    // Error...
}
...
```

Note that you do not need to specify the pointer to the actual color table for a proxy.

Other Operations

Also in common with pixel mapping, the color table can be used to apply a scaling factor and a bias to color component values. You do this with the following function:

```
void glColorTableParameteriv(GLenum target, GLenum pname, GLint *param);
void glColorTableParameterfv(GLenum target, GLenum pname, GLfloat *param);
```

The `glColorTableParameter` function's `target` parameter can be `GL_COLOR_TABLE`, `GL_POST_CONVOLUTION_COLOR_TABLE`, or `GL_POST_COLOR_MATRIX_COLOR_TABLE`. The `pname` parameter sets the scale or bias by using the value `GL_COLOR_TABLE_SCALE` or `GL_COLOR_TABLE_BIAS`, respectively. The final parameter is a pointer to an array of four elements storing the red, green, blue, and alpha scale or bias values to be used.

You can also actually render a color table by using the contents of the color buffer (after some rendering or drawing operation) as the source data for the color table. The function `glCopyColorTable` takes data from the current read buffer (the current `GL_READ_BUFFER`) as its source:

```
void glCopyColorTable(GLenum target, GLenum internalFormat,
                     GLint x, GLint y, GLsizei width);
```

The `target` and `internalFormat` parameters are identical to those used in `glColorTable`. The color table array is then taken from the color buffer starting at the `x,y` location and taking `width` pixels.

You can replace all or part of a color table by using the `glColorSubTable` function:

```
void glColorSubTable(GLenum target, GLsizei start, GLsizei count,
                     GLenum format, GLenum type, const void *data);
```

Here, most parameters correspond directly to the `glColorTable` function, except for `start` and `count`. The `start` parameter is the offset into the color table to begin the replacement, and `count` is the number of color values to replace.

Finally, you can also replace all or part of a color table from the color buffer in a manner similar to `glCopyColorTable` by using the `glCopyColorSubTable` function:

```
void glCopyColorSubTable(GLenum target, GLsizei start,
                        GLint x, GLint y, GLsizei width);
```

Again, the source of the color table is the color buffer, with `x` and `y` placing the position to begin reading color values, `start` being the location within the color table to begin the replacement, and `width` being the number of color values to replace.

Convolutions

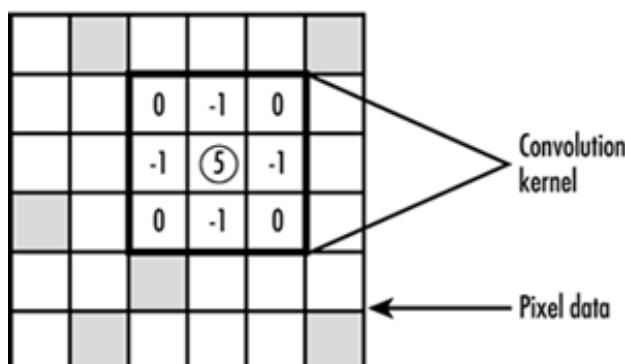
Convolutions are a powerful image-processing technique, with many applications such as blurring, sharpening, and other special effects. A convolution is a filter that processes pixels in an image according to some pattern of weights called a *kernel*. The convolution replaces each pixel with the weighted average value of that pixel and its neighboring pixels, with each pixel's color values being scaled by the weights in the kernel.

Typically, convolution kernels are rectangular arrays of floating-point values that represent the weights of a corresponding arrangement of pixels in the image. For example, the following kernel from the IMAGING sample program performs a sharpening operation:

```
static GLfloat mSharpen[3][3] = { // Sharpen convolution kernel
    {0.0f, -1.0f, 0.0f},
    {-1.0f, 5.0f, -1.0f },
    {0.0f, -1.0f, 0.0f };
```

The center pixel value is 5.0, which places a higher emphasis on that pixel value. The pixels immediately above, below, and to the right and left have a decreased weight, and the corner pixels are not accounted for at all. [Figure 7.12](#) shows a sample block of image data with the convolution kernel superimposed. The 5 in the kernel's center is the pixel being replaced, and you can see the kernel's values as they are applied to the surrounding pixels to derive the new center pixel value (represented by the circle). The convolution kernel is applied to every pixel in the image, resulting in a sharpened image. You can see this process in action by selecting Sharpen Image in the IMAGING sample program.

Figure 7.12. The sharpening kernel in action.



To apply the convolution filter, the IMAGING program simply calls these two functions before the `glDrawPixels` operation:

```
glConvolutionFilter2D(GL_CONVOLUTION_2D, GL_RGB, 3, 3,
                      GL_LUMINANCE, GL_FLOAT, mSharpen);
glEnable(GL_CONVOLUTION_2D);
```

The `glConvolutionFilter2D` function has the following syntax:

```
void glConvolutionFilter2D(GLenum target, GLenum internalFormat,
                           GLsizei width, GLsizei height, GLenum format,
                           GLenum type, const GLvoid *image);
```

The first parameter, `target`, must be `GL_CONVOLUTION_2D`. The second parameter, `internalFormat`, takes the same values as `glColorTable` and specifies to which pixel components the convolution is applied. The `width` and `height` parameters are the width and height of the convolution kernel. Finally, `format` and `type` specify the format and type of pixels stored in `image`. In the case of the sharpening filter, the pixel data is in `GL_RGB` format, and the kernel is `GL_LUMINANCE` because it contains simply a single weight per pixel (as opposed to having a separate weight for each color channel). Convolution kernels are turned on and off simply with `glEnable` or `glDisable` and the parameter `GL_CONVOLUTION_2D`.

Convolutions are a part of the imaging pipeline and can be combined with other imaging operations. For example, the sharpening filter already demonstrated was used in conjunction with pixel zoom to fill the entire window with the image. For a more interesting example, let's combine pixel zoom with the color matrix and a convolution filter. The following code excerpt defines a color matrix that will transform the image into a black-and-white (grayscale) image and a convolution filter that does embossing:

```
// Do a black and white scaling
static GLfloat lumMat[16] = { 0.30f, 0.30f, 0.30f, 0.0f,
                             0.59f, 0.59f, 0.59f, 0.0f,
                             0.11f, 0.11f, 0.11f, 0.0f,
                             0.0f, 0.0f, 0.0f, 1.0f };
static GLfloat mSharpen[3][3] = { // Sharpen convolution kernel
    {0.0f, -1.0f, 0.0f},
    {-1.0f, 5.0f, -1.0f},
    {0.0f, -1.0f, 0.0f} };
static GLfloat mEmboss[3][3] = { // Emboss convolution kernel
    {2.0f, 0.0f, 0.0f},
    {0.0f, -1.0f, 0.0f},
    {0.0f, 0.0f, -1.0f} };
```

When you select Emboss Image from the pop-up menu, the render state is changed to 4, and the following case from the `RenderScene` function is executed before `glDrawPixels`:

```
case 4: // Emboss image
    glConvolutionFilter2D(GL_CONVOLUTION_2D, GL_RGB, 3, 3,
                          GL_LUMINANCE, GL_FLOAT, mEmboss);
    glEnable(GL_CONVOLUTION_2D);
    glMatrixMode(GL_COLOR);
    glLoadMatrixf(lumMat);
    glMatrixMode(GL_MODELVIEW);
    break;
```

The embossed image is displayed in [Figure 7.13](#).

Figure 7.13. Using convolutions and the color matrix for an embossed effect.

From the Color Buffer

Convolution kernels can also be loaded from the color buffer. The following function behaves similarly to loading a color table from the color buffer:

```
void glCopyConvolutionFilter2D(GLenum target, GLenum internalFormat,
                           GLint x, GLint y, GLsizei width, GLsizei height);
```

The *target* value must always be `GL_CONVOLUTION_2D`, and *internalFormat* refers to the format of the color data, as in `glConvolutionFilter2D`. The kernel is loaded from pixel data from the color buffer located at *(x,y)* and the given *width* and *height*.

Separable Filters

A separable convolution filter is one whose kernel can be represented by the matrix outer product of two one-dimensional filters. For example, in [Figure 7.14](#), one-dimensional row and column matrices are multiplied to yield a final 3x3 matrix (the new kernel filter).

Figure 7.14. The outer product to two one-dimensional filters.

$$\begin{bmatrix} -1 \\ 2 \\ -1 \end{bmatrix} \begin{bmatrix} -1 & 2 & -1 \end{bmatrix} = \begin{bmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{bmatrix}$$

The following function is used to specify these two one-dimensional filters:

```
void glSeparableFilter2D(GLenum target, GLenum internalFormat,
                        GLsizei width, GLsizei height,
                        GLenum format, GLenum type,
                        void *row, const GLvoid *column);
```

The parameters all have the same meaning as in `glConvolutionFilter2D`, with the exception that now you have two parameters for passing in the address of the filters: `row` and `col`. The `target` parameter, however, must be `GL_SEPARABLE_2D` in this case.

One-Dimensional Kernels

OpenGL also supports one-dimensional convolution filters, but they are applied only to one-dimensional texture data. They behave in the same manner as two-dimensional convolutions, with the exception that they are applied only to rows of pixels (or actually *texels* in the case of one-dimensional texture maps). These one-dimensional convolutions have one-dimensional kernels, and you can use the corresponding functions for loading and copying the filters:

```
glConvolutionFilter1D(GLenum target, GLenum internalFormat,
                      GLsizei width, GLenum format, GLenum type,
                      const GLvoid *image);
glCopyConvolutionFilter1D(GLenum target, GLenum internalFormat,
                        GLint x, GLint y, GLsizei width);
```

Of course, with these functions the target must be set to `GL_CONVOLUTION_1D`.

Other Convolution Tweaks

When a convolution filter kernel is applied to an image, along the edges of the image the kernel will overlap and fall outside the image's borders. How OpenGL handles this situation is controlled via the convolution border mode. You set the convolution border mode by using the `glConvolutionParameter` function, which has four variations:

```
glConvolutionParameteri(GLenum target, GLenum pname, GLint param);
glConvolutionParameterf(GLenum target, GLenum pname, GLfloat param);
glConvolutionParameteriv(GLenum target, GLenum pname, GLint *params);
glConvolutionParameterfv(GLenum target, GLenum pname, GLfloat *params);
```

The `target` parameter for these functions can be either `GL_CONVOLUTION_1D`, `GL_CONVOLUTION_2D`, or `GL_SEPARABLE_2D`. To set the border mode, you use `GL_CONVOLUTION_BORDER_MODE` as the `pname` parameter and one of the border mode constants as `param`.

If you set `param` to `GL_CONSTANT_BORDER`, the pixels outside the image border are computed from a constant pixel value. To set this pixel value, call `glConvolutionParameterfv` with `GL_CONSTANT_BORDER` and a floating-point array containing the RGBA values to be used as the constant pixel color.

If you set the border mode to `GL_REDUCE`, the convolution kernel is not applied to the edge pixels. Thus, the kernel never overlaps the edge of the image. In this case, however, you should note that you are essentially shrinking the image by the width and height of the convolution filter.

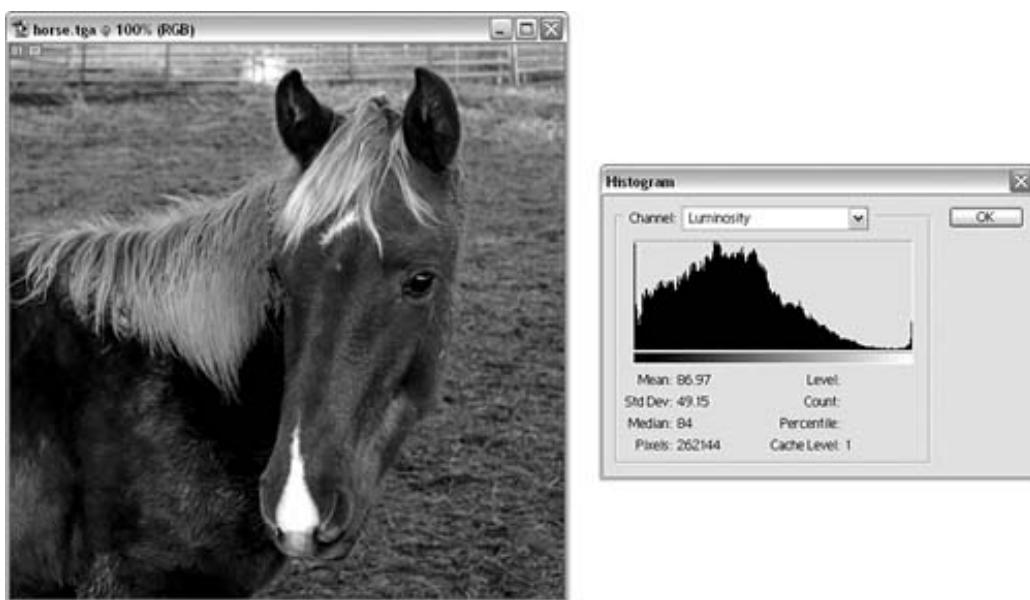
The final border mode is `GL_REPLICATE_BORDER`. In this case, the convolution is applied as if the horizontal and vertical edges of an image are replicated as many times as necessary to prevent overlap.

You can also apply a scale and bias value to kernel values by using `GL_CONVOLUTION_FILTER_BIAS` and/or `GL_CONVOLUTION_FILTER_SCALE` for the parameter name (*pname*) and supplying the bias and scale values in *param* or *params*.

Histogram

A histogram is a graphical representation of an image's frequency distribution. In English, it is simply a count of how many times each color value is used in an image, displayed as a sort of bar graph. Histograms may be collected for an image's intensity values or separately for each color channel. Histograms are frequently employed in image processing, and many digital cameras can display histogram data of captured images. Photographers use this information to determine whether the camera captured the full dynamic range of the subject or if perhaps the image is too over- or underexposed. Popular image-processing packages such as Adobe Photoshop also calculate and display histograms, as shown in Figure 7.15.

Figure 7.15. A histogram display in Photoshop.



When histogram collection is enabled, OpenGL collects statistics about any images as they are written to the color buffer. To prepare to collect histogram data, you must tell OpenGL how much data to collect and in what format you want the data. You do this with the `glHistogram` function:

```
void glHistogram(GLenum target, GLsizei width,
                 GLenum internalFormat, GLboolean sink);
```

The *target* parameter must be either `GL_HISTOGRAM` or `GL_PROXY_HISTOGRAM` (used to determine whether sufficient resources are available to store the histogram). The *width* parameter tells OpenGL how many entries to make in the histogram table. This value must be a power of 2 (1, 2, 4, 8, 16, and so on). The *internalFormat* parameter specifies the data format you expect the histogram to be stored in, corresponding to the valid format parameters for color tables and convolution filters, with the exception that `GL_INTENSITY` is not included. Finally, you can discard the pixels and not draw anything by specifying `GL_TRUE` for the *sink* parameter. You can turn histogram data collection on and off with `glEnable` or `glDisable` by passing in `GL_HISTOGRAM`, as in this example:

```
glEnable(GL_HISTOGRAM);
```

After image data has been transferred, you collect the histogram data with the following function:

```
void glGetHistogram(GLenum target, GLboolean reset, GLenum format,
                   GLenum type, GLvoid *values);
```

The only valid value for `target` is `GL_HISTOGRAM`. Setting `reset` to `GL_TRUE` clears the histogram data. Otherwise, the histogram becomes cumulative, and each pixel transfer continues to accumulate statistical data in the histogram. The `format` parameter specifies the data format of the collected histogram information, and `type` and `values` are the data type to be used and the address where the histogram is to be placed.

Now, let's look at an example using a histogram. In the IMAGING sample program, selecting Histogram from the menu displays a grayscale version of the image and a graph in the lower-left corner that represents the statistical frequency of each color luminance value. The output is shown in [Figure 7.16](#).

Figure 7.16. A histogram of the luminance values of the image.



The first order of business in the `RenderScene` function is to allocate storage for the histogram. The following line creates an array of integers 256 elements long. Each element in the array contains a count of the number of times that corresponding luminance value was used when the image was drawn onscreen:

```
static GLint histoGram[256]; // Storage for histogram statistics
```

Next, if the histogram flag is set (through the menu selection), you tell OpenGL to begin collecting histogram data. The function call to `glHistogram` instructs OpenGL to collect statistics about the 256 individual luminance values that may be used in the image. The sink is set to `false` so that the image is also drawn onscreen:

```

if(bHistogram == GL_TRUE)    // Collect Histogram data
{
    // We are collecting luminance data, use our conversion formula
    // instead of OpenGL's (which just adds color components together)
    glMatrixMode(GL_COLOR);
    glLoadMatrixf(lumMat);
    glMatrixMode(GL_MODELVIEW);
    // Start collecting histogram data, 256 luminance values
    glHistogram(GL_HISTOGRAM, 256, GL_LUMINANCE, GL_FALSE);
    glEnable(GL_HISTOGRAM);
}

```

Note that in this case you also need to set up the color matrix to provide the grayscale color conversion. OpenGL's default conversion to `GL_LUMINANCE` is simply a summing of the red, green, and blue color components. When you use this conversion formula, the histogram graph will have the same shape as the one from Photoshop for the same image display in [Figure 7.15](#).

After the pixels are drawn, you collect the histogram data with the code shown here:

```

// Fetch and draw histogram?
if(bHistogram == GL_TRUE)
{
    // Read histogram data into buffer
    glGetHistogram(GL_HISTOGRAM, GL_TRUE, GL_LUMINANCE, GL_INT, histoGram);
}

```

Now you traverse the histogram data and search for the largest collected value. You do this because you will use this value as a scaling factor to fit the graph in the lower-left corner of the display:

```

// Find largest value for scaling graph down
GLint iLargest = 0;
for(i = 0; i < 255; i++)
    if(iLargest < histoGram[i])
        iLargest = histoGram[i];

```

Finally, it's time to draw the graph of statistics. The following code segment simply sets the drawing color to white and then loops through the histogram data creating a single line strip. The data is scaled by the largest value so that the graph is 256 pixels wide and 128 pixels high. When all is done, the histogram flag is reset to `false` and the histogram data collection is disabled with a call to `glDisable`:

```

// White lines
glColor3f(1.0f, 1.0f, 1.0f);
glBegin(GL_LINE_STRIP);
    for(i = 0; i < 255; i++)
        glVertex2f((GLfloat)i, (GLfloat)histoGram[i] /
(GLfloat) iLargest * 128.0f);
glEnd();
bHistogram = GL_FALSE;
glDisable(GL_HISTOGRAM);
}

```

MinMax Operations

In the preceding sample, you traversed the histogram data to find the largest luminance component for the rendered image. If you need only the largest or smallest components collected, you can choose not to collect the entire histogram for a rendered image, but instead collect the largest and smallest values. This minmax data collection operates in a similar manner to histograms. First, you specify the format of the data on which you want statistics gathered by

using the following function:

```
void glMinmax(GLenum target, GLenum internalFormat, GLboolean sink);
```

Here, *target* is `GL_MINMAX`, and *internalFormat* and *sink* behave precisely as in `glHistogram`. You must also enable minmax data collection:

```
glEnable(GL_MINMAX);
```

The minmax data is collected with the `glGetMinmax` function, which is analogous to `glGetHistogram`:

```
void glGetMinmax(GLenum target, GLboolean reset, GLenum format,
                  GLenum type, GLvoid *values);
```

Again, the *target* parameter is `GL_MINMAX`, and the other parameters map to their counterparts in `glGetHistogram`.

Summary

In this chapter, we have shown that OpenGL provides first-class support for color image manipulation—from reading and writing bitmaps and color images directly to the color buffer, to color processing operations and color lookup maps. Optionally, many OpenGL implementations go even further by supporting the OpenGL imaging subset. The imaging subset makes it easy to add sophisticated image-processing filters and analysis to your graphics intensive programs.

We have also laid the groundwork in this chapter for our return to 3D geometry in the next chapter, where we begin coverage of OpenGL's texture mapping capabilities. You'll find the functions covered in this chapter that load and process image data are used directly when we extend the manipulation of image data by mapping it to 3D primitives.

Reference

glBitmap

Purpose: Draws a bitmap at the current raster position.

Include File: `<gl.h>`

Syntax:

```
void glBitmap(GLsizei width, GLsizei height,
             GLfloat xorig, GLfloat yorig,
             GLfloat xmove, GLfloat ymove, const
             GLubyte *bitmap);
```

Description: In OpenGL, bitmaps are binary image patterns without color data. This function draws a bitmap pattern at the current raster position. The width and height of the bitmap must be specified as well as any offset within the binary image map. The current raster position is updated after the bitmap is transferred by the amount specified by the *xmove* and *ymove* parameters. Bitmaps take the color that was current at the time the raster position was specified with `glRasterPos` or `glWindowPos`.

Parameters:

width, height **GLsizei**: The width and height of the bitmap.

xorig, yorig **GLsizei**: The location of the bitmap's origin measured from the bottom-left corner.

xmove, ymove **GLsizei**: The offset in the x and y direction added to the current raster position after drawing.

bitmap **const GLubyte***: Pointer to the bitmap image.

Returns: None.**See Also:** [glRasterPos](#), [glWindowPos](#), [glDrawPixels](#), [glPixelStore](#)**glColorSubTable****Purpose:** Replaces all or part of an existing color table.**Include File:** [<gl.h>](#)**Syntax:**

```
void glColorSubTable(GLenum target, GLsizei start,
→ GLsizei count,
→ GLenum format, GLenum type,
→ const void *data);
```

Description: This function allows you to replace all or a portion of an already established color table. Color tables consume system resources, and some performance advantage may be gained by replacing all or part of a color table rather than respecifying another table of the same size and format.**Parameters:**

target **GLenum**: Color table target. This parameter has the same meaning as when used for [glColorTable](#) and may be any value in [Table 7.6](#).

start **GLsizei**: Beginning position within the color table to begin the replacement.

count **GLsizei**: The count of color table entries to be replaced.

format **GLenum**: The format of the color table data. It may be any value used for specifying the original table with [glColorTable](#).

Returns: None.**See Also:** [glColorTable](#), [glCopyColorSubTable](#)**glColorTable****Purpose:** Specifies a table of color lookup replacement values.**Include File:** [<gl.h>](#)**Syntax:**

```
void glColorTable(GLenum target, GLenum
  ↪ internalFormat, GLsizei width,
  ↪ GLenum format,
  ↪ GLenum type, const GLvoid *table);
```

Description: Color tables are lookup tables used to replace pixel color values. The component color values of the pixel are used as the lookup values into the table pointed to by *table*. The color value stored at this location is substituted for the original color component value from the pixel being processed. The color table operation must be enabled by calling `glEnable(GL_COLOR_TABLE)`.

Parameters:

target `GLenum`: The color table being loaded. It may be any value from [Table 7.6](#).

internalFormat `GLenum`: The internal OpenGL representation of the table. It may be any of the following symbolic constants: `GL_ALPHA`, `GL_ALPHA4`, `GL_ALPHA8`, `GL_ALPHA12`, `GL_ALPHA16`, `GL_LUMINANCE`, `GL_LUMINANCE4`, `GL_LUMINANCE8`, `GL_LUMINANCE12`, `GL_LUMINANCE16`, `GL_LUMINANCE_ALPHA`, `GL_LUMINANCE4_ALPHA4`, `GL_LUMINANCE6_ALPHA2`, `GL_LUMINANCE8_ALPHA8`, `GL_LUMINANCE12_ALPHA4`, `GL_LUMINANCE12_ALPHA12`, `GL_LUMINANCE16_ALPHA16`, `GL_INTENSITY`, `GL_INTENSITY4`, `GL_INTENSITY8`, `GL_INTENSTIY12`, `GL_INTENSITY16`, `GL_RGB`, `GL_R3_G3_B2`, `GL_RGB4`, `GL_RGB5`, `GL_RGB8`, `GL_RGB10`, `GL_RGB12`, `GL_RGB16`, `GL_RGBA`, `GL_RGBA2`, `GL_RGBA4`, `GL_RGB5_A1`, `GL_RGBA8`, `GL_RGB10_A2`, `GL_RGBA12`, `GL_RGBA16`

width `GLsizei`: The width in pixels of the color table. This value must be a power of 2 (1, 2, 4, 8, 16, 32, 64, and so on).

format `GLenum`: The format of the pixel data. It may be any of the formats from [Table 7.2](#).

type `GLenum`: The data type of the pixel components. Any pixel data type from [Table 7.3](#) is valid.

table `void*`: A pointer to the array comprising the color table.

Returns: None.

See Also: `glColorSubTable`, `glColorTableParameter`, `glCopyColorSubTable`, `glCopyColorTable`

glColorTableParameter

Purpose: Sets a color table's scale and bias values.

Include File: `<gl.h>`

Variations:

```
void glColorTableParameteriv(GLenum target, GLenum
  ↪ pname, GLint *param);
void glColorTableParameterfv(GLenum target, GLenum
  ↪ pname, GLfloat *param);
```

Description: This function allows you to apply a scaling and bias factor to color table lookup values. Scaling factors are multiplied by the color component values, and bias values are added. Color component values are still clamped to the range [0,1].

Parameters:

target `GLenum`: The color table the parameter is to be applied to. It may be `GL_COLOR_TABLE`, `GL_POST_CONVOLUTION_COLOR_TABLE`, or `GL_POST_COLOR_MATRIX_COLOR_TABLE`.

pname `GLenum`: Either `GL_COLOR_TABLE_SCALE` to set the scaling factor or `GL_COLOR_TABLE_BIAS` to set the bias factor.

param `GLint*` or `GLfloat*`: A pointer to an array of values for the scale or bias factors. The array should have the same number of elements as the number of color components in the color table.

Returns: None.

See Also: [glColorTable](#)

glConvolutionFilter1D

Purpose: Specifies a one-dimensional convolution filter.

Include File: `<gl.h>`

Syntax:

```
void glConvolutionFilter1D(GLenum target, GLenum
  ↪ internalformat,
  ↪ GLsizei width, GLenum
  ↪ format, GLenum type,
  ↪ const GLvoid *image);
```

Description: This function sets a one-dimensional convolution filter that will be used with `glTexImage1D`. One-dimensional convolution filters have no effect on 2D imaging operations. You must enable this operation by calling `glEnable(GL_CONVOLUTION_1D)`.

Parameters:

target `GLenum`: The target of the filter. It must be `GL_CONVOLUTION_1D`.

internalformat `GLenum`: The pixel components the filter is to be applied to. Any of the formats listed in [Table 7.3](#) may be specified.

width `GLsizei`: The width of the filter.

format `GLenum`: The color format of the filter data. Any format from [Table 7.2](#) may be used.

type `GLenum`: The data type for the filter image values. Any of the data types from [Table 7.3](#) may be used.

image `GLvoid *:` A pointer to the filter image data.

Returns: None.

See Also: [glConvolutionFilter2D](#)

glConvolutionFilter2D

Purpose: Specifies a two-dimensional convolution filter.

Include File: `<gl.h>`

Syntax:

```
void glConvolutionFilter2D(GLenum target, GLenum
  ↪ internalformat,
  ↪ GLsizei width, GLsizei
  ↪ height, GLenum format,
  ↪ GLenum type, const
  ↪ GLvoid *image);
```

Description: This function sets a two-dimensional convolution filter to be used with `glCopyPixels`, `glDrawPixels`, `glReadPixels`, and `glTexImage2D`. You must enable this operation by calling `glEnable(GL_CONVOLUTION_2D)`.

Parameters:

`target` `GLenum`: The target of the filter. It must be `GL_CONVOLUTION_2D`.

`internalformat` `GLenum`: The pixel components the filter is to be applied to. Any of the formats listed in [Table 7.2](#) may be specified.

`width` `GLsizei`: The width of the filter.

`height` `GLsizei`: The height of the filter.

`format` `GLenum`: The color format of the filter data. Any format from [Table 7.3](#) may be used.

`type` `GLenum`: The data type for the filter image values. Any of the data types from [Table 7.3](#) may be used.

`image` `GLvoid *`: A pointer to the filter image data.

Returns: None.

See Also: [glConvolutionFilter1D](#)

glConvolutionParameter

Purpose: Sets convolution operating parameters.

Include File: `<gl.h>`

Syntax:

```
glConvolutionParameteri(GLenum target, GLenum
  ↪ pname, GLint param);
glConvolutionParameterf(GLenum target, GLenum
  ↪ pname, GLfloat param);
glConvolutionParameteriv(GLenum target, GLenum
  ↪ pname, GLint *params);
glConvolutionParameterfv(GLenum target, GLenum
  ↪ pname, GLfloat *params);
```

Description: This function sets parameters that affect the operation of the target convolution filter.

Parameters:

target GLenum: The convolution filter whose parameter is to be set. It must be `GL_CONVOLUTION_1D`, `GL_CONVOLUTION_2D`, or `GL_SEPARABLE_2D`.

pname GLenum: The convolution parameter to retrieve. It must be one of `GL_CONVOLUTION_BORDER_COLOR`, `GL_CONVOLUTION_BORDER_MODE`, `GL_CONVOLUTION_FILTER_SCALE`, or `GL_CONVOLUTION_FILTER_BIAS`.

param GLint or GLfloat: The value of the parameter to be set.

params GLint* or GLfloat*: A pointer to storage that contains the parameter values to be set.

Returns: None.

See Also: `glGetConvolutionParameter`

glCopyColorSubTable

Purpose: Replaces a portion of a color table with data from the color buffer.

Include File: `<gl.h>`

Syntax:

```
void glCopyColorSubTable(GLenum target, GLsizei start,
                        GLint x, GLint y, GLsizei
  ↪ width);
```

Description: This function replaces a color table's entries using values read from the color buffer.

Parameters:

target GLenum: The color table to operate on. It may be `GL_COLOR_TABLE`, `GL_POST_CONVOLUTION_COLOR_TABLE`, or `GL_POST_COLOR_MATRIX_COLOR_TABLE`.

start GLsizei: Offset into the color table to begin the replacement.

x, y GLsizei: The x and y location in the color buffer to begin the data extraction.

width GLsizei: The number of color table entries to replace.

Returns: None.**See Also:** [glColorTable](#), [glColorSubTable](#), [glCopyColorTable](#)

glCopyColorTable

Purpose: Creates a new color table using data from the color buffer.**Include File:** `<gl.h>`**Syntax:**

```
void glCopyColorTable(GLenum target, GLenum
➥ internalFormat,
➥ GLint x, GLint y, GLsizei
➥ width);
```

Description: This function creates a new color table in the fashion of [glColorTable](#). However, this function reads the color table values from the color buffer instead of from a user-specified data buffer.

Parameters:

`target` `GLenum`: The color table to operate on. It may be `GL_COLOR_TABLE`, `GL_POST_CONVOLUTION_COLOR_TABLE`, or `GL_POST_COLOR_MATRIX_COLOR_TABLE`.

`internalFormat` `GLenum`: The internal OpenGL format of the color data. It may be any of the same set of values used for [glColorTable](#) listed in [Table 7.2](#).

`x, y` `GLint`: The location in the color buffer to begin reading the color table values.

`width` `GLsizei`: The number of color entries to load.

Returns: None.**See Also:** [glColorTable](#), [glCopyColorSubTable](#)

glCopyConvolutionFilter1D

Purpose: Defines a one-dimensional convolution filter from the color buffer.**Include File:** `<gl.h>`**Syntax:**

```
void glCopyConvolutionFilter1D(GLenum target,
➥ GLenum internalFormat,
➥ GLint x, GLint y,
➥ GLsizei width);
```

Description: This function loads a one-dimensional convolution filter using data read from the color buffer.

Parameters:

target `GLenum`: The target of the filter. It must be `GL_CONVOLUTION_1D`.

internalformat `GLenum`: The pixel components the filter is to be applied to. Any of the formats listed in [Table 7.2](#) may be specified.

x, y `GLint`: The x and y location in the color buffer that specifies the beginning of the data source for the convolution filter.

width `GLsizei`: The width of the convolution filter in pixels.

Returns: None.

See Also: `glConvolutionFilter1D`, `glCopyConvolutionFilter2D`

glCopyConvolutionFilter2D

Purpose: Defines a two-dimensional convolution filter from the color buffer.

Include File: `<gl.h>`

Syntax:

```
void glCopyConvolutionFilter2D(GLenum target,
    ➔ GLenum internalFormat,
                    GLint x, GLint y,
    ➔ GLsizei width, GLsizei height);
```

Description: This function loads a two-dimensional convolution filter using data read from the color buffer.

Parameters:

target `GLenum`: The target of the filter. It must be `GL_CONVOLUTION_2D`.

internalformat `GLenum`: The pixel components the filter is to be applied to. Any of the formats listed in [Table 7.2](#) may be specified.

x, y `GLint`: The x and y location in the color buffer that specifies the beginning of the data source for the convolution filter.

width `GLsizei`: The width of the convolution filter in pixels.

height `GLsizei`: The height of the convolution filter in pixels.

Returns: None.

See Also: `glConvolutionFilter2D`, `glCopyConvolutionFilter1D`

glCopyPixels

Purpose: Copies a block of pixels in the framebuffer.

Include File: `<gl.h>`

Syntax:

```
void glCopyPixels(GLint x, GLint y, GLsizei width,
                  GLsizei height, GLenum type);
```

Description: This function copies pixel data from the indicated area in the framebuffer to the current raster position. You use `glRasterPos` or `glWindowPos` to set the current raster position. If the current raster position is not valid, no pixel data is copied. Calls to `glDrawBuffer`, `glPixelMap`, `glPixelTransfer`, `glPixelZoom`, `glReadBuffer`, and the imaging subset affect the operation of `glCopyPixels`.

Parameters:

`x` `GLint`: The lower-left corner window horizontal coordinate.

`y` `GLint`: The lower-left corner window vertical coordinate.

`width` `GLsizei`: The width of the image in pixels. If negative, the image is drawn from right to left. By default, images are drawn from left to right.

`height` `GLsizei`: The height of the image in pixels. If negative, the image is drawn from top to bottom. By default, images are drawn bottom to top.

`type` `GLenum`: The source of the pixel values to be copied. Any of the pixel types from [Table 7.7](#) may be used.

Table 7.7. Pixel Value Types

Type	Description
<code>GL_COLOR</code>	Color buffer values
<code>GL_STENCIL</code>	Stencil buffer values
<code>GL_DEPTH</code>	Depth buffer values

Returns: None.

See Also: `glDrawBuffer`, `glPixelMap`, `glPixelStore`, `glPixelTransfer`, `glPixelZoom`, `glReadBuffer`

glDrawPixels

Purpose: Draws a block of pixels into the framebuffer.

Include `<gl.h>`

File:

Syntax:

```
void glDrawPixels(GLsizei width, GLsizei height,
  ➔ GLenum format,
  GLenum type, const void *pixels);
```

Description: This function copies pixel data from memory to the current raster position. You use `glRasterPos` or `glWindowPos` to set the current raster position. If the current raster position is not valid, no pixel data is copied. Besides the `format` and `type` arguments, several other parameters define the encoding of pixel data in memory and control the processing of pixel data before it is placed in the color buffer.

Parameters:

`width` `GLsizei`: The width of the image in pixels.

`height` `GLsizei`: The height of the image in pixels. If negative, the image is drawn from top to bottom. By default, images are drawn from bottom to top.

`format` `GLenum`: The colorspace of the pixels to be drawn. Any value from [Table 7.2](#) may be used.

`type` `GLenum`: The data type of the color components. Valid data types are listed in [Table 7.3](#).

`pixels` `void *:` A pointer to the pixel data for the image.

Returns: None.

See Also: `glDrawBuffer`, `glPixelMap`, `glPixelStore`, `glPixelTransfer`, `glPixelZoom`

glGetConvolutionFilter

Purpose: Retrieves the current convolution filter.

Include File: `<gl.h>`

Syntax:

```
void glGetConvolutionFilter(GLenum target, GLenum
  ➔ format,
  GLenum type, void *data);
```

Description: This function allows you to query and read back the current convolution filter. You must allocate sufficient space in `*data` to store the convolution filter kernel.

Parameters:

`target` `GLenum`: The convolution filter to retrieve. It must be either `GL_CONVOLUTION_1D` or `GL_CONVOLUTION_2D`.

`format` `GLenum`: The desired pixel format of convolution data. It may be any of the pixel formats from [Table 7.2](#) with the exception of `GL_STENCIL_INDEX` and `GL_DEPTH_COMPONENT`.

`type` `GLenum`: The data type of the data storage. It may be any of the data types from [Table 7.3](#).

`data` `void *:` A pointer to the buffer to receive the convolution filter data.

Returns: None.**See Also:** [glConvolutionFilter1D](#), [glConvolutionFilter2D](#), [glGetConvolutionParameter](#)

glGetConvolutionParameter

Purpose: Queries for the various convolution parameters.**Include File:** `<gl.h>`**Variations:**

```
void glGetConvolutionParameteriv(GLenum target,
➥ GLenum pname, GLint* params);
void glGetConvolutionParameterfv(GLenum target,
➥ GLenum pname, GLfloat* params);
```

Description: This function allows you to query OpenGL for the current state of all the convolution parameters in effect.**Parameters:**

target GLenum: The convolution filter to retrieve. It must be `GL_CONVOLUTION_1D`, `GL_CONVOLUTION_2D`, or `GL_SEPARABLE_2D`.

pname GLenum: The convolution parameter to retrieve. It must be one of `GL_CONVOLUTION_BORDER_COLOR`, `GL_CONVOLUTION_BORDER_MODE`, `GL_CONVOLUTION_FILTER_SCALE`, `GL_CONVOLUTION_FILTER_BIAS`, `GL_CONVOLUTION_FORMAT`, `GL_CONVOLUTION_WIDTH`, `GL_CONVOLUTION_HEIGHT`, `GL_MAX_CONVOLUTION_WIDTH`, or `GL_MAX_CONVOLUTION_HEIGHT`.

params `GLint*` or `GLfloat*`: A pointer to storage that receives the results of the query specified by the *pname* parameter.

Returns: None.**See Also:** [glConvolutionFilter1D](#), [glConvolutionFilter2D](#), [glConvolutionParameter](#)

glGetColorTable

Purpose: Retrieves the contents of the current color table.**Include File:** `<gl.h>`**Syntax:**

```
void glGetColorTable(GLenum target, GLenum format,
➥ GLenum type, void *table);
```

Description: This function allows you to retrieve the contents of any of the current color tables. If necessary, the color table data is converted to the desired format.**Parameters:**

target `GLenum`: The color table being loaded. It may be any value from [Table 7.6](#).

format `GLenum`: The format of the pixel data. It may be any of the formats from [Table 7.2](#).

type `GLenum`: The data type of the data storage. It may be any of the data types from [Table 7.3](#).

table `void*`: A pointer to the buffer where the color table is to be copied.

Returns: None.

See Also: [glColorTable](#)

glGetColorTableParameter

Purpose: Retrieves the values of the color table parameter settings.

Include File: `<gl.h>`

Variations:

```
void glGetColorTableParameteriv(GLenum target,
→ GLenum pname, GLint* params);
void glGetColorTableParameterfv(GLenum target,
→ GLenum pname, GLfloat* params);
```

Description: This function allows you to query the values of any of the color table parameter settings. The results can be read back as either integers or floating-point values.

Parameters:

target `GLenum`: One of the color table names listed in [Table 7.6](#).

pname `GLenum`: One of the following color table parameters: `GL_COLOR_TABLE_SCALE`, `GL_COLOR_TABLE_BIAS`, `GL_COLOR_TABLE_FORMAT`, `GL_COLOR_TABLE_WIDTH`, `GL_COLOR_TABLE_RED_SIZE`, `GL_COLOR_TABLE_GREEN_SIZE`, `GL_COLOR_TABLE_BLUE_SIZE`, `GL_COLOR_TABLE_ALPHA_SIZE`, `GL_COLOR_TABLE_LUMINANCE_SIZE`, `GL_COLOR_TABLE_INTENSITY_SIZE`.

params `GLint*` or `GLfloat*`: The address of the variable to receive the retrieved parameter value.

Returns: None.

See Also: [glColorTableParameter](#)

glGetHistogram

Purpose: Returns collected histogram statistics.

Include File: `<gl.h>`

Syntax:

```
void glGetHistogram(GLenum target, GLboolean reset
→ , GLenum format,
GLenum type, GLvoid *values);
```

Description: This function retrieves the histogram data that was collected after a previous call to `glHistogram`. You must also enable histogram collection by enabling `GL_HISTOGRAM`.

Parameters:

`target` `GLenum`: The histogram target to retrieve. It must be `GL_HISTOGRAM`.

`reset` `GLboolean`: A flag to indicate whether the histogram statistics should be cleared after they are copied to values.

`format` `GLenum`: The storage format of the color counts. Any of the values from [Table 7.2](#) may be used.

`type` `GLenum`: The data type of the storage for the color counts. Any of the values from [Table 7.3](#) may be used.

`values` `void*`: A pointer to the buffer to receive the histogram statistics. Sufficient space must be allocated ahead of time to store the histogram data.

Returns: None.

See Also: `glHistogram`, `glResetHistogram`

glGetHistogramParameter

Purpose: Retrieves information about the current histogram parameters.

Include File: `<gl.h>`

Variations:

```
void glGetHistogramParameteriv(GLenum target,
→ GLenum pname, GLint* params);
void glGetHistogramParameterfv(GLenum target,
→ GLenum pname, GLfloat* params);
```

Description: These functions retrieve information about the way the current histogram is set up. The parameter `GL_HISTOGRAM_SINK` cannot be used with `GL_PROXY_HISTOGRAM`. Histogram proxies are used to determine whether a specific histogram can be used with the current system resources.

Parameters:

`target` `GLenum`: The histogram target about which to retrieve information. It must be `GL_HISTOGRAM` or `GL_PROXY_HISTOGRAM`.

pname GLenum: The parameter to retrieve. It may be one of the following:
params GLint* or GLfloat*: The address of the variable to receive the queried parameter.

Returns: None.

See Also: [glHistogram](#), [glGetHistogram](#)

glGetMinmax

Purpose: Gets the minimum and maximum color values.

Include File: `<gl.h>`

Syntax:

```
void glGetMinmax(GLenum target, GLboolean reset,
  ↪ GLenum format,
  GLenum type, GLvoid *values);
```

Description: This function gets the minimum and maximum color values that have been used.

Parameters:

target GLenum: The min/max buffer target. It must be [GL_MINMAX](#).

reset GLboolean: A value of [GL_TRUE](#) resets the minimum and maximum buffer values.

format GLenum: The format of the values array, taken from [Table 7.2](#).

type GLenum: The type of values in the array, taken from [Table 7.3](#).

values GLvoid *: A pointer to the values array.

Returns: None.

See Also: [glMinMax](#), [glResetMinMax](#)

glGetSeparableFilter

Purpose: Retrieves the contents of the current separable filter.

Include File: `<gl.h>`

Syntax:

```
void glGetSeparableFilter(GLenum target, GLenum
  ↪ format, GLenum type,
  ↪ void *row, void *column
  ↪ , const GLvoid *span);
```

Description: This function allows you to retrieve the size and contents of the current separable convolution filter.

Parameters:

target `GLenum`: The separable filter to retrieve. It must be `GL_SEPARABLE_2D`.

format `GLenum`: The desired format of the values. It may be any of the pixel values from [Table 7.2](#).

type `GLenum`: The data type of the data storage. It may be any of the data types from [Table 7.3](#).

row, column `void *:` Destination buffers for the row and column separable convolution filters.

span `void*:` An unused parameter reserved for future use.

Returns: None.

See Also: [glSeparableFilter2D](#)

glHistogram

Purpose: Defines how histogram statistics should be stored.

Include File: `<gl.h>`

Syntax:

```
void glHistogram(GLenum target, GLsizei width,
  ↪ GLenum internalFormat, GLboolean
  ↪ sink);
```

Description: This function defines how OpenGL should collect and store histogram data. If *target* is set to `GL_PROXY_HISTOGRAM`, the `glGetHistogramParameter` function can be used to see whether sufficient system resources are available to honor the request.

Parameters:

target `GLenum`: The histogram target. It must be either `GL_HISTOGRAM` or `GL_PROXY_HISTOGRAM`.

width `GLsizei`: The number of entries in the histogram table. This value must be a power of 2.

internalFormat **GLenum**: The way histogram data should be stored. Allowable values are **GL_ALPHA**, **GL_ALPHA4**, **GL_ALPHA8**, **GL_ALPHA12**, **GL_ALPHA16**, **GL_LUMINANCE**, **GL_LUMINANCE4**, **GL_LUMINANCE8**, **GL_LUMINANCE12**, **GL_LUMINANCE16**, **GL_LUMINANCE_ALPHA**, **GL_LUMINANCE4_ALPHA4**, **GL_LUMINANCE6_ALPHA2**, **GL_LUMINANCE8_ALPHA8**, **GL_LUMINANCE12_ALPHA4**, **GL_LUMINANCE12_ALPHA12**, **GL_LUMINANCE16_ALPHA16**, **GL_RGB**, **GL_R3_G3_B2**, **GL_RGB4**, **GL_RGB5**, **GL_RGB8**, **GL_RGB10**, **GL_RGB12**, **GL_RGB16**, **GL_RGBA**, **GL_RGBA2**, **GL_RGBA4**, **GL_RGB5_A1**, **GL_RGBA8**, **GL_RGB10_A2**, **GL_RGBA12**, **GL_RGBA16**.

sink **GLboolean**: Flag to determine whether pixels continue in the imaging pipeline.

Returns: None.

See Also: [glGetHistogram](#), [glResetHistogram](#)

glMinmax

Purpose: Initializes the min/max buffer.

Include File: `<gl.h>`

Syntax:

```
void glMinmax(GLenum target, GLenum internalFormat
  ↵ , GLboolean sink);
```

Description: This function initializes the min/max buffer.

Parameters:

target **GLenum**: This value must be **GL_MINMAX**.

internalFormat **GLenum**: The internal format of the values array. It may be any of the formats used by [glHistogram](#).

sink **GLboolean**: Flag to determine whether pixels are passed down the imaging pipeline.

Returns: None.

See Also: [glGetMinMax](#), [glResetMinMax](#)

glPixelMap

Purpose: Defines a lookup table for pixel transfers.

Include File: `<gl.h>`

Variations:

```

void glPixelMapfv(GLenum map, GLint mapsize, const
  ↪ GLfloat *values);
void glPixelMapuiv(GLenum map, GLint mapsize,
  ↪ const GLuint *values);
void glPixelMapusv(GLenum map, GLint mapsize,
  ↪ const GLushort *values);

```

Description: This function sets lookup tables for `glCopyPixels`, `glDrawPixel`, `glReadPixels`, `glTexImage1D`, and `glTexImage2D`. These lookup tables, or maps, are used only if the corresponding `GL_MAP_COLOR` or `GL_MAP_STENCIL` option is enabled with `glPixelTransfer`. Maps are applied prior to drawing and after reading values from the framebuffer.

Parameters:

`map` `GLenum`: The type of map being defined. Valid values are listed in [Table 7.5](#).

`mapsize` `GLint`: The size of the lookup table.

`values` `GLfloat *` or `GLuint *` or `GLushort *`: The lookup table.

Returns: None.

See Also: `glCopyPixels`, `glDrawPixels`, `glPixelStore`, `glPixelTransfer`, `glReadPixels`, `glTexImage1D`, `glTexImage2D`

glPixelStore

Purpose: Controls how pixels are stored or read from memory.

Include File: `<gl.h>`

Variations:

```

void glPixelStorei(GLenum pname, GLint param);
void glPixelStoref(GLenum pname, GLfloat param);

```

Description: This function controls how pixels are stored with `glReadPixels` and read for `glDrawPixels`, `glTexImage1D`, and `glTexImage2D`. It does not affect the operation of `glCopyPixels`.

Parameters:

`pname` `GLenum`: The parameter to set. It must be one of the values from [Table 7.8](#).

`param` `GLint` or `GLfloat`: The parameter value. Valid values for the different parameters are given in [Table 7.8](#).

Table 7.8. Pixel Storage Types

Name	Default	Description
<code>GL_PACK_SWAP_BYTES</code>	<code>GL_TRUE</code>	If true, all multibyte values have their bytes swapped when stored in memory.
<code>GL_PACK_LSB_FIRST</code>	<code>GL_FALSE</code>	If true, bitmaps have their leftmost pixel stored in bit 0 instead of bit 7.
<code>GL_PACK_ROW_LENGTH</code>	0	This storage type sets the pixel width of the image. If 0, the width argument is used instead.
<code>GL_PACK_SKIP_PIXELS</code>	0	This storage type sets the number of pixels to skip horizontally in the image.
<code>GL_PACK_SKIP_ROWS</code>	0	This storage type sets the number of pixels to skip vertically in the image.
<code>GL_PACK_ALIGNMENT</code>	4	This storage type sets the alignment of each scanline in the image.
<code>GL_UNPACK_SWAP_BYTES</code>	<code>GL_TRUE</code>	If true, all multibyte values have their bytes swapped when read from memory.
<code>GL_UNPACK_LSB_FIRST</code>	<code>GL_FALSE</code>	If true, bitmaps have their leftmost pixel read from bit 0 instead of bit 7.
<code>GL_UNPACK_ROW_LENGTH</code>	0	This storage type sets the pixel width of the image. If 0, the width argument is used instead.
<code>GL_UNPACK_SKIP_PIXELS</code>	0	This storage type sets the number of pixels to skip horizontally in the image.
<code>GL_UNPACK_SKIP_ROWS</code>	0	This storage type sets the number of pixels to skip vertically in the image.
<code>GL_UNPACK_ALIGNMENT</code>	4	This storage type sets the alignment of each scanline in the image.

Returns: None.

See Also: `glDrawPixels`, `glReadPixels`, `glTexImage1D`, `glTexImage2D`

glPixelTransfer

Purpose: Sets pixel transfer modes.

Include File: `<gl.h>`

Syntax:

```
void glPixelTransferi(GLenum pname, GLint param);
void glPixelTransferf(GLenum pname, GLfloat param);
```

Description: This function sets pixel transfer modes for `glCopyPixels`, `glDrawPixels`, `glReadPixels`, `glTexImage1D`, and `glTexImage2D`.

Parameters:

pname **GLenum**: The transfer parameter to set. It may be any value from [Table 7.4](#).

param **GLint** or **GLfloat**: The parameter value.

Returns: None.

See Also: [glCopyPixels](#), [glDrawPixels](#), [glPixelMap](#), [glReadPixels](#), [glTexImage1D](#), [glTexImage2D](#)

glPixelZoom

Purpose: Sets the scaling for pixel transfers.

Include File: `<gl.h>`

Syntax:

```
void glPixelZoom(GLfloat xfactor, GLfloat yfactor)
```

Description: This function sets pixel scaling for [glCopyPixels](#), [glDrawPixels](#), [glReadPixels](#), [glTexImage1D](#), and [glTexImage2D](#). Pixels are scaled using the nearest neighbor algorithm when they are read from memory or the framebuffer. In the case of [glCopyPixels](#) and [glDrawPixels](#), the scaled pixels are drawn in the framebuffer at the current raster position.

Parameters:

xfactor **GLfloat**: The horizontal scaling factor (1.0 is no scaling).

yfactor **GLfloat**: The vertical scaling factor (1.0 is no scaling).

Returns: None.

See Also: [glCopyPixels](#), [glDrawPixels](#), [glReadPixels](#), [glTexImage1D](#), [glTexImage2D](#)

glRasterPos

Purpose: Sets the current raster position.

Include File: `<gl.h>`

Variations:

```
void glRasterPos2d(GLdouble x, GLdouble y);
void glRasterPos2dv(GLdouble *v);
void glRasterPos2f(GLfloat x, GLfloat y);
void glRasterPos2fv(GLfloat *v);
void glRasterPos2i(GLint x, GLint y);
void glRasterPos2iv(Glint *v);
void glRasterPos2s(GLshort x, GLshort y);
void glRasterPos2sv(GLshort *v);
void glRasterPos3d(GLdouble x, GLdouble y,
    GLdouble z);
void glRasterPos3dv(GLdouble *v);
void glRasterPos3f(GLfloat x, GLfloat y, GLfloat z);
```

```

void glRasterPos3fv(GLfloat *v);
void glRasterPos3i(GLint x, GLint y, GLint z);
void glRasterPos3iv(GLint *v);
void glRasterPos3s(GLshort x, GLshort y, GLshort z);
void glRasterPos3sv(GLshort *v);
void glRasterPos4d(GLdouble x, GLdouble y,
  GLdouble z, GLdouble w);
void glRasterPos4dv(GLdouble *v);
void glRasterPos4f(GLfloat x, GLfloat y, GLfloat z
  , GLfloat w);
void glRasterPos4fv(GLfloat *v);
void glRasterPos4i(GLint x, GLint y, GLint z,
  GLint w);
void glRasterPos4iv(GLint *v);
void glRasterPos4s(GLshort x, GLshort y, GLshort z
  , GLshort w);
void glRasterPos4sv(GLshort *v);

```

Description: This function sets the current raster position. The coordinates supplied are transformed and projected by the current modelview and projection matrix, resulting in a 2D window position. If the raster position is outside the window boundaries, it is invalid, and no raster operations will occur. The current color for bitmaps (`glBitmap`) is taken from the current color when the raster position is set.

Parameters:

`x, y, z, w` `GLdouble` or `GLfloat` or `GLint` or `GLshort`: The x, y, z, and w coordinates of the raster position.

`v` `GLdouble*` or `GLfloat*` or `GLint*` or `GLshort*`: A pointer to an array containing the coordinates of the raster position.

Returns: None.

See Also: `glWindowPos`

glReadPixels

Purpose: Reads a block of pixels from the framebuffer.

Include File: `<gl.h>`

Syntax:

```

void glReadPixels(GLint x, GLint y, GLsizei width,
  GLsizei height,
  GLenum format, GLenum type,
  const void *pixels);

```

Description: This function copies pixel data from the framebuffer to memory. Besides the *format* and *type* arguments, several other parameters define the encoding of pixel data in memory and control the processing of pixel data before it is placed in the memory buffer. Pixel values are modified according to the settings of the current pixel map, pixel storage mode, pixel transfer operation, and current read buffer.

Parameters:

x **GLint**: The lower-left corner window horizontal coordinate.

y **GLint**: The lower-left corner window vertical coordinate.

width **GLsizei**: The width of the image in pixels.

height **GLsizei**: The height of the image in pixels.

format **GLenum**: The colorspace of the pixels to be read. It may be taken from the constants defined in [Table 7.2](#).

type **GLenum**: The desired data type for the returned pixels. This may be any of the values listed in [Table 7.3](#).

pixels **void ***: A pointer to the pixel data for the image.

Returns: None.

See Also: [glPixelMap](#), [glPixelStore](#), [glPixelTransfer](#), [glReadBuffer](#)

glResetHistogram

Purpose: Clears the histogram buffer.

Include File: [<gl.h>](#)

Syntax:

```
void glResetHistogram(GLenum target);
```

Description: This function clears the histogram buffer.

Parameters:

target **GLenum**: The histogram target. It must be [GL_HISTOGRAM](#).

Returns: None.

See Also: [glGetHistogram](#), [glHistogram](#)

glResetMinmax

Purpose: Resets the min/max color values.

Include File: [<gl.h>](#)

Syntax:

```
void glResetMinMax(GLenum target);
```

Description: This function resets the min/max color values.

Parameters:

target *GLenum*: The min/max target. It must be *GL_MINMAX*.

Returns: None.

See Also: *glGetMinMax*, *glMinMax*

glSeparableFilter2D

Purpose: Defines a two-dimensional separable convolution filter.

Include File: *<gl.h>*

Syntax:

```
void glSeparableFilter2D(GLenum target, GLenum
→  internalFormat,
→           GLsizei width, GLsizei
→  height,
→           GLenum format, GLenum type,
→           void *row, const GLvoid
→  *column);
```

Description: This function specifies a two-dimensional separable convolution filter. This filter is specified with two one-dimensional arrays. The final two-dimensional convolution filter is derived from the outer product of these two arrays.

Parameters:

target *GLenum*: The separable filter target. It must be *GL_SEPARABLE_2D*.

internalformat *GLenum*: The pixel components the filter is to be applied to. Any of the formats listed in [Table 7.2](#) may be specified.

width *GLsizei*: The width of the filter.

height *GLsizei*: The height of the filter.

format *GLenum*: The color format of the filter data. Any format from [Table 7.3](#) may be used.

type *GLenum*: The data type for the filter image values. Any of the data types from [Table 7.3](#) may be used.

row *void *:* The row array data.

column *void *:* The column array data.

Returns: None.

See Also: *glGetSeparableFilter*

glWindowPos**Purpose:** Sets the raster position in window coordinates.**Include File:** `<gl.h>`**Variations:**

```
void glWindowPos2d(GLdouble x, GLdouble y);
void glWindowPos2dv(GLdouble *v);
void glWindowPos2f(GLfloat x, GLfloat y);
void glWindowPos2fv(GLfloat *v);
void glWindowPos2i(GLint x, GLint y);
void glWindowPos2iv(GLint *v);
void glWindowPos2s(GLshort x, GLshort y);
void glWindowPos2sv(GLshort *v);
void glWindowPos3d(GLdouble x, GLdouble y,
➥ GLdouble z);
void glWindowPos3dv(GLdouble *v);
void glWindowPos3f(GLfloat x, GLfloat y, GLfloat z);
void glWindowPos3fv(GLfloat *v);
void glWindowPos3i(GLint x, GLint y, GLint z);
void glWindowPos3iv(GLint *v);
void glWindowPos3s(GLshort x, GLshort y, GLshort z);
void glWindowPos3sv(GLshort *v);
```

Description: This function sets the current raster position in window coordinates. If the specified position falls outside the window, the raster position is invalid, and no raster operations are performed. Unlike `glRasterPos`, this function does not transform the coordinates by the transformation matrices, but allows specification of the raster position directly in window coordinates.**Parameters:****x, y, z** `GLdouble` or `GLfloat` or `GLint` or `GLshort`: The x, y, and z coordinates of the raster position in window coordinates.**v** `GLdouble*` or `GLfloat*` or `GLint*` or `GLshort*`: A pointer to an array containing the coordinates of the raster position.**Returns:** None.**See Also:** `glRasterPos`

Chapter 8. Texture Mapping: The Basics

by Richard S. Wright, Jr.

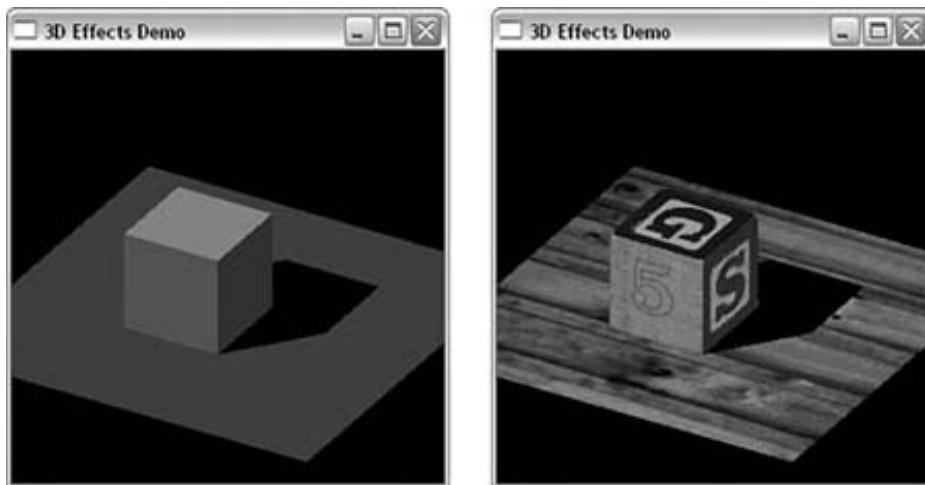
WHAT YOU'LL LEARN IN THIS CHAPTER

How To	Functions You'll Use
Load texture images	<code>glTexImage</code> , <code>glTexSubImage</code>

Map textures to geometry	<code>glTexCoord</code>
Change the texture environment	<code>glTexEnv</code>
Set texture mapping parameters	<code>glTexParameter</code>
Generate mipmaps	<code>gluBuildMipmaps</code>
Manage multiple textures	<code>glBindTexture</code>

In the preceding chapter, we covered in detail the groundwork for loading image data into OpenGL. Image data, unless modified by pixel zoom, generally has a one-to-one correspondence between a pixel in an image to a pixel on the screen. In fact, this is where we get the term *pixel* (picture element). In this chapter, we extend this knowledge further by applying images to three-dimensional primitives. When we apply image data to a geometric primitive, we call this a *texture* or *texture map*. [Figure 8.1](#) shows the dramatic difference that can be achieved by texture mapping geometry. The cube on the left is a lit and shaded featureless surface, whereas the cube on the right shows a richness in detail that can be reasonably achieved only with texture mapping.

Figure 8.1. The stark contrast between textured and untextured geometry.



A texture image when loaded has the same makeup and arrangement as pixmaps, but now a one-to-one correspondence seldom exists between *texels* (the individual picture elements in a texture) and pixels on the screen. This chapter covers the basics of loading a texture map into memory and all the ways in which it may be mapped to and applied to geometric primitives.

Loading Textures

The first necessary step in applying a texture map to geometry is to load the texture into memory. Once loaded, the texture becomes part of the current *texture state* (more on this later). Three OpenGL functions are most often used to load texture data from a memory buffer (which is, for example, read from a disk file):

```
void glTexImage1D(GLenum target, GLint level, GLint internalformat,
                  GLsizei width, GLint border,
                  GLenum format, GLenum type, void *data);
void glTexImage2D(GLenum target, GLint level, GLint internalformat,
                  GLsizei width, GLsizei height, GLint border,
                  GLenum format, GLenum type, void *data);
void glTexImage3D(GLenum target, GLint level, GLint internalformat,
                  GLsizei width, GLsizei height, GLsizei depth, GLint border,
```

```
GLenum format, GLenum type, void *data);
```

These three rather lengthy functions tell OpenGL everything it needs to know about how to interpret the texture data pointed to by the *data* parameter.

The first thing you should notice about these functions is that they are essentially three flavors of the same root function, `glTexImage`. OpenGL supports one-, two-, and three-dimensional texture maps and uses the corresponding function to load that texture and make it current. You should also be aware that OpenGL copies the texture information from *data* when you call one of these functions. This data copy can be quite expensive, and in the section "Texture Objects" we discuss some ways to help mitigate this problem.

The *target* argument for these functions should be `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, or `GL_TEXTURE_3D`, respectively. You may also specify proxy textures in the same manner that you used proxies in the preceding chapter by specifying `GL_PROXY_TEXTURE_1D`, `GL_PROXY_TEXTURE_2D`, or `GL_PROXY_TEXTURE_3D` and using the function `glGetTexParameter` to retrieve the results of the proxy query.

The *level* parameter specifies the mipmap level being loaded. Mipmaps are covered in an upcoming section called "Mipmapping", so for non-mipmapped textures (just your plain old ordinary texture mapping), always set this to 0 (zero) for the moment.

Next, you have to specify the *internalformat* parameter of the texture data. This information tells OpenGL how many color components you want stored per texel and possibly the storage size of the components and/or whether you want the texture compressed (see the next chapter for information about texture compression). Table 8.1 lists the most common values for this function. A complete listing is given in the reference section.

`GL_ALPHA`

Store the texels as alpha values

`GL_LUMINANCE`

Store the texels as luminance values

`GL_LUMINANCE_ALPHA`

Store the texels with both luminance and alpha values

`GL_RGB`

Store the texels as red, green, and blue components

`GL_RGBA`

Store the texels as red, green, blue, and alpha components

Table 8.1. Most Common Texture Internal Formats

Constant	Meaning

The *width*, *height*, and *depth* parameters (where appropriate) specify the dimensions of the texture being loaded. It is important to note that these dimensions must be integer powers of 2 (1, 2, 4, 8, 16, 32, 64, and so on). There is no requirement that texture maps be square (all dimensions equal), but a texture loaded with a nonpower of 2 dimensions will cause texturing to be implicitly disabled.

The *border* parameter allows you to specify a border width for texture maps. Texture borders allow you to extend the width, height, or depth of a texture map by an extra set of texels along the borders. Texture borders play an important role in the discussion of texture filtering to come. For the time being, always set this value to 0 (zero).

The last three parameters—*format*, *type*, and *data*—are identical to the corresponding arguments when you used `glDrawPixels` to place image data into the color buffer. For the sake of convenience, we listed the valid constants for *format* and *type* in Tables 8.2 and 8.3.

`GL_RGB`

Colors are in red, green, blue order.

`GL_RGBA`

Colors are in red, green, blue, alpha order.

`GL_BGR /GL_BGR_EXT`

Colors are in blue, green, red order.

`GL_BGRA /GL_BGRA_EXT`

Colors are in blue, green, red, alpha order.

`GL_RED`

Each pixel contains a single red component.

`GL_GREEN`

Each pixel contains a single green component.

`GL_BLUE`

Each pixel contains a single blue component.

`GL_ALPHA`

Each pixel contains a single alpha component.

`GL_LUMINANCE`

Each pixel contains a single luminance (intensity) component.

GL_LUMINANCE_ALPHA

Each pixel contains a luminance followed by an alpha component.

GL_STENCIL_INDEX

Each pixel contains a single stencil index.

GL_DEPTH_COMPONENT

Each pixel contains a single depth component.

Table 8.2. Texel Formats for `glTexImage`

Constant	Description

GL_UNSIGNED_BYTE

Each color component is an 8-bit unsigned integer

GL_BYTE

Signed 8-bit integer

GL_BITMAP

Single bits, no color data; same as `glBitmap`

GL_UNSIGNED_SHORT

Unsigned 16-bit integer

GL_SHORT

Signed 16-bit integer

GL_UNSIGNED_INT

Unsigned 32-bit integer

GL_INT

Signed 32-bit integer

GL_FLOAT

Single precision float

`GL_UNSIGNED_BYTE_3_2_2`

Packed RGB values

`GL_UNSIGNED_BYTE_2_3_3_REV`

Packed RGB values

`GL_UNSIGNED_SHORT_5_6_5`

Packed RGB values

`GL_UNSIGNED_SHORT_5_6_5_REV`

Packed RGB values

`GL_UNSIGNED_SHORT_4_4_4_4`

Packed RGBA values

`GL_UNSIGNED_SHORT_4_4_4_4_REV`

Packed RGBA values

`GL_UNSIGNED_SHORT_5_5_5_1`

Packed RGBA values

`GL_UNSIGNED_SHORT_1_5_5_5_REV`

Packed RGBA values

`GL_UNSIGNED_INT_8_8_8_8`

Packed RGBA values

`GL_UNSIGNED_INT_8_8_8_8_REV`

Packed RGBA values

`GL_UNSIGNED_INT_10_10_10_2`

Packed RGBA values

`GL_UNSIGNED_INT_2_10_10_10_REV`

Packed RGBA values

**Table 8.3. Data Types
for Pixel Data**

Constant	Description

Loaded textures are not applied to geometry unless the appropriate texture state is enabled. You can call `glEnable` or `glDisable` with `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, or `GL_TEXTURE_3D` to turn texturing on or off for a given texture state. Only one of these texture states may be on at a time for a given texture unit (see the next chapter for a discussion of multitexturing). However, it is good practice to turn unused texture states off when not in use. For example, to switch between one-dimensional and two-dimensional texturing, you would call

```
glDisable(GL_TEXTURE_1D);
glEnable(GL_TEXTURE_2D);
```

A final word about texture loading: Texture data loaded by the `glTexImage` functions goes through the same pixel and imaging pipeline covered in the preceding chapter. This means pixel packing, pixelzoom, color tables, convolutions, and so on are applied to the texture data when it is loaded.

Using the Color Buffer

One- and two-dimensional textures may also be loaded using data from the color buffer. You can read an image from the color buffer and use it as a new texture by using the following two functions:

```
void glCopyTexImage1D(GLenum target, GLint level, GLenum internalformat,
                      GLint x, GLint y,
                      GLsizei width, GLint border);
void glCopyTexImage2D(GLenum target, GLint level, GLenum internalformat,
                      GLint x, GLint y,
                      GLsizei width, GLsizei height, GLint border);
```

These functions operate similarly to `glTexImage`, but in this case, `x` and `y` specify the location in the color buffer to begin reading the texture data. The source buffer is set using `glReadBuffer` and behaves just like `glReadPixels`.

Updating Textures

Repeatedly loading new textures can become a performance bottleneck in time-sensitive applications such as games or simulation applications. If a loaded texture map is no longer needed, it may be replaced entirely or in part. Replacing a texture map can often be done much quicker than reloading a new texture directly with `glTexImage`. The function you use to accomplish this is `glTexSubImage`, again in three variations:

```
void glTexSubImage1D(GLenum target, GLint level,
                      GLint xOffset,
                      GLsizei width,
                      GLenum format, GLenum type, const GLvoid *data);
void glTexSubImage2D(GLenum target, GLint level,
                      GLint xOffset, GLint yOffset,
                      GLsizei width, GLsizei height,
                      GLenum format, GLenum type, const GLvoid *data);
void glTexSubImage3D(GLenum target, GLint level,
                      GLint xOffset, GLint yOffset, GLint zOffset,
```

```
GLsizei width, GLsizei height, GLsizei depth,
GLenum format, GLenum type, const GLvoid *data);
```

Most of the arguments correspond exactly to the parameters used in `glTexImage`. The `xOffset`, `yOffset`, and `zOffset` parameters specify the offsets into the existing texture map to begin replacing texture data. The `width`, `height`, and `depth` values specify the dimensions of the texture being "inserted" into the existing texture.

A final set of functions allows you to combine reading from the color buffer and inserting or replacing part of a texture. These `glCopyTexSubImage` variations do just that:

```
void glCopyTexSubImage1D(GLenum target, GLint level,
                         GLint xoffset,
                         GLint x, GLint y,
                         GLsizei width);
void glCopyTexSubImage2D(GLenum target, GLint level,
                         GLint xoffset, GLint yoffset,
                         GLint x, GLint y,
                         GLsizei width, GLsizei height);
void glCopyTexSubImage3D(GLenum target, GLint level,
                         GLint xoffset, GLint yoffset, GLint zoffset,
                         GLint x, GLint y,
                         GLsizei width, GLsizei height);
```

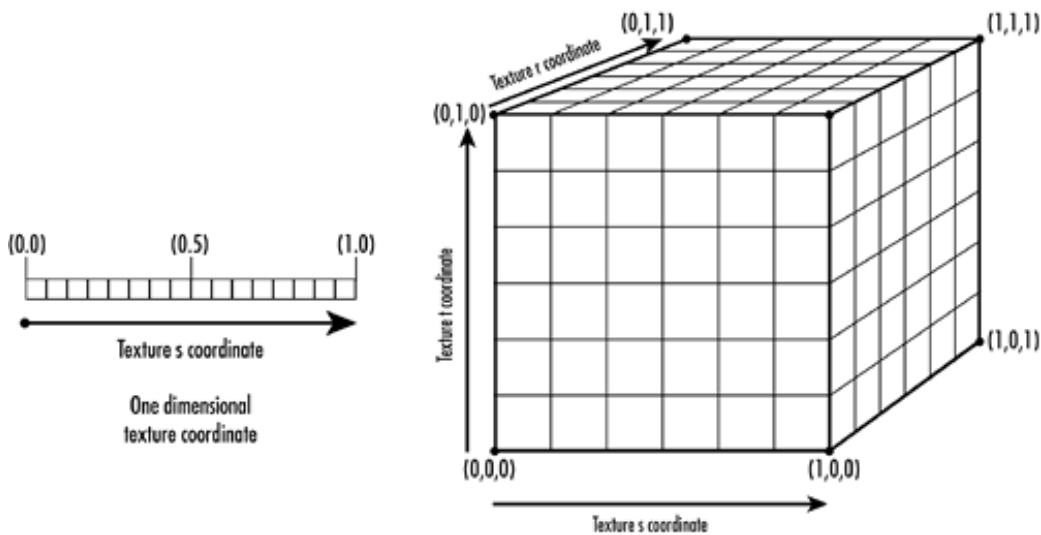
You may have noticed that no `glCopyTexImage3D` function is listed here. The reason is that the color buffer is 2D, and there simply is no corresponding way to use a 2D color image as a source for a 3D texture. However, you can use `glCopyTexSubImage3D` to use the color buffer data to set a plane of texels in a three-dimensional texture.

Mapping Textures to Geometry

Loading a texture and enabling texturing cause OpenGL to apply the texture to any of the OpenGL primitives. You must, however, provide OpenGL with information about how to map the texture to the geometry. You do this by specifying a *texture coordinate* for each vertex. Texels in a texture map are addressed not as a memory location (as you would for pixmaps), but as a more abstract (usually floating-point values) texture coordinate. Typically, texture coordinates are specified as floating-point values that are clamped in the range 0.0 to 1.0. Texture coordinates are named *s*, *t*, *r*, and *q* (similar to vertex coordinates *x*, *y*, *z*, and *w*) supporting from one- to three-dimensional texture coordinates, and optionally a way to scale the coordinates.

[Figure 8.2](#) shows one-, two-, and three-dimensional textures and the way the texture coordinates are laid out with respect to their texels.

Figure 8.2. How texture coordinates address texels.



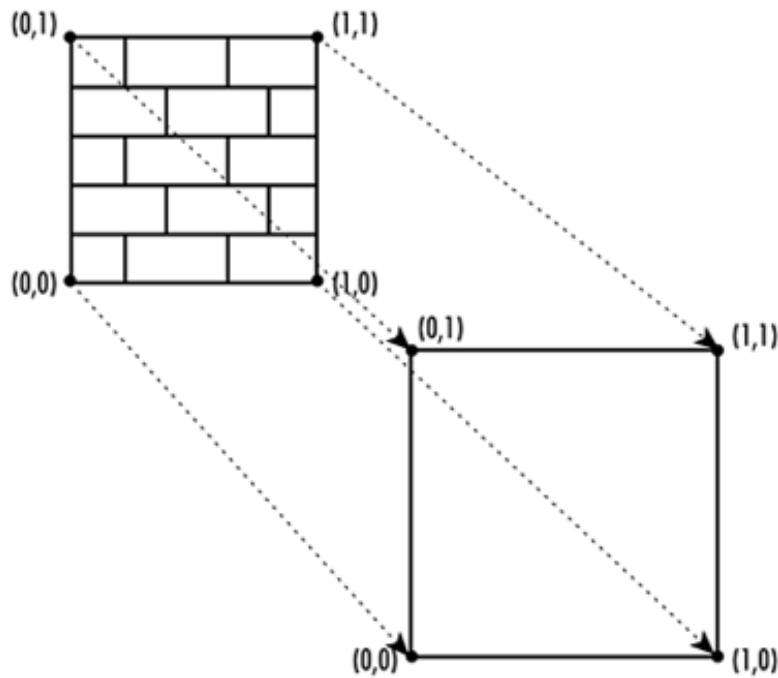
Because there are no four-dimensional textures, you might ask what the q coordinate is for. The q coordinate corresponds to the w geometric coordinate. This is a scaling factor applied to the other texture coordinates; that is, the actual values used for the texture coordinates are s/q , t/q , and r/q . By default, q is set to 1.0.

You specify a texture coordinate using the `glTexCoord` function. Much like vertex coordinates, surface normals, and color values, this function comes in a variety of familiar flavors that are all listed in the reference section. The following are three simple variations used in the sample programs:

```
void glTexCoord1f(GLfloat s);
void glTexCoord2f(GLfloat s, GLfloat t);
void glTexCoord3f(GLfloat s, GLfloat t, GLfloat r);
```

One texture coordinate is applied using these functions for each vertex. OpenGL then stretches or shrinks the texture as necessary to apply the texture to the geometry as mapped. (This stretching or shrinking is applied using the current texture *filter*; we discuss this issue shortly as well.) [Figure 8.3](#) shows an example of a two-dimensional texture being mapped to a `GL_QUAD`. Note the corners of the texture correspond to the corners of the quad. As you do with other vertex properties (materials, normals, and so on), you must specify the texture coordinate before the vertex!

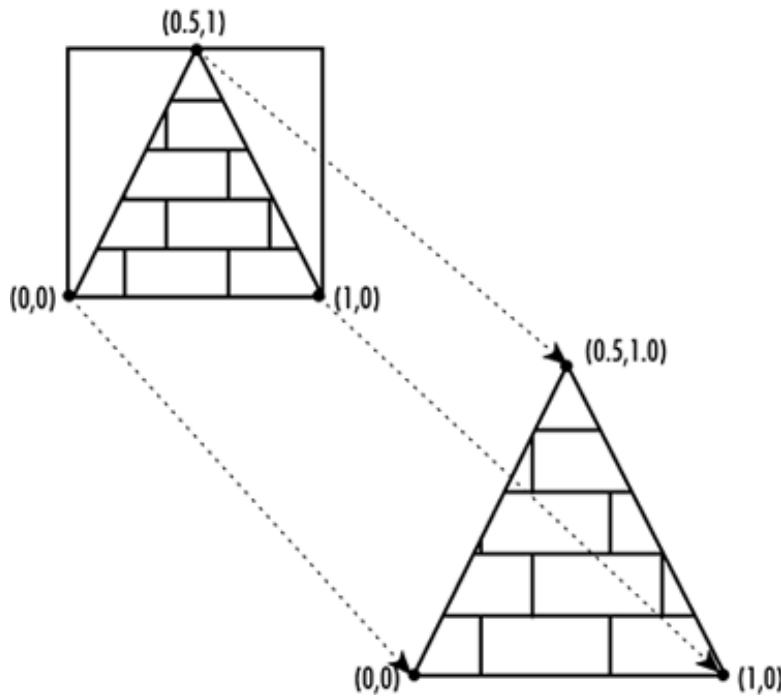
Figure 8.3. Applying a two-dimensional texture to a quad.



Rarely, however, do you have such a nice fit of a square texture mapped to a square piece of geometry. To help you better understand texture coordinates, we provide another example in [Figure 8.4](#). This figure also shows a square texture map, but the geometry is a triangle.

Superimposed on the texture map are the texture coordinates of the locations in the map being extended to the vertices of the triangle.

Figure 8.4. Applying a portion of a texture map to a triangle.



Texture Matrix

Texture coordinates can also be transformed via the texture matrix. The texture matrix stack works just like the other matrices previously discussed (modelview, projection, and color). You make the texture matrix the target of matrix function calls by calling `glMatrixMode` with the argument `GL_TEXTURE`:

```
glMatrixMode(GL_TEXTURE);
```

The texture matrix stack is required to be only two elements deep for the purposes of `glPushMatrix` and `glPopMatrix`. Texture coordinates can be translated, scaled, and even rotated.

A Simple 2D Example

Loading a texture and providing texture coordinates are the fundamental requirements for texture mapping. There are a number of issues we have yet to address, such as coordinate wrapping, texture filters, and the texture environment. What do they mean, and how do we make use of them? Let's pause here first and examine a simple example that uses a 2D texture. In the code that follows, we use the functions covered so far and add several new ones. We use this example, then, as a framework to describe these additional texture mapping issues.

[Listing 8.1](#) shows all the code for the sample program PYRAMID. This program draws a simple lit four-sided pyramid made up of triangles. A stone texture is applied to each face and the bottom of the pyramid. You can spin the pyramid around with the arrow keys much like the samples in earlier chapters. [Figure 8.5](#) shows the output of the PYRAMID program.

Listing 8.1. The PYRAMID Sample Program Source Code

```
// Pyramid.c
// Demonstrates Simple Texture Mapping
// OpenGL SuperBible
// Richard S. Wright Jr.
#include "../../../../Common/OpenGLSB.h" // System and OpenGL Stuff
#include "../../../../Common/GLTools.h" // GLTools
// Rotation amounts
static GLfloat xRot = 0.0f;
static GLfloat yRot = 0.0f;
// Change viewing volume and viewport. Called when window is resized
void ChangeSize(int w, int h)
{
    GLfloat fAspect;
    // Prevent a divide by zero
    if(h == 0)
        h = 1;
    // Set Viewport to window dimensions
    glViewport(0, 0, w, h);
    fAspect = (GLfloat)w/(GLfloat)h;
    // Reset coordinate system
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    // Produce the perspective projection
    gluPerspective(35.0f, fAspect, 1.0, 40.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

// This function does any needed initialization on the rendering
// context. Here it sets up and initializes the lighting for
// the scene.
void SetupRC()
{
    GLubyte *pBytes;
    GLint iWidth, iHeight, iComponents;
    GLenum eFormat;
    // Light values and coordinates
    GLfloat whiteLight[] = { 0.05f, 0.05f, 0.05f, 1.0f };
```

```

GLfloat  sourceLight[] = { 0.25f, 0.25f, 0.25f, 1.0f };
GLfloat  lightPos[] = { -10.f, 5.0f, 5.0f, 1.0f };
glEnable(GL_DEPTH_TEST); // Hidden surface removal
glFrontFace(GL_CCW); // Counter clock-wise polygons face out
glEnable(GL_CULL_FACE); // Do not calculate inside of jet
// Enable lighting
glEnable(GL_LIGHTING);
// Setup and enable light 0
glLightModelfv(GL_LIGHT_MODEL_AMBIENT,whiteLight);
glLightfv(GL_LIGHT0,GL_AMBIENT,sourceLight);
glLightfv(GL_LIGHT0,GL_DIFFUSE,sourceLight);
glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
glEnable(GL_LIGHT0);
// Enable color tracking
glEnable(GL_COLOR_MATERIAL);
// Set Material properties to follow glColor values
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
// Black blue background
glClearColor(0.0f, 0.0f, 0.0f, 1.0f );
// Load texture
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
pBytes = gltLoadTGA("Stone.tga", &iWidth, &iHeight,
                     &iComponents, &eFormat);
glTexImage2D(GL_TEXTURE_2D, 0, iComponents, iWidth, iHeight,
             0, eFormat, GL_UNSIGNED_BYTE, pBytes);
free(pBytes);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
glEnable(GL_TEXTURE_2D);
}

// Respond to arrow keys
void SpecialKeys(int key, int x, int y)
{
    if(key == GLUT_KEY_UP)
        xRot-= 5.0f;
    if(key == GLUT_KEY_DOWN)
        xRot += 5.0f;
    if(key == GLUT_KEY_LEFT)
        yRot -= 5.0f;
    if(key == GLUT_KEY_RIGHT)
        yRot += 5.0f;
    xRot = (GLfloat)((const int)xRot % 360);
    yRot = (GLfloat)((const int)yRot % 360);
    // Refresh the Window
    glutPostRedisplay();
}

// Called to draw scene
void RenderScene(void)
{
    GLTVector3 vNormal;
    GLTVector3 vCorners[5] = { { 0.0f, .80f, 0.0f }, // Top 0
                             { -0.5f, 0.0f, -.50f }, // Back left 1
                             { 0.5f, 0.0f, -.50f }, // Back right 2
                             { 0.5f, 0.0f, 0.5f }, // Front right 3
                             { -0.5f, 0.0f, 0.5f } }; // Front left 4
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // Save the matrix state and do the rotations
}

```

```

glPushMatrix();
    // Move object back and do in place rotation
    glTranslatef(0.0f, -0.25f, -4.0f);
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);
    // Draw the Pyramid
    glColor3f(1.0f, 1.0f, 1.0f);
    glBegin(GL_TRIANGLES);
        // Bottom section - two triangles
        glNormal3f(0.0f, -1.0f, 0.0f);
        glTexCoord2f(1.0f, 1.0f);
        glVertex3fv(vCorners[2]);
        glTexCoord2f(0.0f, 0.0f);
        glVertex3fv(vCorners[4]);
        glTexCoord2f(0.0f, 1.0f);
        glVertex3fv(vCorners[1]);
        glTexCoord2f(1.0f, 1.0f);
        glVertex3fv(vCorners[2]);

        glTexCoord2f(1.0f, 0.0f);
        glVertex3fv(vCorners[3]);
        glTexCoord2f(0.0f, 0.0f);
        glVertex3fv(vCorners[4]);
        // Front Face
        gltGetNormalVector(vCorners[0], vCorners[4], vCorners[3], vNormal);
        glNormal3fv(vNormal);
        glTexCoord2f(0.5f, 1.0f);
        glVertex3fv(vCorners[0]);
        glTexCoord2f(0.0f, 0.0f);
        glVertex3fv(vCorners[4]);
        glTexCoord2f(1.0f, 0.0f);
        glVertex3fv(vCorners[3]);
        // Left Face
        gltGetNormalVector(vCorners[0], vCorners[1], vCorners[4], vNormal);
        glNormal3fv(vNormal);
        glTexCoord2f(0.5f, 1.0f);
        glVertex3fv(vCorners[0]);
        glTexCoord2f(0.0f, 0.0f);
        glVertex3fv(vCorners[1]);
        glTexCoord2f(1.0f, 0.0f);
        glVertex3fv(vCorners[4]);
        // Back Face
        gltGetNormalVector(vCorners[0], vCorners[2], vCorners[1], vNormal);
        glNormal3fv(vNormal);
        glTexCoord2f(0.5f, 1.0f);
        glVertex3fv(vCorners[0]);
        glTexCoord2f(0.0f, 0.0f);
        glVertex3fv(vCorners[2]);
        glTexCoord2f(1.0f, 0.0f);
        glVertex3fv(vCorners[1]);
        // Right Face
        gltGetNormalVector(vCorners[0], vCorners[3], vCorners[2], vNormal);
        glNormal3fv(vNormal);
        glTexCoord2f(0.5f, 1.0f);
        glVertex3fv(vCorners[0]);
        glTexCoord2f(0.0f, 0.0f);
        glVertex3fv(vCorners[3]);
        glTexCoord2f(1.0f, 0.0f);
        glVertex3fv(vCorners[2]);
    glEnd();
    // Restore the matrix state

```

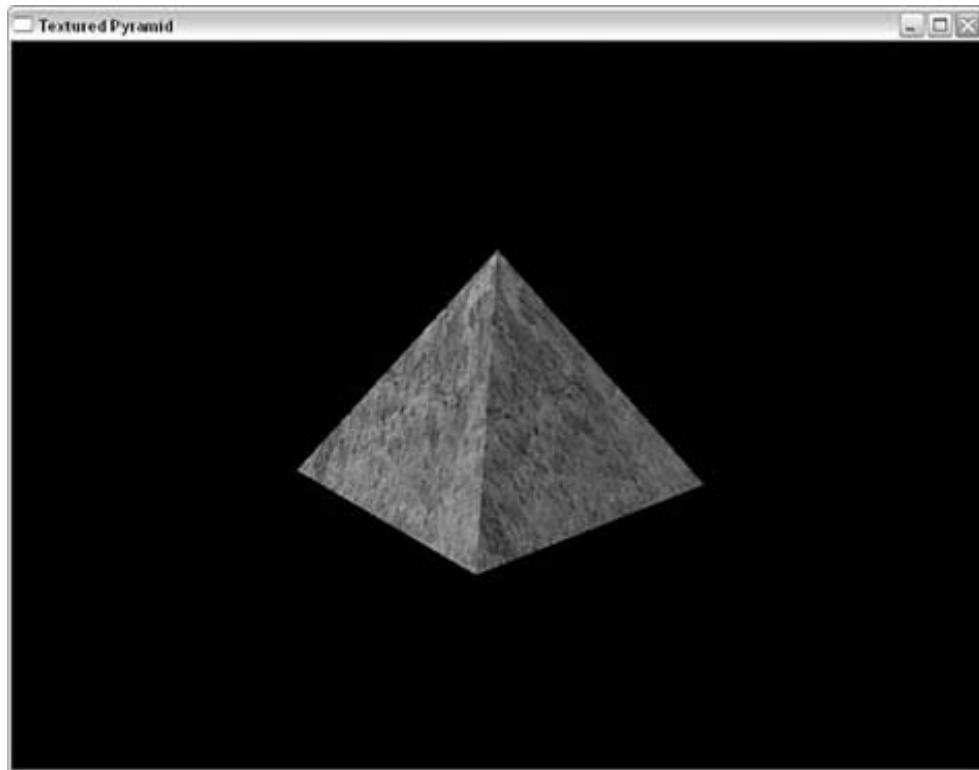
```

glPopMatrix();
// Buffer swap
glutSwapBuffers();
}

int main(int argc, char *argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(800, 600);
    glutCreateWindow("Textured Pyramid");
    glutReshapeFunc(ChangeSize);
    glutSpecialFunc(SpecialKeys);
    glutDisplayFunc(RenderScene);
    SetupRC();
    glutMainLoop();
    return 0;
}

```

Figure 8.5. Output from the PYRAMID sample program.



The `SetupRC` function does all the necessary initialization for this program, including loading the texture using the `gltLoadTGA` function presented in the preceding chapter and supplying the bits to the `glTexImage2D` function:

```

// Load texture
pBytes = gltLoadTGA("Stone.tga", &iWidth, &iHeight,
                     &iComponents, &eFormat);
glTexImage2D(GL_TEXTURE_2D, 0, iComponents, iWidth, iHeight,
             0, eFormat, GL_UNSIGNED_BYTE, pBytes);
free(pBytes);

```

Of course, texture mapping must also be turned on:

```
glEnable(GL_TEXTURE_2D);
```

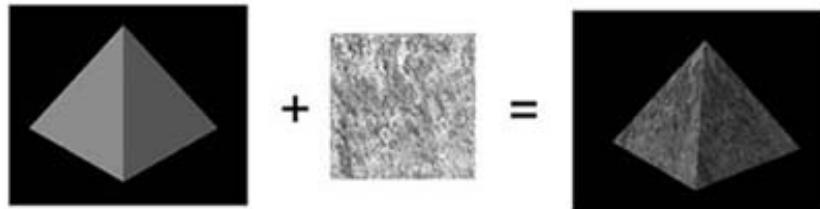
The `RenderScene` function draws the pyramid as a series of texture-mapped triangles. The following excerpt shows one face being constructed as a normal (calculated using the corner vertices) is specified for the face, followed by three texture and vertex coordinates:

```
// Front Face
glGetNormalVector(vCorners[0], vCorners[4], vCorners[3], vNormal);
glNormal3fv(vNormal);
glTexCoord2f(0.5f, 1.0f);
glVertex3fv(vCorners[0]);
glTexCoord2f(0.0f, 0.0f);
glVertex3fv(vCorners[4]);
glTexCoord2f(1.0f, 0.0f);
glVertex3fv(vCorners[3]);
```

Texture Environment

In the PYRAMID sample program, the pyramid is drawn with white material properties, and the texture is applied in such a way that its colors are scaled by the coloring of the lit geometry. [Figure 8.6](#) shows the untextured pyramid alongside the source texture and the textured but shaded pyramid.

Figure 8.6. Lit Geometry + Texture = Shaded Texture.



How OpenGL combines the colors from texels with the color of the underlying geometry is controlled by the *texture environment mode*. You set this mode by calling the `glTexEnv` function:

```
void glTexEnvi(GLenum target, GLenum pname, GLint param);
void glTexEnvf(GLenum target, GLenum pname, GLfloat param);
void glTexEnviv(GLenum target, GLenum pname, GLint *param);
void glTexEnvfv(GLenum target, GLenum pname, GLfloat *param);
```

This function comes in a variety of flavors, as shown here, and controls a number of more advanced texturing options covered in the next chapter. In the PYRAMID sample program, this function set the environment mode to `GL_MODULATE` before any texture application was performed:

```
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

The modulate environment mode multiplies the texel color by the geometry color (after lighting calculations). This is why the shaded color of the pyramid comes through and makes the texture appear to be shaded. Using this mode, you can change the color tone of textures by using colored geometry. For example, a black-and-white brick texture applied to red, yellow, and brown geometry would yield red, yellow, and brown bricks with only a single texture.

If you want to simply replace the color of the underlying geometry, you can specify `GL_REPLACE` for the environment mode. Doing so replaces fragment colors from the geometry directly with the texel colors. Making this change eliminates any effect on the texture from the underlying geometry. If the texture has an alpha channel, you can enable blending, and you can use this mode to create transparent geometry patterned after the alpha channel in the texture map.

If the texture doesn't have an alpha component, `GL_DECAL` behaves the same way as `GL_REPLACE`. It simply "decals" the texture over the top of the geometry and any color values that have been calculated for the fragments. However, if the texture has an alpha component, the decal can be applied in such a way that the geometry shows through where the alpha is blended with the underlying fragments.

Textures can also be blended with a constant blending color using the `GL_BLEND` texture environment. If you set this environment mode, you must also set the texture environment color:

```
GLfloat fColor[4] = { 1.0f, 0.0f, 0.0f, 0.0f };
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_BLEND);
glTexEnvfv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR, fColor);
```

Finally, you can simply add texel color values to the underlying fragment values by setting the environment mode to `GL_ADD`. Any color component values that exceed 1.0 are clamped, and you may get saturated color values (basically, white or closer to white than you might intend).

We have not presented an exhaustive list of the texture environment constants here. See the reference section and next chapter for more modes and texturing effects that are enabled and controlled through this function. We also revisit some additional uses in coming sections and sample programs.

Texture Parameters

More effort is involved in texture mapping than slapping an image on the side of a triangle. Many parameters affect the rendering rules and behaviors of texture maps as they are applied. These texture parameters are all set via variations on the function `glTexParameter`:

```
void glTexParameterf(GLenum target, GLenum pname, GLfloat param);
void glTexParameteri(GLenum target, GLenum pname, GLint param);
void glTexParameterfv(GLenum target, GLenum pname, GLfloat *params);
void glTexParameteriv(GLenum target, GLenum pname, GLint *params);
```

The first argument, `target`, specifies which texture mode the parameter is to be applied to and may be either `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, or `GL_TEXTURE_3D`. The second argument, `pname`, specifies which texture parameter is being set, and finally, the `param` or `params` arguments set the value of the particular texture parameter.

Basic Filtering

Unlike pixmaps being drawn to the color buffer, when a texture is applied to geometry, there is almost never a one-to-one correspondence between texels in the texture map and pixels on the screen. A careful programmer could achieve this result, but only by texturing geometry that was carefully planned to appear onscreen such that the texels and pixels lined up. Consequently, texture images are always either stretched or shrunk as they are applied to geometric surfaces. Because of the orientation of the geometry, a given texture could even be stretched and shrunk at the same time across the surface of some object.

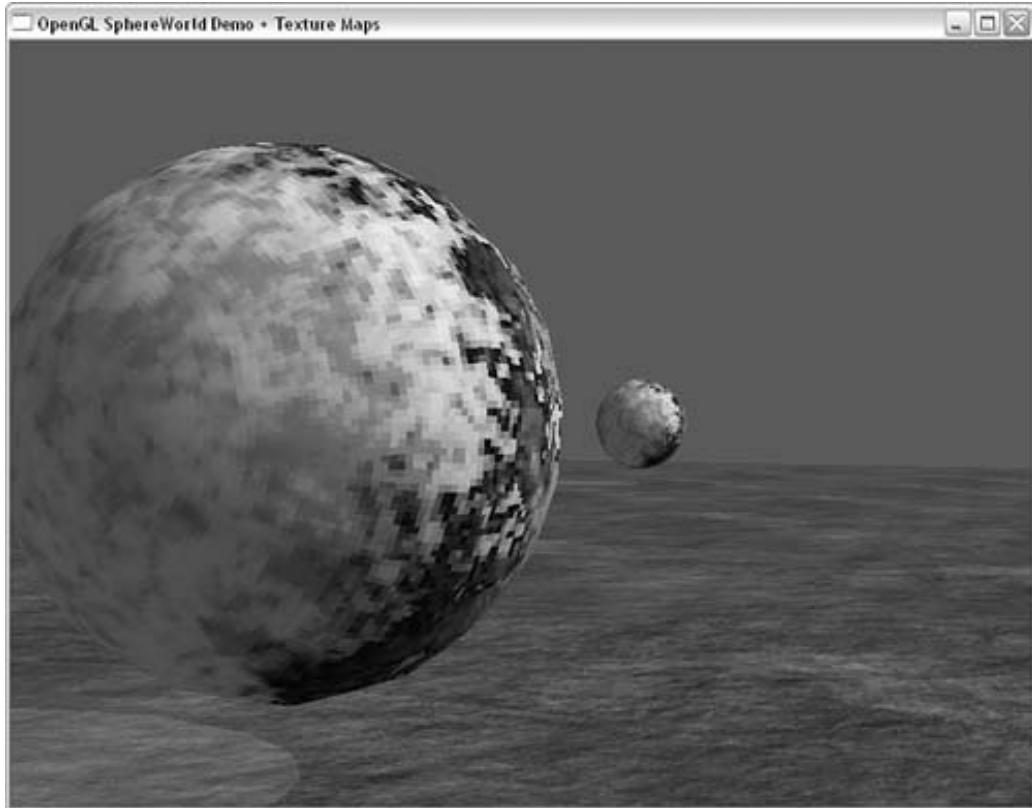
The process of calculating color fragments from a stretched or shrunken texture map is called texture *filtering*. Using the texture parameter function, OpenGL allows you to set both *magnification* and *minification* filters. The parameter names for these two filters are `GL_TEXTURE_MAG_FILTER` and `GL_TEXTURE_MIN_FILTER`. You can select two basic texture filters for them, `GL_NEAREST` and `GL_LINEAR`, which correspond to nearest neighbor and linear filtering.

Nearest neighbor filtering is the simplest and fastest filtering method you can choose. Texture coordinates are evaluated and plotted against a texture's texels, and whichever texel the

coordinate falls in, that color is used for the fragment texture color. Nearest neighbor filtering is characterized by large blocky pixels when the texture is stretched especially large. An example is shown in [Figure 8.7](#). You can set the texture filter (for `GL_TEXTURE_2D`) for both the minification and magnification filter by using these two function calls:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

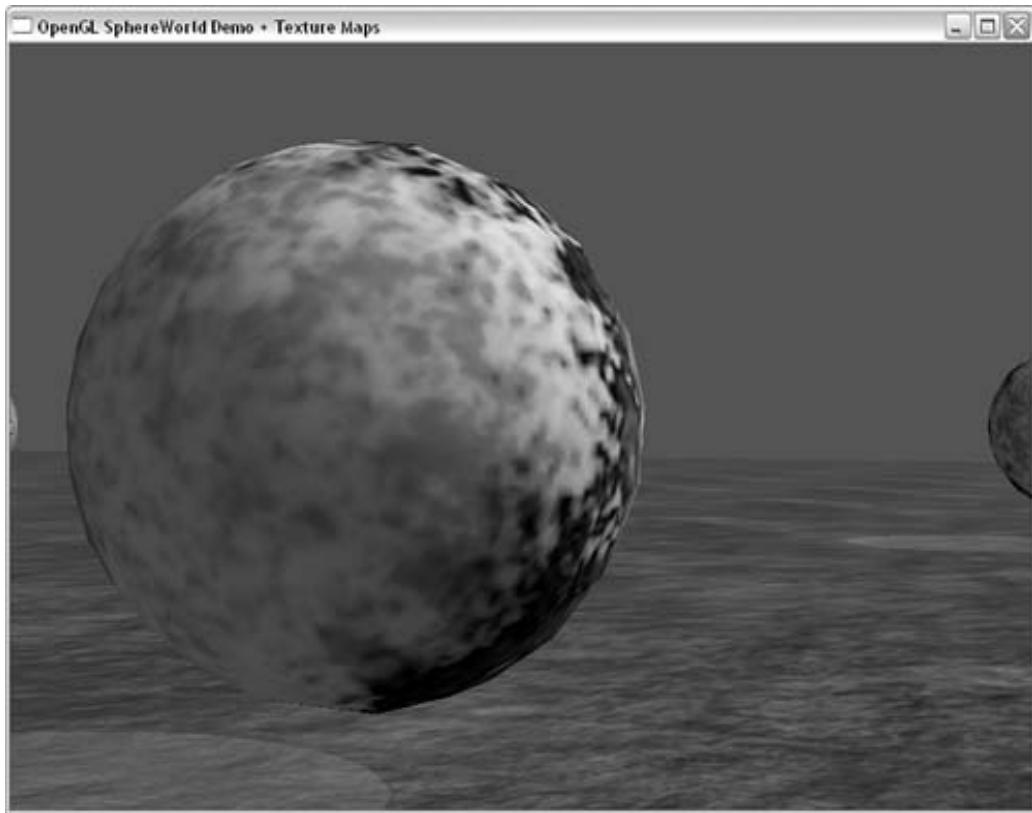
Figure 8.7. Nearest neighbor filtering up close.



Linear filtering requires more work than nearest neighbor, but often is worth the extra overhead. On today's commodity hardware, the extra cost of linear filtering is negligible. Linear filtering works by not taking the nearest texel to the texture coordinate, but by applying the weighted average of the texels surrounding the texture coordinate (a linear interpolation). For this interpolated fragment to match the texel color exactly, the texture coordinate needs to fall directly in the center of the texel. Linear filtering is characterized by "fuzzy" graphics when a texture is stretched. This fuzziness, however, often lends to a more realistic and less artificial look than the jagged blocks of the nearest neighbor filtering mode. A contrasting example to [Figure 8.7](#) is shown in [Figure 8.8](#). You can set linear filtering (for `GL_TEXTURE_2D`) simply enough by using the following lines, which are also included in the `SetupRC` function in the PYRAMID example:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

Figure 8.8. Linear filtering up close.



Texture Wrap

Normally, you specify texture coordinates between 0.0 and 1.0 to map out the texels in a texture map. If texture coordinates fall outside this range, OpenGL handles them according to the current texture wrapping mode. You can set the wrap mode for each coordinate individually by calling `glTexParameter` with `GL_TEXTURE_WRAP_S`, `GL_TEXTURE_WRAP_T`, or `GL_TEXTURE_WRAP_R` as the parameter name. The wrap mode can then be set to one of the following values: `GL_REPEAT`, `GL_CLAMP`, `GL_CLAMP_TO_EDGE`, or `GL_CLAMP_TO_BORDER`.

The `GL_REPEAT` wrap mode simply causes the texture to repeat in the direction in which the texture coordinate has exceeded 1.0. The texture repeats again for every integer texture coordinate. This mode is very useful for applying a small tiled texture to large geometric surfaces. Well-done seamless textures can lend the appearance of a seemingly much larger texture, but at the cost of a much smaller texture image. The other modes do not repeat, but are "clamped"—thus their name.

If the only implication of the wrap mode is whether the texture repeats, you would need only two wrap modes: repeat and clamp. However, the texture wrap mode also has a great deal of influence on how texture filtering is done at the edges of the texture maps. For `GL_NEAREST` filtering, there are no consequences to the wrap mode because the texture coordinates are always snapped to some particular texel within the texture map. However, the `GL_LINEAR` filter takes an average of the pixels surrounding the evaluated texture coordinate, and this creates a problem for texels that lie along the edges of the texture map.

This problem is resolved quite neatly when the wrap mode is `GL_REPEAT`. The texel samples are simply taken from the next row or column, which in repeat mode wraps back around to the other side of the texture. This mode works perfectly for textures that wrap around an object and meet on the other side (such as spheres).

The clamped texture wrap modes offer a number of options for the way texture edges are handled. For `GL_CLAMP`, the needed texels are taken from the texture border or the `TEXTURE_BORDER_COLOR` (set with `glTexParameterfv`). The `GL_CLAMP_TO_EDGE` wrap mode simply

ignores texel samples that go over the edge and does not include them in the average. Finally, `GL_CLAMP_TO_BORDER` uses only border texels whenever the texture coordinates fall outside the range 0.0 to 1.0.

A typical application of the clamped modes occurs when you must texture a large area that would require a single texture too large to fit into memory, or that may be loaded into a single texture map. In this case, the area is chopped up into smaller "tiles" that are then placed side by side. In such a case, not using a wrap mode such as `GL_CLAMP_TO_EDGE` causes visible filtering artifacts along the seams between tiles.

In the PYRAMID sample program, texture coordinates along the bottom of the pyramid might leave a dark seam, except that you have set the wrap mode to clamp the bilinear filtering to within the texture map:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

Cartoons with Texture

The first example for this chapter used 2D textures because they are usually the simplest and easiest to understand. Most people can quickly get an intuitive feel for putting a 2D picture on the side of a piece of 2D geometry (such as a triangle). We will step back now and present a one-dimensional texture mapping example that is commonly used in computer games to render geometry that appears onscreen shaded like a cartoon. *Toon-shading*, which is often referred to as *cell-shading*, uses a one-dimensional texture map as a lookup table to fill in geometry with a solid color (using `GL_NEAREST`) from the texture map.

The basic idea is to use a surface normal from the geometry and a normal from the light source to find the intensity of the light striking the surface of the model. The dot product of these two vectors gives a value between 0.0 and 1.0 and is used as a one-dimensional texture coordinate. The sample program TOON presented in [Listing 8.2](#) draws a green torus using this technique. The output from TOON is shown in [Figure 8.9](#).

Listing 8.2. Source Code for the TOON Sample Program

```
// Toon.c
// OpenGL SuperBible
// Demonstrates Cell/Toon shading with a 1D texture
// Program by Richard S. Wright Jr.
#include "../../../Common/OpenGLSB.h"      // System and OpenGL Stuff
#include "../../../Common/GLTools.h"        // OpenGL toolkit
#include <math.h>
// Draw a torus (doughnut), using the current 1D texture for light shading
void toonDrawTorus(GLfloat majorRadius, GLfloat minorRadius,
                   int numMajor, int numMinor, GLTVector3 vLightDir)
{
    GLTMatrix mModelViewMatrix;
    GLTVector3 vNormal, vTransformedNormal;
    double majorStep = 2.0f*GLT_PI / numMajor;
    double minorStep = 2.0f*GLT_PI / numMinor;
    int i, j;
    // Get the modelview matrix
    glGetFloatv(GL_MODELVIEW_MATRIX, mModelViewMatrix);
    // Normalize the light vector
    gltNormalizeVector(vLightDir);
    // Draw torus as a series of triangle strips
    for (i=0; i<numMajor; ++i)
    {
        double a0 = i * majorStep;
        double a1 = a0 + majorStep;
        // Draw a strip of triangles
        for (j=0; j<numMinor; ++j)
        {
            double b0 = j * minorStep;
            double b1 = b0 + minorStep;
            // Calculate the texture coordinate
            GLfloat t = (b0 + 0.5f) / numMinor;
            // Calculate the surface normal
            vNormal.x = sin(a0) * cos(b0);
            vNormal.y = sin(a0) * sin(b0);
            vNormal.z = cos(a0);
            // Transform the normal
            vTransformedNormal = gltVector3Normal(mModelViewMatrix, vNormal);
            // Calculate the light intensity
            GLfloat intensity = gltVector3DotProduct(vTransformedNormal, vLightDir);
            // Map the intensity to a color
            GLfloat color = intensity * 255.0f;
            // Draw the triangle
            gltDrawTriangle(majorRadius, minorRadius, color);
        }
    }
}
```

```

GLfloat x0 = (GLfloat) cos(a0);
GLfloat y0 = (GLfloat) sin(a0);
GLfloat x1 = (GLfloat) cos(a1);
GLfloat y1 = (GLfloat) sin(a1);
glBegin(GL_TRIANGLE_STRIP);
for (j=0; j<=numMinor; ++j)
{
    {
        double b = j * minorStep;
        GLfloat c = (GLfloat) cos(b);
        GLfloat r = minorRadius * c + majorRadius;
        GLfloat z = minorRadius * (GLfloat) sin(b);
        // First point
        vNormal[0] = x0*c;
        vNormal[1] = y0*c;
        vNormal[2] = z/minorRadius;
        gltNormalizeVector(vNormal);
        gltRotateVector(vNormal, mModelViewMatrix, vTransformedNormal);
        // Texture coordinate is set by intensity of light
        glTexCoord1f(gltVectorDotProduct(vLightDir, vTransformedNormal));
        glVertex3f(x0*r, y0*r, z);
        // Second point
        vNormal[0] = x1*c;
        vNormal[1] = y1*c;
        vNormal[2] = z/minorRadius;
        gltNormalizeVector(vNormal);
        gltRotateVector(vNormal, mModelViewMatrix, vTransformedNormal);
        // Texture coordinate is set by intensity of light
        glTexCoord1f(gltVectorDotProduct(vLightDir, vTransformedNormal));
        glVertex3f(x1*r, y1*r, z);
    }
    glEnd();
}
}

// Called to draw scene
void RenderScene(void)
{
    // Rotation angle
    static GLfloat yRot = 0.0f;
    // Where is the light coming from
    GLTVector3 vLightDir = { -1.0f, 1.0f, 1.0f };
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glTranslatef(0.0f, 0.0f, -2.5f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);
    toonDrawTorus(0.35f, 0.15f, 50, 25, vLightDir);
    glPopMatrix();
    // Do the buffer Swap
    glutSwapBuffers();
    // Rotate 1/2 degree more each frame
    yRot += 0.5f;
}
}

// This function does any needed initialization on the rendering
// context.
void SetupRC()
{
    {
        // Load a 1D texture with toon shaded values
        // Green, greener...
        GLbyte toonTable[4][3] = { { 0, 32, 0 },
                                  { 0, 64, 0 },
                                  { 0, 128, 0 },
                                  { 0, 192, 0 } };
    }
}

```

```

// Bluish background
glClearColor(0.0f, 0.0f, .50f, 1.0f );
glEnable(GL_DEPTH_TEST);
glEnable(GL_CULL_FACE);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glTexImage1D(GL_TEXTURE_1D, 0, GL_RGB, 4, 0, GL_RGB,
             GL_UNSIGNED_BYTE, toonTable);
glEnable(GL_TEXTURE_1D);
}

///////////////////////////////
// Called by GLUT library when idle (window not being
// resized or moved)
void TimerFunction(int value)
{
    // Redraw the scene with new coordinates
    glutPostRedisplay();
    glutTimerFunc(33,TimerFunction, 1);
}

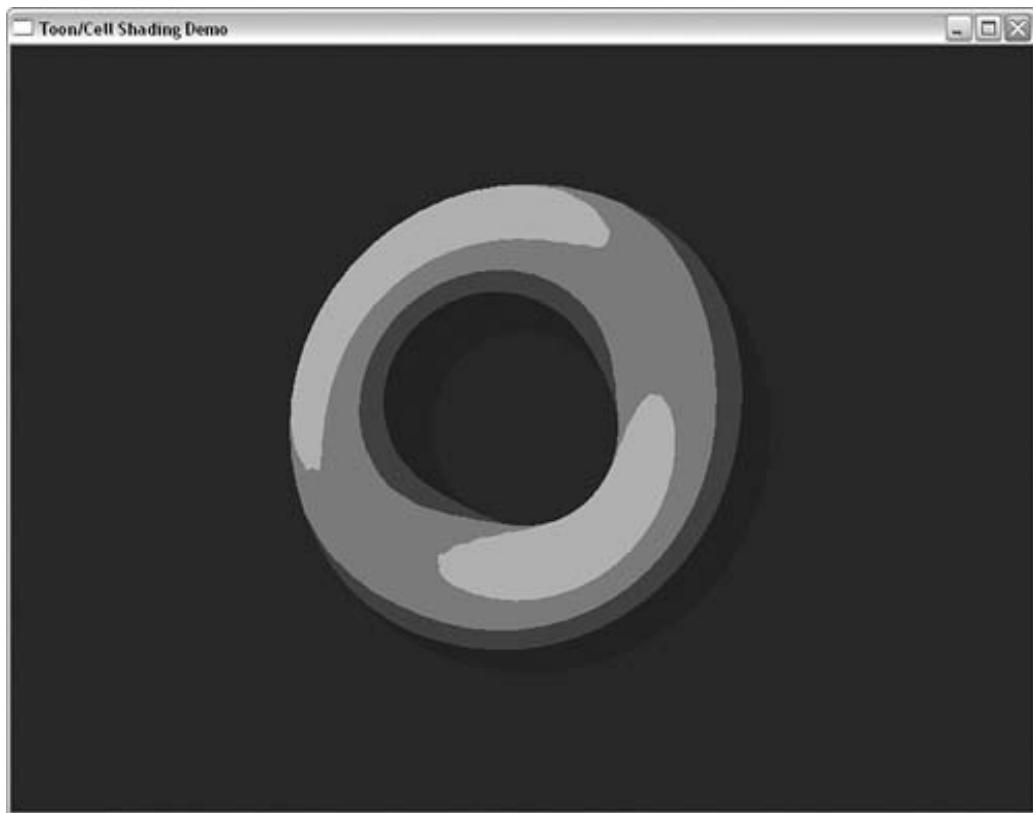
void ChangeSize(int w, int h)
{
    GLfloat fAspect;
    // Prevent a divide by zero, when window is too short
    // (you can't make a window of zero width).
    if(h == 0)
        h = 1;

    glViewport(0, 0, w, h);
    fAspect = (GLfloat)w / (GLfloat)h;
    // Reset the coordinate system before modifying
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    // Set the clipping volume
    gluPerspective(35.0f, fAspect, 1.0f, 50.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

///////////////////////////////
// Program entry point
int main(int argc, char* argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(800,600);
    glutCreateWindow("Toon/Cell Shading Demo");
    glutReshapeFunc(ChangeSize);
    glutDisplayFunc(RenderScene);
    glutTimerFunc(33, TimerFunction, 1);
    SetupRC();
    glutMainLoop();
    return 0;
}

```

Figure 8.9. A cell-shaded torus.



Mipmapping

Mipmapping is a powerful texturing technique that can improve both the rendering performance and visual quality of a scene. It does this by addressing two common problems with standard texture mapping. The first is an effect called *scintillation* (aliasing artifacts) that appears on the surface of objects rendered very small onscreen compared to the relative size of the texture applied. Scintillation can be seen as a sort of sparkling that occurs as the sampling area on a texture map moves disproportionately to its size on the screen. The negative effects of scintillation are most noticeable when the camera or the objects are in motion.

The second issue is more performance related, but is due to the same scenario that leads to scintillation. That is, a large amount of texture memory must be loaded and processed through filtering to display a small number of fragments onscreen. This causes texturing performance to suffer greatly as the size of the texture increases.

The solution to both of these problems is to simply use a smaller texture map. However, this solution then creates a new problem, that when near the same object, it must be rendered larger, and a small texture map will then be stretched to the point of creating a hopelessly blurry or blocky textured object.

The solution to both of these issues is mipmapping. Mipmapping gets its name from the Latin phrase *multum in parvo*, which means "many things in a small place." In essence, you load not a single image into the texture state, but a whole series of images from largest to smallest into a single "mipmapped" texture state. OpenGL then uses a new set of filter modes to choose the best-fitting texture or textures for the given geometry. At the cost of some extra memory (and possibly considerably more processing work), you can eliminate scintillation and the texture memory processing overhead for distant objects simultaneously, while maintaining higher resolution versions of the texture available when needed.

A mipmapped texture consists of a series of texture images, each one half the size of the previous image. This scenario is shown in [Figure 8.10](#). Mipmap levels do not have to be square, but the halving of the dimensions continues until the last image is 1x1 texel. When one of the dimensions reaches 1, further divisions occur on the other dimension only. Using a square set of mipmaps requires about one third more memory than not using mipmaps.

Figure 8.10. A series of mipmapped images.

Mipmap levels are loaded with `glTexImage`. Now the `level` parameter comes into play because it specifies which mip level the image data is for. The first level is 0, then 1, 2, and so on. If mipmaping" is not being used, only level 0 is ever loaded. By default, to use mipmaps, all mip levels must be populated. You can, however, specifically set the base and maximum levels to be used with the `GL_TEXTURE_BASE_LEVEL` and `GL_TEXTURE_MAX_LEVEL` texture parameters. For example, if you want to specify that only mip levels 0 through 4 need to be loaded, you call `glTexParameter` twice as follows:

```
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 0);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 4);
```

Although `GL_TEXTURE_BASE_LEVEL` and `GL_TEXTURE_MAX_LEVEL` control which mip levels are loaded, you can also specifically limit the range of loaded mip levels to be used by using the parameters `GL_TEXTURE_MIN_LOD` and `GL_TEXTURE_MAX_LOD` instead.

Mipmap Filtering

Mipmapping adds a new twist to the two basic texture filtering modes `GL_NEAREST` and `GL_LINEAR` by giving four permutations for mipmapped filtering modes. They are listed in [Table 8.4](#).

Table 8.4. Mipmapped Texture Filters

Constant	Description
<code>GL_NEAREST</code>	Perform nearest neighbor filtering on the base mip level
<code>GL_LINEAR</code>	Perform linear filtering on the base mip level
<code>GL_NEAREST_MIPMAP_NEAREST</code>	Select the nearest mip level and perform nearest neighbor filtering
<code>GL_NEAREST_MIPMAP_LINEAR</code>	Perform a linear interpolation between mip levels and perform nearest neighbor filtering
<code>GL_LINEAR_MIPMAP_NEAREST</code>	Select the nearest mip level and perform linear filtering
<code>GL_LINEAR_MIPMAP_LINEAR</code>	Perform a linear interpolation between mip levels and perform linear filtering; also called <i>trilinear</i> mipmapping

Just loading the mip levels with `glTexImage` does not by itself enable mipmapping. If the texture filter is set to `GL_LINEAR` or `GL_NEAREST`, only the base texture level is used, and any mip levels loaded are ignored. You must specify one of the mipmapped filters listed for the loaded mip levels to be used. The constants have the form `GL_FILTER_MIPMAP_SELECTOR`, where `FILTER` specifies

the texture filter to be used on the mip level selected. The *SELECTOR* specifies how the mip level is selected; for example, `GL_NEAREST` selects the nearest matching mip level. Using `GL_LINEAR` for the selector creates a linear interpolation between the two nearest mip levels, which is again filtered by the chosen texture filter. Selecting one of the mipmapped filtering modes without loading the mip levels has the effect of disabling texture mapping.

Which filter you select varies depending on the application and the performance requirements at hand. `GL_NEAREST_MIPMAP_NEAREST`, for example, gives very good performance and low aliasing (scintillation) artifacts, but nearest neighbor filtering is often not visually pleasing. `GL_LINEAR_MIPMAP_NEAREST` is often used to speed up games because a higher quality linear filter is used, but a fast selection (nearest) is made between the different sized mip levels available.

Using nearest as the mipmap selector (as in both examples in the preceding paragraph), however, can also leave an undesirable visual artifact. For oblique views, you can often see the transition from one mip level to another across a surface. It can be seen as a distortion line or a sharp transition from one level of detail to another. The `GL_LINEAR_MIPMAP_LINEAR` and `GL_NEAREST_MIPMAP_LINEAR` filters perform an additional interpolation between mip levels to eliminate this transition zone, but at the extra cost of substantially more processing overhead. The `GL_LINEAR_MIPMAP_LINEAR` filter is often referred to as *trilinear* mipmaping and until recently was the gold standard (highest fidelity) of texture filtering. More recently, *anisotropic* texture filtering (covered in the next chapter) has become widely available on OpenGL hardware but even further increases the cost (performance wise) of texture mapping.

Generating Mip Levels

As mentioned previously, mipmaping requires approximately one third more texture memory than just loading the base texture image. It also requires that all the smaller versions of the base texture image be available for loading. Sometimes this can be inconvenient because the lower resolution images may not necessarily be available either to the programmer or the end user of your software. The GLU library does include a function named `gluScaleImage` (see the reference section) that you could use to repeatedly scale and load an image until all the needed mip levels are loaded. More frequently, however, an even more convenient function is available; it automatically creates the scaled images for you and loads them appropriately with `glTexImage`. This function, `gluBuildMipmaps`, comes in three flavors and supports one-, two-, and three-dimensional texture maps:

```
int gluBuild1DMipmaps(GLenum target, GLint internalFormat,
                      GLint width,
                      GLenum format, GLenum type, const void *data);
int gluBuild2DMipmaps(GLenum target, GLint internalFormat,
                      GLint width, GLint height,
                      GLenum format, GLenum type, const void *data);
int gluBuild3DMipmaps(GLenum target, GLint internalFormat,
                      GLint width, GLint height, GLint depth,
                      GLenum format, GLenum type, const void *data);
```

The use of these functions closely parallels the use of `glTexImage`, but they do not have a *level* parameter for specifying the mip level, nor do they provide any support for a texture border. You should also be aware that using these functions may not produce mip level images with the same quality you can obtain with other tools such as Photoshop. The GLU library uses a *box filter* to reduce images, which can lead to an undesirable loss of fine detail as the image shrinks.

With newer versions of the GLU library, you can also obtain a finer-grained control over which mip levels are loaded with these functions:

```
int gluBuild1DMipmapLevels(GLenum target, GLint internalFormat,
                           GLint width,
```

```

GLenum format, GLenum type, GLint level,
GLint base, GLint max, const void *data);

int gluBuild2DMipmapLevels(GLenum target, GLint internalFormat,
                           GLint width, GLint height,
                           GLenum format, GLenum type, GLint level,
                           GLint base, GLint max, const void *data);

int gluBuild3DMipmapLevels(GLenum target, GLint internalFormat,
                           GLint width, GLint height, GLint depth,
                           GLenum format, GLenum type, GLint level,
                           GLint base, GLint max, const void *data);

```

With these functions, `level` is the mip level specified by the `data` parameter. This texture data is used to build mip levels `base` through `max`.

Hardware Generation of Mipmaps

If you know beforehand that you want all mip levels loaded, you can also use OpenGL hardware acceleration to quickly generate all the necessary mip levels. You do so by setting the texture parameter `GL_GENERATE_MIPMAP` to `GL_TRUE`:

```
glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, GL_TRUE);
```

When this parameter is set, all calls to `glTexImage` or `glTexSubImage` that update the base texture map (mip level 0) automatically update all the lower mip levels. By making use of the graphics hardware, this feature is substantially faster than using `gluBuildMipmaps`. However, you should be aware that this feature was originally an extension and was promoted to the OpenGL core API only as of version 1.4.

LOD BIAS

When mipmapping is enabled, OpenGL uses a formula to determine which mip level should be selected based on the size of the mipmap levels and the onscreen area the geometry occupies. OpenGL does its best to make a close match between the mipmap level chosen and the texture's representation onscreen. You can tell OpenGL to move its selection criteria back (lean toward larger mip levels) or forward (lean toward smaller mip levels). This can have the effect of increasing performance (using smaller mip levels) or increasing the sharpness of texture-mapped objects (using larger mip levels). This bias one way or the other is selected with the texture environment parameter `GL_TEXTURE_LOD_BIAS`, as shown here:

```
glTexEnvf(GL_TEXTURE_FILTER_CONTROL, GL_TEXTURE_LOD_BIAS, -1.5);
```

In this example, the texture level of detail is shifted slightly toward using higher levels of detail (smaller level parameters), resulting in sharper looking textures, at the expense of slightly more texture processing overhead.

Texture Objects

So far, you have seen how to load a texture and set texture parameters to affect how texture maps are applied to geometry. The texture image and parameters set with `glTexParameter` comprise the *texture state*. Loading and maintaining the texture state occupies a considerable portion of many texture-heavy OpenGL applications (such as games in particular).

Especially time consuming are function calls such as `glTexImage`, `glTexSubImage`, and `gluBuildMipmaps`. These functions move a large amount of memory around and possibly need to reformat the data to match some internal representation. Switching between textures or reloading a different texture image would ordinarily be a costly operation.

Texture objects allow you to load up more than one texture state at a time, including texture images, and switch between them very quickly. The texture state is maintained by the currently bound texture object, which is identified by an unsigned integer. You allocate a number of texture objects with the following function:

```
void glGenTextures(GLsizei n, GLuint *textures);
```

With this function, you specify the number of texture objects and a pointer to an array of unsigned integers that will be populated with the texture object identifiers. You can think of them as handles to different available texture states. To "bind" to one of these states, you call the following function:

```
void glBindTexture(GLenum target, GLuint texture);
```

The *target* parameter needs to specify either `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, or `GL_TEXTURE_3D`, and *texture* is the specific texture object to bind to. Hereafter, all texture loads and texture parameter settings affect only the currently bound texture object. To delete texture objects, you call the following function:

```
void glDeleteTextures(GLsizei n, GLuint *textures);
```

The arguments here have the same meaning as for `glGenTextures`. You do not need to generate and delete all your texture objects at the same time. Multiple calls to `glGenTextures` have very little overhead. Calling `glDeleteTextures` multiple times may incur some delay, but only because you are deallocating possibly large amounts of texture memory.

You can test texture object names (or handles) to see whether they are valid by using the following function:

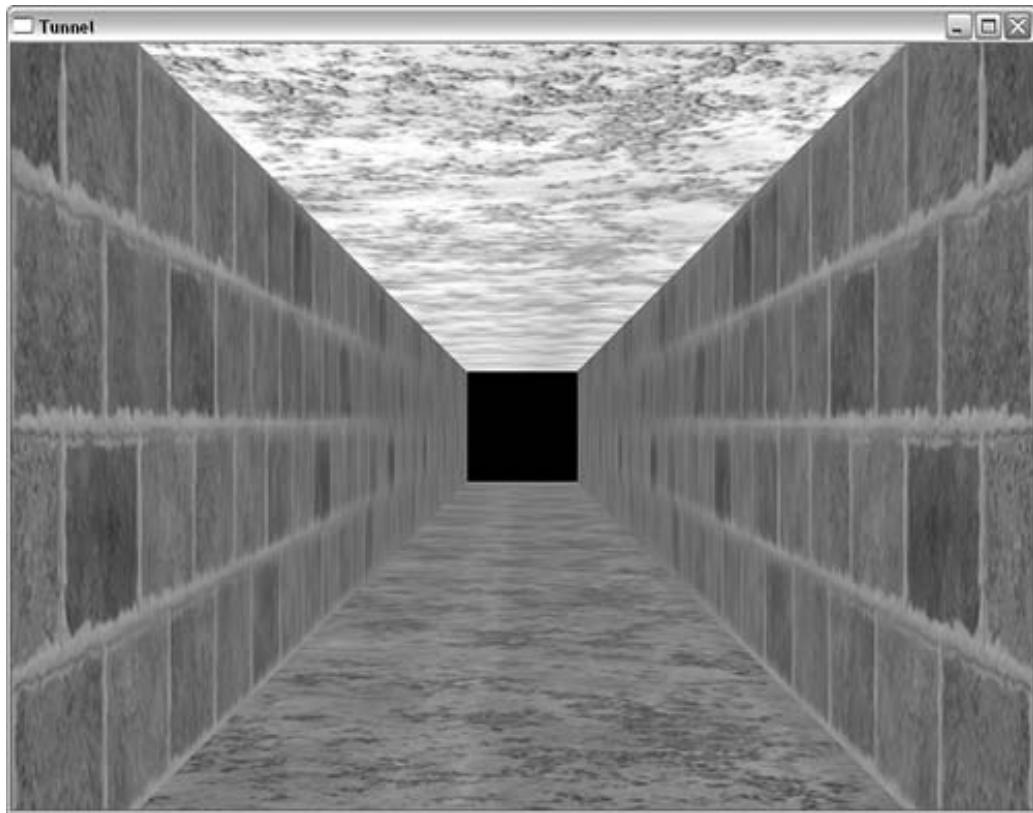
```
GLboolean glIsTexture(GLuint texture);
```

This function returns `GL_TRUE` if the integer is a previously allocated texture object name or `GL_FALSE` if not.

Managing Multiple Textures

Generally, texture objects are used to load up several textures at program initialization and switch between them quickly during rendering. These texture objects are then deleted when the program shuts down. The TUNNEL sample program loads three textures at startup and then switches between them to render a tunnel. The tunnel has a brick wall pattern with different materials on the floor and ceiling. The output from TUNNEL is shown in [Figure 8.11](#).

Figure 8.11. A tunnel rendered with three different textures.



The TUNNEL sample program also shows off mipmapping and the different mipmapped texture filtering modes. Pressing the up- and down-arrow keys moves the point of view back and forth in the tunnel, and the context menu (right-click menu) allows you to switch among six different filtering modes to see how they affect the rendered image. The complete source code is provided in [Listing 8.3](#).

Listing 8.3. Source Code for the TUNNEL Sample Program

```

// Tunnel.c
// Demonstrates mipmapping and using texture objects
// OpenGL SuperBible
// Richard S. Wright Jr.
#include "../../Common/OpenGLSB.h" // System and OpenGL Stuff
#include "../../Common/GLTools.h" // GLTools
// Rotation amounts
static GLfloat zPos = -60.0f;
// Texture objects
#define TEXTURE_BRICK 0
#define TEXTURE_FLOOR 1
#define TEXTURE_CEILING 2
#define TEXTURE_COUNT 3
GLuint textures[TEXTURE_COUNT];
const char *szTextureFiles[TEXTURE_COUNT] =
    { "brick.tga", "floor.tga", "ceiling.tga" };
///////////////////////////////
// Change texture filter for each texture object
void ProcessMenu(int value)
{
    GLint iLoop;
    for(iLoop = 0; iLoop < TEXTURE_COUNT; iLoop++)
    {
        glBindTexture(GL_TEXTURE_2D, textures[iLoop]);
        switch(value)
        {

```

```

    case 0:
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                        GL_NEAREST);
        break;
    case 1:
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                        GL_LINEAR);
        break;
    case 2:
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                        GL_NEAREST_MIPMAP_NEAREST);
        break;
    case 3:
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                        GL_NEAREST_MIPMAP_LINEAR);
        break;
    case 4:
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                        GL_LINEAR_MIPMAP_NEAREST);
        break;
    case 5:
    default:
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                        GL_LINEAR_MIPMAP_LINEAR);
        break;
    }
}

// Trigger Redraw
glutPostRedisplay();
}

////////////////////////////////////////////////////////////////
// This function does any needed initialization on the rendering
// context. Here it sets up and initializes the texture objects.
void SetupRC()
{
    GLubyte *pBytes;
    GLint iWidth, iHeight, iComponents;
    GLenum eFormat;
    GLint iLoop;
    // Black background
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    // Textures applied as decals, no lighting or coloring effects
    glEnable(GL_TEXTURE_2D);
    glTexEnv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
    // Load textures
    glGenTextures(TEXTURE_COUNT, textures);
    for(iLoop = 0; iLoop < TEXTURE_COUNT; iLoop++)
    {
        // Bind to next texture object
        glBindTexture(GL_TEXTURE_2D, textures[iLoop]);
        // Load texture, set filter and wrap modes
        pBytes = gltLoadTGA(szTextureFiles[iLoop], &iWidth, &iHeight,
                            &iComponents, &eFormat);
        gluBuild2DMipmaps(GL_TEXTURE_2D, iComponents, iWidth, iHeight, eFormat,
                          GL_UNSIGNED_BYTE, pBytes);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                        GL_LINEAR_MIPMAP_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        // Don't need original texture data any more
        free(pBytes);
    }
}

```

```

        }

///////////
// Shutdown the rendering context. Just deletes the
// texture objects
void ShutdownRC(void)
{
    glDeleteTextures(TEXTURE_COUNT, textures);
}
///////////
// Respond to arrow keys, move the viewpoint back
// and forth
void SpecialKeys(int key, int x, int y)
{
    if(key == GLUT_KEY_UP)
        zPos += 1.0f;
    if(key == GLUT_KEY_DOWN)
        zPos -= 1.0f;
    // Refresh the Window
    glutPostRedisplay();
}
///////////
// Change viewing volume and viewport. Called when window is resized
void ChangeSize(int w, int h)
{
    GLfloat fAspect;
    // Prevent a divide by zero
    if(h == 0)
        h = 1;
    // Set Viewport to window dimensions
    glViewport(0, 0, w, h);
    fAspect = (GLfloat)w/(GLfloat)h;

    // Reset coordinate system
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    // Produce the perspective projection
    gluPerspective(90.0f,fAspect,1,120);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
///////////
// Called to draw scene
void RenderScene(void)
{
    GLfloat z;
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT);
    // Save the matrix state and do the rotations
    glPushMatrix();
        // Move object back and do in place rotation
        glTranslatef(0.0f, 0.0f, zPos);
        // Floor
        for(z = 60.0f; z >= 0.0f; z -= 10)
        {
            glBindTexture(GL_TEXTURE_2D, textures[TEXTURE_FLOOR]);
            glBegin(GL_QUADS);
                glTexCoord2f(0.0f, 0.0f);
                glVertex3f(-10.0f, -10.0f, z);
                glTexCoord2f(1.0f, 0.0f);
                glVertex3f(10.0f, -10.0f, z);

```

```

glTexCoord2f(1.0f, 1.0f);
glVertex3f(10.0f, -10.0f, z - 10.0f);

glTexCoord2f(0.0f, 1.0f);
glVertex3f(-10.0f, -10.0f, z - 10.0f);
glEnd();
// Ceiling
glBindTexture(GL_TEXTURE_2D, textures[TEXTURE_CEILING]);
glBegin(GL_QUADS);
    glTexCoord2f(0.0f, 1.0f);
    glVertex3f(-10.0f, 10.0f, z - 10.0f);
    glTexCoord2f(1.0f, 1.0f);
    glVertex3f(10.0f, 10.0f, z - 10.0f);
    glTexCoord2f(1.0f, 0.0f);
    glVertex3f(10.0f, 10.0f, z);
    glTexCoord2f(0.0f, 0.0f);
    glVertex3f(-10.0f, 10.0f, z);
glEnd();
// Left Wall
glBindTexture(GL_TEXTURE_2D, textures[TEXTURE_BRICK]);
glBegin(GL_QUADS);
    glTexCoord2f(0.0f, 0.0f);
    glVertex3f(-10.0f, -10.0f, z);
    glTexCoord2f(1.0f, 0.0f);
    glVertex3f(-10.0f, -10.0f, z - 10.0f);
    glTexCoord2f(1.0f, 1.0f);
    glVertex3f(-10.0f, 10.0f, z - 10.0f);
    glTexCoord2f(0.0f, 1.0f);
    glVertex3f(-10.0f, 10.0f, z);
glEnd();
// Right Wall
glBegin(GL_QUADS);
    glTexCoord2f(0.0f, 1.0f);
    glVertex3f(10.0f, 10.0f, z);

    glTexCoord2f(1.0f, 1.0f);
    glVertex3f(10.0f, 10.0f, z - 10.0f);
    glTexCoord2f(1.0f, 0.0f);
    glVertex3f(10.0f, -10.0f, z - 10.0f);
    glTexCoord2f(0.0f, 0.0f);
    glVertex3f(10.0f, -10.0f, z);
glEnd();
}
// Restore the matrix state
glPopMatrix();
// Buffer swap
glutSwapBuffers();
}

///////////////
// Program entry point
int main(int argc, char *argv[])
{
    // Standard initialization stuff
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(800, 600);
    glutCreateWindow("Tunnel");
    glutReshapeFunc(ChangeSize);
    glutSpecialFunc(SpecialKeys);
    glutDisplayFunc(RenderScene);
    // Add menu entries to change filter
    glutCreateMenu(ProcessMenu);
}

```

```

glutAddMenuEntry( "GL_NEAREST", 0 );
glutAddMenuEntry( "GL_LINEAR", 1 );
glutAddMenuEntry( "GL_NEAREST_MIPMAP_NEAREST", 2 );
glutAddMenuEntry( "GL_NEAREST_MIPMAP_LINEAR", 3 );
glutAddMenuEntry( "GL_LINEAR_MIPMAP_NEAREST", 4 );
glutAddMenuEntry( "GL_LINEAR_MIPMAP_LINEAR", 5 );
glutAttachMenu(GLUT_RIGHT_BUTTON);
// Startup, loop, shutdown
SetupRC();
glutMainLoop();
ShutdownRC();
return 0;
}

```

In this example, you first create identifiers for the three texture objects. The array `textures` will contain three integers, which will be addressed by using the macros `TEXTURE_BRICK`, `TEXTURE_FLOOR`, and `TEXTURE_CEILING`. For added flexibility, you also create a macro that defines the maximum number of textures that will be loaded and an array of character strings containing the names of the texture map files:

```

// Texture objects
#define TEXTURE_BRICK    0
#define TEXTURE_FLOOR    1
#define TEXTURE_CEILING  2
#define TEXTURE_COUNT    3
GLuint  textures[TEXTURE_COUNT];
const char *szTextureFiles[TEXTURE_COUNT] =
    { "brick.tga", "floor.tga", "ceiling.tga" };

```

The texture objects are allocated in the `SetupRC` function:

```
glGenTextures(TEXTURE_COUNT, textures);
```

Then a simple loop binds to each texture object in turn and loads its texture state with the texture image and texturing parameters:

```

for(iLoop = 0; iLoop < TEXTURE_COUNT; iLoop++)
{
    // Bind to next texture object
    glBindTexture(GL_TEXTURE_2D, textures[iLoop]);
    // Load texture, set filter and wrap modes
    pBytes = gltLoadTGA(szTextureFiles[iLoop],&iWidth, &iHeight,
                        &iComponents, &eFormat);
    gluBuild2DMipmaps(GL_TEXTURE_2D, iComponents, iWidth, iHeight, eFormat,
                      GL_UNSIGNED_BYTE, pBytes);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_LINEAR_MIPMAP_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    // Don't need original texture data any more
    free(pBytes);
}

```

With each of the three texture objects initialized, you can easily switch between them during rendering to change textures:

```
glBindTexture(GL_TEXTURE_2D, textures[TEXTURE_FLOOR]);
glBegin(GL_QUADS);
```

```

glTexCoord2f(0.0f, 0.0f);
glVertex3f(-10.0f, -10.0f, z);
glTexCoord2f(1.0f, 0.0f);
glVertex3f(10.0f, -10.0f, z);
...
...

```

Finally, when the program is terminated, you only need to delete the texture objects for the final cleanup:

```

///////////////////////////////
// Shutdown the rendering context. Just deletes the
// texture objects
void ShutdownRC(void)
{
    glDeleteTextures(TEXTURE_COUNT, textures);
}

```

Also, note that when the mipmapped texture filter is set in the TUNNEL sample program, it is selected only for the minification filter:

```

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);

```

This is typically the case because after OpenGL selects the largest available mip level, no larger levels are available to select between. Essentially, this is to say that once a certain threshold is passed, the largest available texture image is used and there are no additional mipmap levels to choose from.

Resident Textures

Most OpenGL implementations support a limited amount of high-performance texture memory. Textures located in this memory are accessed very quickly, and performance is high. Initially, any loaded texture is stored in this memory; however, only a limited amount of memory is typically available, and at some point textures may need to be stored in slower memory. As is often the case, this slower memory may even be located outside the OpenGL hardware (such as in a PC's system memory as opposed to being stored on the graphics card or in AGP memory).

To optimize rendering performance, OpenGL automatically moves frequently accessed textures into this high-performance memory. Textures in this high-performance memory are called *resident* textures. To determine whether a bound texture is resident, you can call `glGetTexParameter` and find the value associated with `GL_TEXTURE_RESIDENT`. Testing a group of textures to see whether they are resident may be more useful, and you can perform this test using the following function:

```

GLboolean glAreTexturesResident(GLsizei n, const GLuint *textures,
                                GLboolean *residences);

```

This function takes the number of texture objects to check, an array of the texture object names, and finally an array of Boolean flags set to `GL_TRUE` or `GL_FALSE` to indicate the status of each texture object. If all the textures are resident, the array is left unchanged, and the function returns `GL_TRUE`. This feature is meant to save the time of having to check through an entire array to see whether all the textures are resident.

Texture Priorities

By default, most OpenGL implementations use a Most Frequently Used (MFU) algorithm to decide which textures can stay resident. However, if several smaller textures are used only slightly more frequently than, say, a much larger texture, texturing performance can suffer considerably. You

can provide hints to whatever mechanism an implementation uses to decide texture residency by setting each texture's priority with this function:

```
void glPrioritizeTextures(GLsizei n, const GLuint *textures,
                           const GLclampf *priorities);
```

This function takes an array of texture object names and a corresponding array of texture object priorities that are clamped between 0 and 1.0. A low priority tells the implementation that this texture object should be left out of resident memory whenever space becomes tight. A higher priority (such as 1.0) tells the implementation that you want that texture object to remain resident if possible, even if the texture seems to be used infrequently.

Summary

In this chapter, we extended the simple image loading and display methods from the preceding chapter to applying images as texture maps to real three-dimensional geometry. You learned how to load a texture map and use texture coordinates to map the image to the vertices of geometry. You also learned the different ways in which texture images can be filtered and blended with the geometry color values and how to use mipmaps to improve both performance and visual fidelity. Finally, we discussed how to manage multiple textures and switch between them quickly and easily, and how to tell OpenGL which textures should have priority if any high-performance (or local) texture memory is available.

Reference

glAreTexturesResident

Purpose: Determine whether a set of texture objects is resident.

Include File: `<gl.h>`

Syntax:

```
GLboolean glAreTexturesResident(GLsizei n, const
→ GLuint *textures,
→ GLboolean *residences);
```

Description: Many OpenGL implementations support a high-performance set of textures that are said to be resident. Resident textures are kept in local or fast memory accessible directly by the OpenGL hardware. Texturing with resident textures is substantially faster than using nonresident textures. This function allows you to test an array of texture object handles for residency. If all the texture object's queries are resident, the function returns `GL_TRUE`.

Parameters:

n `GLsizei`: The size of the array of texture objects.

textures `GLuint*`: The array containing texture object identifiers to be queried.

residences `GLboolean*`: The array to be populated with corresponding flags (`GL_TRUE` or `GL_FALSE`) for each texture object. If all the texture objects specified in *textures* are resident, the array is not modified.

Returns: None

See Also: `glGenTextures`, `glBindTexture`, `glDeleteTextures`, `glPrioritizeTextures`

glBindTexture**Purpose:** Binds the current texture state to the named target.**Include File:** `<gl.h>`**Syntax:**

```
void glBindTexture(GLenum target, GLuint texture);
```

Description: This function enables you to create or switch to a named texture state. On first use, this function creates a new texture state identified by the texture name, which is an unsigned integer. Subsequent calls with the same texture identifier select that texture state and make it current.**Parameters:**

target `GLenum`: The texture target to bind to. It must be `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, or `GL_TEXTURE_3D`.

texture `GLuint`: The name or handle of the texture object.

Returns: None.

See Also: `glGenTextures`, `glDeleteTextures`, `glAreTexturesResident`, `glPrioritizeTextures`, `glIsTexture`

glCopyTexImage**Purpose:** Copies pixels from the color buffer into a texture.**Include File:** `<gl.h>`**Variations:**

```
void glCopyTexImage1D(GLenum target, GLint level,
    ➔ GLenum internalFormat,
                GLint x, GLint y,
    ➔ GLsizei width, GLint border);
void glCopyTexImage2D(GLenum target, GLint level,
    ➔ GLenum internalFormat,
                GLint x, GLint y, GLsizei width,
    ➔ GLsizei height, GLint border);
```

Description: These functions define a one- or two-dimensional texture image using data read directly from the color buffer. If the source of a texture map is to be the result of a rendering operation, be sure to call `glFinish` to ensure that OpenGL has completed the rendering operation before calling this function. Data is read from the color buffer specified by the `glReadBuffer` function.**Parameters:**

target `GLenum`: The texture target. It must be `GL_TEXTURE_1D` for `glCopyTexImage1D` and `GL_TEXTURE_2D` for `glCopyTexImage2D`.

<i>level</i>	<code>GLint</code> : The mipmap level to be loaded.
<i>internal Format</i>	<code>GLenum</code> : The internal format and resolution of the texture data. This must be one of the constants for texture formats acceptable by <code>glTexImage</code> , with the exception that you cannot use the values 1, 2, 3, or 4 for the number of color components.
<i>x, y</i>	<code>GLint</code> : The location in the color buffer to begin reading color data.
<i>width, height</i>	<code>GLsizei</code> : The width and height (for <code>glCopyTexImage2D</code> only) of the color data rectangle to be read from the color buffer.
<i>border</i>	<code>GLint</code> : The width of the texture border. Only 1 or 0 is allowed.
Returns:	None.
See Also:	<code>glTexImage</code> , <code>glCopyTexSubImage</code>

glCopyTexSubImage

Purpose: Replaces part of a texture map using data from the color buffer.

Include File: `<gl.h>`

Variations:

```
void glCopyTexSubImage1D(GLenum target, GLint level,
                         GLint xoffset,
                         GLint x, GLint y,
                         GLsizei width);
void glCopyTexSubImage2D(GLenum target, GLint level,
                         GLint xoffset,
                         ➔ GLint yoffset,
                         GLint x, GLint y,
                         GLsizei width,
                         ➔ GLsizei height);
void glCopyTexSubImage3D(GLenum target, GLint level,
                         GLint xoffset,
                         ➔ GLint yoffset, GLint zoffset,
                         GLint x, GLint y,
                         GLsizei width,
                         ➔ GLsizei height);
```

Description: This function replaces part of an existing texture map with data read directly from the color buffer. If the source of a texture map is to be the result of a rendering operation, be sure to call `glFinish` to ensure that OpenGL has completed the rendering operation before calling this function. Data is read from the color buffer specified by the `glReadBuffer` function.

Parameters:

<i>target</i>	<code>GLenum</code> : The texture target. It must be <code>GL_TEXTURE_1D</code> for <code>glCopyTexSubImage1D</code> , <code>GL_TEXTURE_2D</code> for <code>glCopyTexSubImage2D</code> , and <code>GL_TEXTURE_3D</code> for <code>glCopyTexSubImage3D</code> .
<i>level</i>	<code>GLint</code> : The mipmap level being updated.

xoffset **GLint**: The x offset into the texture map to begin replacing data.

yoffset **GLint**: The y offset into the two- or three-dimensional texture map to begin replacing data.

zoffset **GLint**: The z offset into the three-dimensional texture map to begin replacing data.

x, y **GLint**: The x,y location in the color buffer to begin reading the texture data.

width, height **GLsizei**: The width and height (for two and three dimensions only) of the data to be read from the color buffer.

Returns: None.

See Also: [glTexImage](#), [glCopyTexImage](#), [glTexSubImage](#)

glDeleteTextures

Purpose: Deletes a set of texture objects.

Include File: `<gl.h>`

Syntax:

```
void glDeleteTextures(GLsizei n, const GLuint
→ *textures);
```

Description: This function deletes a set of texture objects. When a texture object is deleted, it is available to be redefined by a subsequent call to [glBindTexture](#). Any memory used by the existing texture objects is released and available for other textures. Any invalid texture object names in the array are ignored.

Parameters:

n **GLsizei**: The number of texture objects to delete.

textures **GLuint***: An array containing the list of texture objects to be deleted.

Returns: None.

See Also: [glGenTextures](#), [glBindTexture](#)

glGenTextures

Purpose: Generates a list of texture object names.

Include File: `<gl.h>`

Syntax:

```
void glGenTextures(GLsizei n, GLuint *textures);
```

Description: This function fills an array with the requested number of texture objects. Texture object names are unsigned integers, but there is no guarantee that the returned array will contain a continuous sequence of integer names. Texture object names returned by this function are always be unique, unless they have been previously deleted with `glDeleteTextures`.

Parameters:

`n` `GLsizei`: The number of texture object names to generate.

`textures` `GLuint*`: An array containing the list of newly generated texture object names.

Returns: None.

See Also: `glDeleteTextures`, `glBindTexture`, `glIsTexture`

glGetTexLevelParameter

Purpose: Returns texture parameters for a specific mipmap level.

Include File: `<gl.h>`

Variations:

```
void glGetTexLevelParameterfv(GLenum target, GLint
  ↪ level, GLenum pname,
  ↪ GLfloat *params);
void glGetTexLevelParameteriv(GLenum target, GLint
  ↪ level, GLenum pname,
  ↪ GLint *params);
```

Description: This function enables you to query for the value of a number of different parameters that may be valid for a specific texture mipmap level. [Table 8.5](#) lists the specific texture level parameters that may be queried. Returned results may be contained in one or more floating-point or integer values.

Parameters:

`target` `GLenum`: The texture target being queried. It must be `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D`, `GL_PROXY_TEXTURE_1D`, `GL_PROXY_TEXTURE_2D`, `GL_PROXY_TEXTURE_3D`, `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, or `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`.

`level` `GLint`: The mipmap level being queried.

`pname` `GLenum`: A constant value from [Table 8.5](#) that specifies the texture parameter being queried.

`params` `GLfloat*` or `GLint*`: The returned parameter value or values are stored here.

Returns: None.

See Also: `glGetTexParameter`, `glTexParameter`

Table 8.5. Texture-Level Parameter Query Constants

Constant	Description
<code>GL_TEXTURE_WIDTH</code>	The width of the texture image, including the texture border if defined
<code>GL_TEXTURE_HEIGHT</code>	The height of the texture image, including the texture border if defined
<code>GL_TEXTURE_DEPTH</code>	The depth of the texture image, including the texture border if defined
<code>GL_TEXTURE_INTERNAL_FORMAT</code>	The internal format of the texture image
<code>GL_TEXTURE_BORDER</code>	The width of the texture border
<code>GL_TEXTURE_RED_SIZE</code>	The resolution in bits of the red component of a texel
<code>GL_TEXTURE_GREEN_SIZE</code>	The resolution in bits of the green component of a texel
<code>GL_TEXTURE_BLUE_SIZE</code>	The resolution in bits of the blue component of a texel
<code>GL_TEXTURE_ALPHA_SIZE</code>	The resolution in bits of the alpha component of a texel
<code>GL_TEXTURE_LUMINANCE_SIZE</code>	The resolution in bits of the luminance of a texel
<code>GL_TEXTURE_INTENSITY_SIZE</code>	The resolution in bits of the intensity of a texel
<code>GL_TEXTURE_COMPONENTS</code>	The number of color components in the texture image
<code>GL_TEXTURE_COMPRESSED_IMAGE_SIZE</code>	The size in bytes of the compressed texture image (must have a compressed internal format)

glGetTexParameter

Purpose: Queries for texture parameter values.

Include File: `<gl.h>`

Variations:

```
void glGetTexParameterfv(GLenum target, GLenum
  ↪ pname, GLfloat *params);
void glGetTexParameteriv(GLenum target, GLenum
  ↪ pname, GLint *params);
```

Description: This function enables you to query for the value of a number of different texture parameters. This function is frequently used with one of the proxy texture targets to see whether a texture can be loaded. [Table 8.6](#) lists the specific texture parameters that may be queried. Returned results may be contained in one or more floating-point or integer values.

Parameters:

`target` `GLenum`: The texture target being queried. It must be `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D`, `GL_PROXY_TEXTURE_1D`, `GL_PROXY_TEXTURE_2D`, `GL_PROXY_TEXTURE_3D`, or `GL_TEXTURE_CUBE_MAP`.

pname **GLenum**: The parameter value being queried. It may be any constant from [Table 8.6](#).

params **GLfloat*** or **GLint***: Address of variable or variables to receive the parameter value or values.

Returns: None.

See Also: [glGetTexLevelParameter](#), [glTexParameter](#)

Table 8.6. Texture Parameter Query Constants

Constant	Description
GL_TEXTURE_MAG_FILTER	Returns the texture magnification filter value
GL_TEXTURE_MIN_FILTER	Returns the texture minification filter
GL_TEXTURE_MIN_LOD	Returns the minimum level of detail value
GL_TEXTURE_MAX_LOD	Returns the maximum level of detail value
GL_TEXTURE_BASE_LEVEL	Returns the base texture mipmap level
GL_TEXTURE_MAX_LEVEL	Returns the maximum mipmap array level
GL_TEXTURE_LOD_BIAS	Texture Level of Detail bias
GL_TEXTURE_WRAP_S	Returns the wrap mode in the s coordinate direction
GL_TEXTURE_WRAP_T	Returns the wrap mode in the t coordinate direction
GL_TEXTURE_WRAP_R	Returns the wrap mode in the r coordinate direction
GL_TEXTURE_BORDER_COLOR	Returns the texture border color
GL_TEXTURE_PRIORITY	Returns the current texture priority settings
GL_TEXTURE_RESIDENT	Returns GL_TRUE if the texture is resident, GL_FALSE otherwise
GL_DEPTH_TEXTURE_MODE	Returns the depth texture mode
GL_TEXTURE_COMPARE_MODE	Returns the texture comparison mode
GL_TEXTURE_COMPARE_FUNC	Returns the texture comparison function
GL_TEXTURE_GENERATE_MIPMAP	Returns GL_TRUE if automatic mipmap generation is enabled

glGetTexImage

Purpose: Returns a texture image.

Include File: `<gl.h>`

Syntax:

```
void glGetTexImage(GLenum target, GLint level,
    ➔ GLenum format,
    ➔ GLenum type, void *pixels);
```

Description: This function enables you to fill a data buffer with the data that comprises the current texture. This function performs the reverse operation of `glTexImage`, which loads a texture with a supplied data buffer.

Parameters:

`target` GLenum: The texture target to be copied. It must be `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D`, `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, or `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`.

`level` GLint: The mipmap level to be read.

`format` GLenum: The desired pixel format of the returned data. It may be `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_RGB`, `GL_RGBA`, `GL_LUMINANCE`, `GL_BGR`, `GL_BGRA`, or `GL_LUMINANCE_ALPHA`.

`type` GLenum: The pixel type for the returned data. It may be any pixel type listed in [Table 8.3](#).

`pixels` void*: Pointer to a memory buffer to accept the texture image.

Returns: None.

See Also: `glTexImage`

glIsTexture

Purpose: Determines whether a texture object name is valid.

Include File: `<gl.h>`

Syntax:

```
GLboolean glIsTexture(GLuint texture);
```

Description: This function enables you to determine whether a given integer value is a valid texture object name. A valid texture object is an integer name that is in use. This means that `glBindTexture` has been used to bind to this texture object name at least once, and it has not been subsequently deleted with `glDeleteTextures`.

Parameters:

`texture` GLuint: The texture object in question.

Returns: `GL_TRUE` if the texture object is a valid (in-use) texture object name; `GL_FALSE` otherwise.

See Also: `glGenTextures`, `glDeleteTextures`, `glBindTexture`, `glAreTexturesResident`, `glPrioritizeTextures`

glPrioritizeTextures**Purpose:** Sets priority of a texture object.**Include File:** `<gl.h>`**Syntax:**

```
void glPrioritizeTextures(GLsizei n, GLuint *textures,
→ const GLclampf *priorities);
```

Description: This function assigns *n* priorities to the texture objects contained in the *textures* array. The array of priorities is specified in *priorities*. A texture priority is a clamped value between 0.0 and 1.0 that provides the implementation with a hint that the given texture should remain resident. Resident textures are textures that are stored in high performance or local memory. A texture priority of 1.0 signifies a strong hint to keep the texture resident, whereas 0.0 signifies that the texture may be swapped out if necessary.

Parameters:

n `GLsizei`: The number of texture object names contained in the *textures* array.

textures `GLuint*`: An array of texture object names. Each array element corresponds to a priority to be set in the *priorities* array.

priorities `const GLclampf*`: An array of clamped floating-point values that specify a texture object's priority. The array element corresponds to a particular texture object name in the same location in the *textures* array.

Returns: None.

See Also: `glAreTexturesResident`, `glGenTextures`, `glBindTexture`, `glDeleteTexture`, `glIsTexture`

glTexCoord**Purpose:** Specifies the current texture image coordinate for textured polygon rendering.**Include File:** `<gl.h>`**Variations:**

```
void glTexCoord1f(GLfloat s);
void glTexCoord1fv(GLfloat *v);
void glTexCoord1d(GLdouble s);
void glTexCoord1dv(GLdouble *v);
void glTexCoord1i(GLint s);
void glTexCoord1liv(GLint *v);
void glTexCoord1s(GLshort s);
void glTexCoord1sv(GLfloat *v);
void glTexCoord2f(GLfloat s, GLfloat t);
void glTexCoord2fv(GLfloat *v);
void glTexCoord2d(GLdouble s, GLdouble t);
void glTexCoord2dv(GLdouble *v);
void glTexCoord2i(GLint s, GLint t);
```

```

void glTexCoord2iv(GLint *v);
void glTexCoord2s(GLshort s, GLshort t);
void glTexCoord2sv(GLfloat *v);
void glTexCoord3f(GLfloat s, GLfloat t, GLfloat r);
void glTexCoord3fv(GLfloat *v);
void glTexCoord3d(GLdouble s, GLdouble t, GLdouble r);
void glTexCoord3dv(GLdouble *v);
void glTexCoord3i(GLint s, GLint t, GLint r);
void glTexCoord3iv(GLint *v);
void glTexCoord3s(GLshort s, GLshort t, GLshort r);
void glTexCoord3sv(GLfloat *v);
void glTexCoord4f(GLfloat s, GLfloat t, GLfloat r,
  GLfloat q);
void glTexCoord4fv(GLfloat *v);
void glTexCoord4d(GLdouble s, GLdouble t, GLdouble
  r, GLdouble q);
void glTexCoord4dv(GLdouble *v);
void glTexCoord4i(GLint s, GLint t, GLint r, GLint q);
void glTexCoord4iv(GLint *v);
void glTexCoord4s(GLshort s, GLshort t, GLshort r,
  GLshort q);
void glTexCoord4sv(GLfloat *v);

```

Description: These functions set the current texture image coordinate in one to four dimensions. Texture coordinates can be updated anytime between `glBegin` and `glEnd`, and correspond to the following `glVertex` call. The texture `q` coordinate is used to scale the `s`, `t`, and `r` coordinate values and by default is 1.0. You can perform any valid matrix operation on texture coordinates by specifying `GL_TEXTURE` as the target of `glMatrixMode`.

Parameters:

`s` GLdouble or GLfloat or GLint or GLshort: The horizontal texture image coordinate.

`t` GLdouble or GLfloat or GLint or GLshort: The vertical texture image coordinate.

`r` GLdouble or GLfloat or GLint or GLshort: The texture image depth coordinate.

`q` GLdouble or GLfloat or GLint or GLshort: The texture image scaling value coordinate.

`v` GLdouble* or GLfloat* or GLint* or GLshort*: An array of values that contain the one, two, three, or four values needed to specify the texture coordinate.

Returns: None.

See Also: `glTexGen`, `glTexImage`, `glTexParameter`

glTexEnv

Purpose: Sets the texture environment parameters.

Include File: `<gl.h>`

Variations:

```
void glTexEnvf(GLenum target, GLenum pname,
  GLfloat param);
void glTexEnvfv(GLenum target, GLenum pname,
  GLfloat *param);
void glTexEnvi(GLenum target, GLenum pname, GLint
  param);
void glTexEnviv(GLenum target, GLenum pname, GLint
  *param);
```

Description: These functions set texture mapping environment parameters. The texture environment is set per texture unit and exists outside the state bound by `glBindTexture`. Thus, all texture objects that can be bound to an active texture unit are influenced by the texture environment.

Parameters:

target GLenum: The texture environment to define. It must be `GL_TEXTURE_ENV` or `GL_TEXTURE_FILTER_CONTROL`.

pname GLenum: The parameter name to define.

When the target is `GL_TEXTURE_FILTER_CONTROL`, the parameter name must be `GL_TEXTURE_LOD_BIAS`, and the parameter is the value that biases the mipmap level of the detail selection.

When the target is `GL_TEXTURE_ENV`, the parameter name may be `GL_TEXTURE_ENV_MODE`, `GL_TEXTURE_ENV_COLOR`, `GL_COMBINE_RGB`, or `GL_COMBINE_ALPHA`.

When the parameter name is `GL_TEXTURE_ENV_COLOR`, the parameter points to an array containing the color values of the texture environment color.

When the parameter name is `GL_TEXTURE_ENV_MODE`, the valid parameters are `GL_REPLACE`, `GL_DECAL`, `GL_MODULATE`, `GL_BLEND`, `GL_ADD`, or `GL_COMBINE`. These environment modes are described in [Table 8.7](#).

param The parameter value. It must be one of the values above (`GL_REPLACE`, `GL_DECAL`, `GL_MODULATE`, `GL_BLEND`, `GL_ADD`, or `GL_COMBINE`), or for `GL_TEXTURE_ENV_COLOR`, an array containing the RGBA color components of the texture environment color.

Returns: None.

See Also: `glTexParameter`

Table 8.7. Texture Environment Modes

Constant	Description
<code>GL_DECAL</code>	Texel values are applied to geometry fragment values. If blending is enabled and the texture contains an alpha channel, the geometry blends through the texture according to the current blend function.

GL_REPLACE	Texel values replace geometry fragment values. If blending is enabled and the texture contains an alpha channel, the texture's alpha values are used to replace the geometry fragment colors in the color buffer.
GL_MODULATE	Texel color values are multiplied by the geometry fragment color values.
GL_ADD	Texel color values are added to the geometry color values.
GL_BLEND	Texel color values are multiplied by the texture environment color.
GL_COMBINE	Texel color values are combined with a second texture unit according to the texture combine function (see the next chapter).

glTexImage

Purpose: Defines a one-, two-, or three-dimensional texture image.

Include File: `<gl.h>`

Variations:

```
void glTexImage1D(GLenum target, GLint level,
  ➔ GLint internalformat,
    GLsizei width, GLint border,
    GLenum format, GLenum type, void
  ➔ *data);

void glTexImage2D(GLenum target, GLint level,
  ➔ GLint internalformat,
    GLsizei width, GLsizei height,
  ➔ GLint border,
    GLenum format, GLenum type, void
  ➔ *data);

void glTexImage3D(GLenum target, GLint level,
  ➔ GLint internalformat,
    GLsizei width, GLsizei height,
  ➔ GLsizei depth, GLint border,
    GLenum format, GLenum type, void
  ➔ *data);
```

Description: This function defines a one-, two-, or three-dimensional texture image. The image data is subject to modes defined with `glPixelMap`, `glPixelStore`, and `glPixelTransfer`.

Parameters:

target `GLenum`: The texture target being specified. Must be `GL_TEXTURE_1D` or `GL_PROXY_TEXTURE_1D` for `glTexImage1D`, `GL_TEXTURE_2D` or `GL_PROXY_TEXTURE_2D` for `glTexImage2D`, `GL_TEXTURE_3D` or `GL_TEXTURE_PROXY_3D` for `glTexImage3D`. For 2D cube maps only, it may also be `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, or `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`.

level `GLint`: The level of detail. Usually, 0 unless mipmapping is used.

internalFormat `GLint`: The internal format of the image data. It may contain the numbers 1–4 to specify the number of color components, or one of the following constants: `GL_ALPHA`, `GL_ALPHA4`, `GL_ALPHA8`, `GL_ALPHA12`, `GL_ALPHA16`, `GL_LUMINANCE`, `GL_LUMINANCE4`, `GL_LUMINANCE8`, `GL_LUMINANCE12`, `GL_LUMINANCE16`, `GL_LUMINANCE_ALPHA`, `GL_LUMINANCE4_ALPHA4`, `GL_LUMINANCE6_ALPHA2`, `GL_LUMINANCE8_ALPHA8`, `GL_LUMINANCE12_ALPHA4`, `GL_LUMINANCE12_ALPHA12`, `GL_LUMINANCE16_ALPHA16`, `GL_INTENSITY`, `GL_INTENSITY4`, `GL_INTENSITY8`, `GL_INTENSTIY12`, `GL_INTENSITY16`, `GL_RGB`, `GL_R3_G3_B2`, `GL_RGB4`, `GL_RGB5`, `GL_RGB8`, `GL_RGB10`, `GL_RGB12`, `GL_RGB16`, `GL_RGBA`, `GL_RGBA2`, `GL_RGBA4`, `GL_RGBA5_A1`, `GL_RGBA8`, `GL_RGB10_A2`, `GL_RGBA12`, `GL_RGBA16`.

width `GLsizei`: The width of the one-, two-, or three-dimensional texture image. This must be a power of 2 but may include a border.

height `GLsizei`: The height of the two- or three-dimensional texture image. This must be a power of 2 but may include a border.

depth `GLsizei`: The depth of a three-dimensional texture image. This must be a power of 2 but may include a border.

border `GLint`: The width of the border. All implementations must support 0, 1, and 2 texel borders.

format `GLenum`: The format of the pixel data. Any texel format type from [Table 8.2](#) is valid.

type `GLenum`: The data type of each texel value. Any data type from [Table 8.3](#) is valid.

pixels `GLvoid *`: The pixel data.

Returns: None.

See Also: `glTexSubImage`, `glCopyTexImage`, `glCopyTexSubImage`

glTexParameter

Purpose: Sets texture mapping parameters.

Include File: `<gl.h>`

Variations:

```
void glTexParameterf(GLenum target, GLenum pname,
→ GLfloat param);
void glTexParameterfv(GLenum target, GLenum pname,
→ GLfloat *param);
void glTexParameteri(GLenum target, GLenum pname,
→ GLint param);
void glTexParameteriv(GLenum target, GLenum pname,
→ GLint *param);
```

Description: This function sets several texture mapping parameters. These parameters are bound to the current texture state that can be made current with `glBindTexture`.

Parameters:

<i>target</i>	GLenum: The texture target for which this parameter applies. Must be one of <code>GL_TEXTURE_1D</code> , <code>GL_TEXTURE_2D</code> , <code>GL_TEXTURE_3D</code> , or <code>GL_TEXTURE_CUBE_MAP</code> .
<i>pname</i>	GLenum: The texturing parameter to set. Valid names are as follows:
	<code>GL_TEXTURE_MIN_FILTER</code> : Specifies the texture image minification (reduction) method or filter. Any texture filter from Table 8.4 may be used.
	<code>GL_TEXTURE_MAX_FILTER</code> : Specifies the texture image magnification (enlargement) method or filter. Any texture filter from Table 8.4 may be used.
	<code>GL_TEXTURE_MAX_LOD</code> : Specifies the maximum texture LOD to be used for mipmapping.
	<code>GL_TEXTURE_MIN_LOD</code> : Specifies the minimum texture LOD to be used for mipmapping.
	<code>GL_TEXTURE_BASE_LEVEL</code> : Specifies the minimum texture LOD to be loaded.
	<code>GL_TEXTURE_MAX_LEVEL</code> : Specifies the maximum texture LOD to be loaded.
	<code>GL_TEXTURE_WRAP_S</code> : Specifies handling of texture s coordinates outside the range of 0.0 to 1.0 (<code>GL_CLAMP</code> , <code>GL_CLAMP_TO_EDGE</code> , <code>GL_CLAMP_TO_BORDER</code> , <code>GL_REPEAT</code> , <code>GL_MIRRORED_REPEAT</code>).
	<code>GL_TEXTURE_WRAP_T</code> : Specifies handling of texture t coordinates outside the range of 0.0 to 1.0 (<code>GL_CLAMP</code> , <code>GL_CLAMP_TO_EDGE</code> , <code>GL_CLAMP_TO_BORDER</code> , <code>GL_REPEAT</code> , <code>GL_MIRRORED_REPEAT</code>).
	<code>GL_TEXTURE_WRAP_R</code> : Specifies handling of texture r coordinates outside the range of 0.0 to 1.0 (<code>GL_CLAMP</code> , <code>GL_CLAMP_TO_EDGE</code> , <code>GL_CLAMP_TO_BORDER</code> , <code>GL_REPEAT</code>).
	<code>GL_BORDER_COLOR</code> : Specifies a border color for textures without borders.
	<code>GL_GENERATE_MIPMAP</code> : Specifies whether mipmap LODs should be automatically built (<code>GL_TRUE</code> = yes).
	<code>GL_TEXTURE_PRIORITY</code> : Sets the texture priority for this texture. It must be a clamped value in the range 0.0 to 1.0.
	<code>GL_DEPTH_TEXTURE_MODE</code> : Sets the depth texture mode (see Chapter 18).
	<code>GL_TEXTURE_COMPARE_MODE</code> : Sets the texture comparison mode (see Chapter 18).
	<code>GL_TEXTURE_COMPARE_FUNC</code> : Sets the texture comparison function (see Chapter 18).
<i>param</i>	<code>GLfloat</code> or <code>GLfloat*</code> or <code>GLint</code> or <code>GLint*</code> : Value of the parameter specified by <i>pname</i> .
Returns:	None.
See Also:	<code>glTexEnv</code> , <code>glTexGen</code> , <code>glBindTexture</code>

glTexSubImage

Purpose: Replaces a portion of an existing texture map.

Include File: `<gl.h>`

Variations:

```

void glTexSubImage1D(GLenum target, GLint level,
                     GLint xOffset,
                     GLsizei width,
                     GLenum format,
                     → GLenum type, const GLvoid *data);
void glTexSubImage2D(GLenum target, GLint level,
                     GLint xOffset,
                     → GLint yOffset,
                     GLsizei width,
                     → GLsizei height,
                     GLenum format,
                     → GLenum type, const GLvoid *data);
void glTexSubImage3D(GLenum target, GLint level,
                     GLint xOffset,
                     → GLint yOffset, GLint zOffset,
                     GLsizei width,
                     → GLsizei height, GLsizei depth,
                     GLenum format,
                     → GLenum type, const GLvoid *data);

```

Description: This function replaces a portion of an existing one-, two-, or three-dimensional texture map. Updating all or part of an existing texture map may be considerably faster than reloading a texture image with `glTexImage`. You cannot perform an initial texture load with this function; it is used only to update an existing texture.

Parameters:

`target` GLenum: The texture target. Must be `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, or `GL_TEXTURE_3D`.

`level` GLint: Mipmap level to be updated.

`xOffset` GLint: Offset within the existing one-, two-, or three-dimensional texture in the x direction to begin the update.

`yOffset` GLint: Offset within the existing two- or three-dimensional texture in the y direction to begin the update.

`zOffset` GLint: Offset within the existing three-dimensional texture in the z direction to begin the update.

`width` GLsizei: The width of the one-, two-, or three-dimensional texture data being updated.

`height` GLsizei: The height of the two- or three-dimensional texture data being updated.

`depth` GLsizei: The depth of the three-dimensional texture data being updated.

`format` GLenum: Any valid texture format. See [Table 8.2](#).

`type` GLenum: Any valid pixel data type. See [Table 8.3](#).

`data` const void*: Pointer to the data that is being used to update the texture target.

Returns: None.

See Also: `glTexImage`, `glCopyTexSubImage`

gluBuildMipmapLevels**Purpose:** Automatically generates and updates a range of mipmap levels.**Include File:** <gl.h>**Variations:**

```

int gluBuild1DMipmapLevels(GLenum target, GLint
  ↪ internalFormat,
                           GLint width,
                           GLenum
  ↪ format, GLenum type, GLint level,
                           GLint base,
  ↪ GLint max, const void *data);
int gluBuild2DMipmapLevels(GLenum target, GLint
  ↪ internalFormat,
                           GLint width,
  ↪ GLint height,
                           GLenum
  ↪ format, GLenum type, GLint level,
                           GLint base,
  ↪ GLint max, const void *data);
int gluBuild3DMipmapLevels(GLenum target, GLint
  ↪ internalFormat,
                           GLint width,
  ↪ GLint height, GLint depth,
                           GLenum
  ↪ format, GLenum type, GLint level,
                           GLint base,
  ↪ GLint max, const void *data);

```

Description: This function uses `gluScaleImage` to automatically scale and load a series of mipmap levels based on an initial level provided. The advantage to using this function is that it allows only a selected range of mip levels to be updated.**Parameters:**

`target` **GLenum:** The texture target. Must be `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, or `GL_TEXTURE_3D`.

`internal Format` **GLint:** Any valid texture internal format recognized by `glTexImage`.

`width` **GLint:** The width of the one-, two-, or three-dimensional texture source.

`height` **GLint:** The height of the two- or three-dimensional texture source.

`depth` **GLint:** The depth of the three-dimensional texture source.

`format` **GLenum:** Any valid texel format. See [Table 8.2](#).

`type` **GLenum:** Any valid texel data type. See [Table 8.3](#).

`level` **GLint:** The base mipmap level being specified by the data in `data`.

`base` **GLint:** The starting mip level to begin mipmap generation.

max *GLint*: The highest mip level (smallest image) to generate.

data *void**: The supplied texture image data.

Returns: None.

See Also: [gluBuildMipmaps](#)

gluBuildMipmaps

Purpose: Automatically creates and loads a set of complete mipmaps.

Include File: `<gl.h>`

Variations:

```
int gluBuild1DMipmaps(GLenum target, GLint
  ↪ internalFormat,
              GLint width,
              GLenum format,
  ↪ GLenum type, const void *data);
int gluBuild2DMipmaps(GLenum target, GLint
  ↪ internalFormat,
              GLint width,
  ↪ GLint height,
              GLenum format,
  ↪ GLenum type, const void *data);
int gluBuild3DMipmaps(GLenum target, GLint
  ↪ internalFormat,
              GLint width,
  ↪ GLint height, GLint depth,
              GLenum format,
  ↪ GLenum type, const void *data);
```

Description: This function takes the texture data given for the base mipmapped texture level and automatically scales the image repeatedly and loads each mip level in turn. This work is done by the client CPU and not by the OpenGL implementation, and thus may be a time-consuming operation:

Parameters:

target *GLenum*: The texture target. Must be `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, or `GL_TEXTURE_3D`.

internalFormat *GLint*: Any valid texture internal format recognized by `glTexImage`.

width *GLint*: The width of the one-, two-, or three-dimensional texture source.

height *GLint*: The height of the two- or three-dimensional texture source.

depth *GLint*: The depth of the three-dimensional texture source.

format *GLenum*: Any valid texel format. See [Table 8.2](#).

type *GLenum*: Any valid texel data type. See [Table 8.3](#).

data `void*`: The base mipmap level texture data.

Returns: None:

See Also: `gluBuildMipmapLevels`

Chapter 9. Texture Mapping: Beyond the Basics

by Richard S. Wright, Jr.

WHAT YOU'LL LEARN IN THIS CHAPTER:

How To	Functions You'll Use
Add specular highlights to textured objects	<code>glLightModel</code> , <code>glSecondaryColor</code>
Use anisotropic texture filtering	<code>glTexParameterf</code>
Load and use compressed textures	<code>glCompressedTexImage</code> , <code>glCompressedTexSubImage</code>

Texture mapping is perhaps one of the most exciting features of OpenGL (well, close behind shaders anyway!) and is heavily relied on in the games and simulation industry. In [Chapter 8](#), "Texture Mapping: The Basics," you learned the basics of loading and applying texture maps to geometry. In this chapter, we expand on this knowledge and cover some of the finer points of texture mapping in OpenGL.

Secondary Color

Applying to geometry, in regards to how lighting works, causes a hidden and often undesirable side effect. In general, you set the texture environment to `GL_MODULATE`, causing lit geometry to be combined with the texture map in such a way that the textured geometry also appears lit. Normally, OpenGL performs lighting calculations and calculates the color of individual fragments according to the standard light model. These fragment colors are then combined with the filtered texels being applied to the geometry. However, this process has the side effect of greatly reducing the visibility of specular highlights on the geometry.

For example, [Figure 9.1](#) shows the original lit SPHEREWORLD sample from [Chapter 5](#), "Color, Materials, and Lighting: The Basics." In this figure, you can see clearly the specular highlights reflecting off the surface of the torus. In contrast, [Figure 9.2](#) shows the SPHEREWORLD sample from the preceding chapter. In this figure, you can see the effects of having the texture applied after the lighting has been added.

Figure 9.1. Original SPHEREWORLD torus with specular highlights.

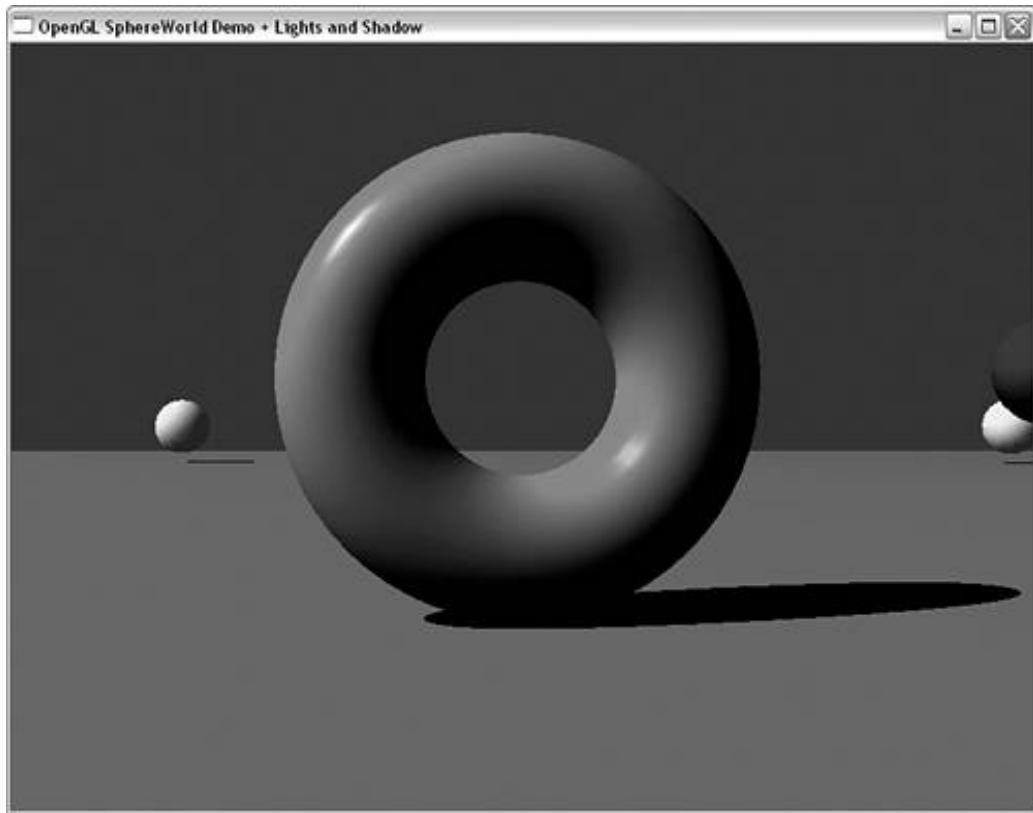
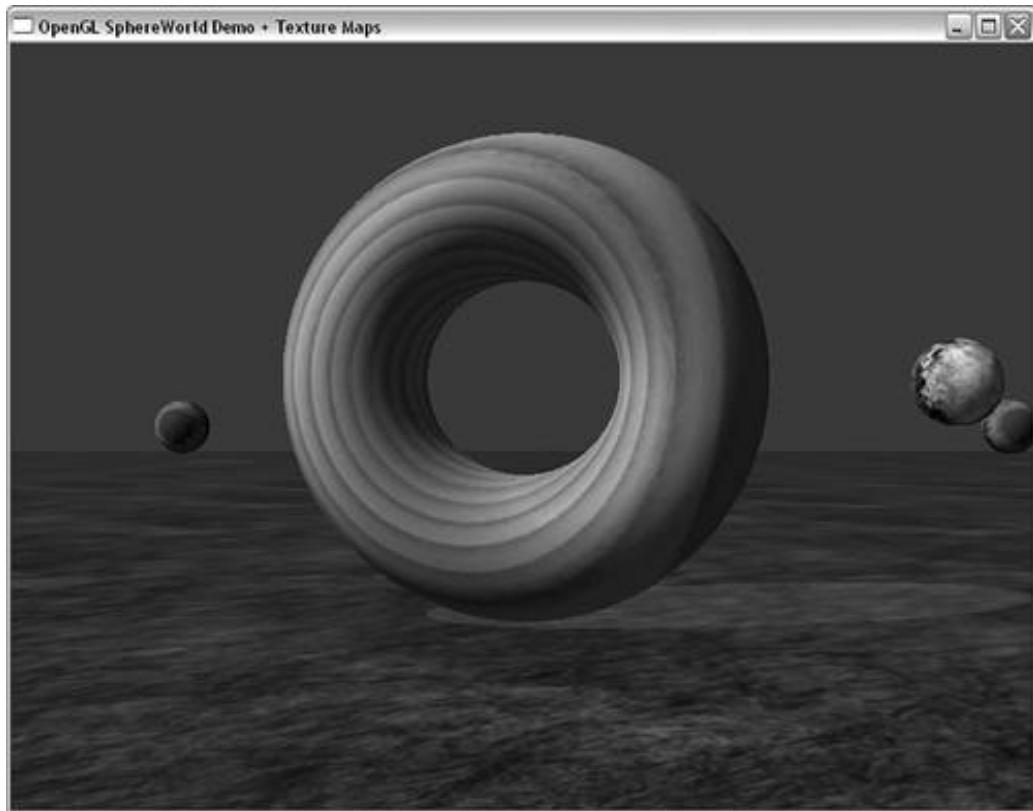


Figure 9.2. Textured torus with muted highlights.



The solution to this problem is to apply the specular highlights after texturing. This technique, called the *secondary specular color*, can be applied manually or automatically calculated by the lighting model. Usually, you do this using the normal OpenGL lighting model and simply turn it on using `glLightModel`, as shown here:

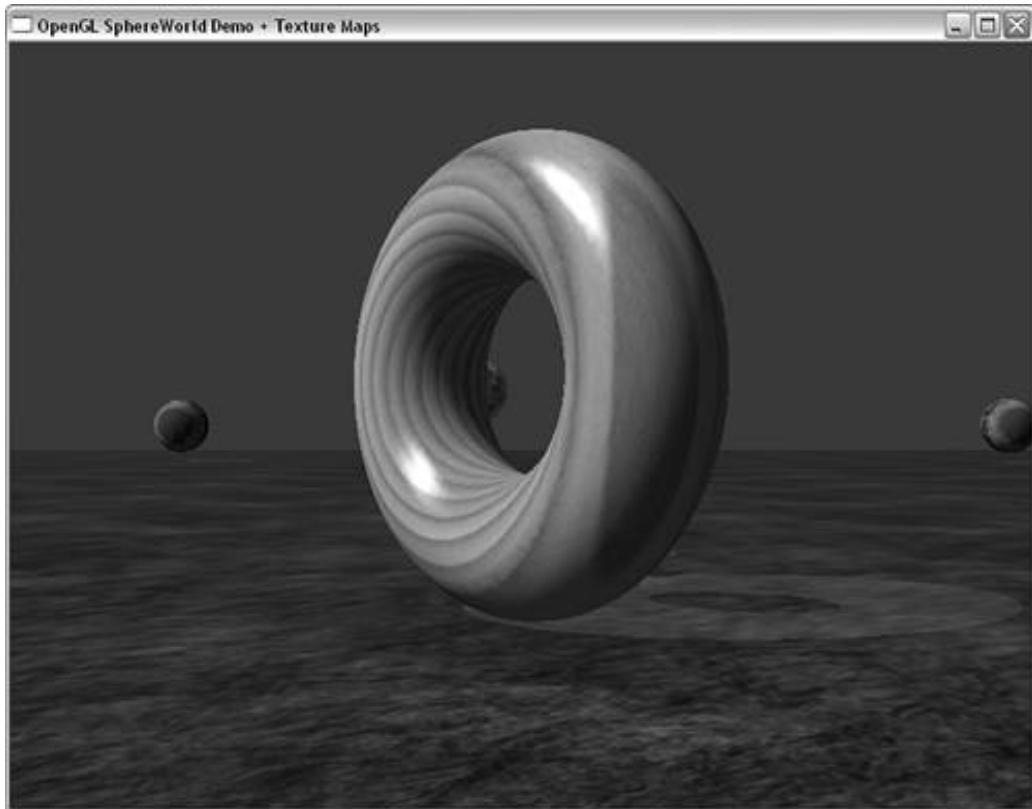
```
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR);
```

You can switch back to the normal lighting model by specifying `GL_SINGLE_COLOR` for the light model parameter:

```
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_COLOR_SINGLE);
```

Figure 9.3 shows the output from this chapter's version of SPHEREWORLD with the restored specular highlights on the torus. We do not provide a listing for this sample because it simply contains the addition of the preceding single line of code.

Figure 9.3. Highlights restored to textured torus.



You can also directly specify a secondary color after texturing when you are not using lighting (lighting is disabled) using the `glSecondaryColor` function. This function comes in many variations just as `glColor` does and is fully documented in the reference section. You should also note that if you specify a secondary color, you must also explicitly enable the use of the secondary color by enabling the `GL_COLOR_SUM` flag:

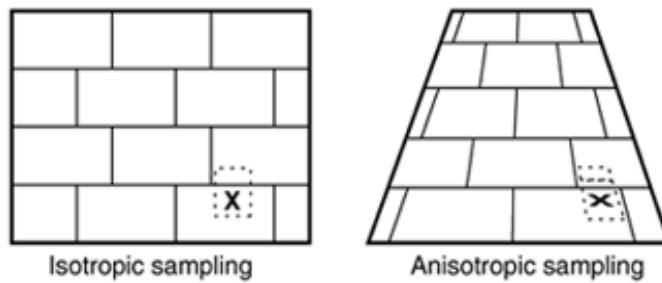
```
 glEnable(GL_COLOR_SUM);
```

Anisotropic Filtering

Anisotropic texture filtering is not a part of the core OpenGL specification, but it is a widely supported extension that can dramatically improve the quality of texture filtering operations. Texture filtering was covered in the preceding chapter, where you learned about the two basic texture filters: nearest neighbor (`GL_NEAREST`) and linear (`GL_LINEAR`). When a texture map is filtered, OpenGL uses the texture coordinates to figure out where in the texture map a particular fragment of geometry falls. The texels immediately around that position are then sampled using either the `GL_NEAREST` or `GL_LINEAR` filtering operations.

This process works perfectly when the geometry being textured is viewed directly perpendicular to the viewpoint, as shown to the left in [Figure 9.4](#). However, when the geometry is viewed from an angle more oblique to the point of view, a regular sampling of the surrounding texels results in the loss of some information in the texture (it looks blurry!). A more realistic and accurate sample would be elongated along the direction of the plane containing the texture. This result is shown to the right in [Figure 9.4](#). Taking this viewing angle into account for texture filtering is called *anisotropic filtering*.

Figure 9.4. Normal texture sampling versus anisotropic sampling.



You can apply anisotropic filtering to any of the basic or mipmapped texture filtering modes; applying it requires three steps. First, you must determine whether the extension is supported. You can do this by querying for the extension string `GL_EXT_texture_filter_anisotropic`. You can use the `glTools` function named `glIsExtensionSupported` for this task:

```
if(glIsExtensionSupported("GL_EXT_texture_filter_anisotropic"))
    // Set Flag that extension is supported
```

After you determine that this extension is supported, you can find the maximum amount of *anisotropy* supported. You can query for it using `glGetFloatv` and the parameter

`GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT`:

```
GLfloat fLargest;
. . .
. . .
glGetFloatv(GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT, &fLargest);
```

The larger the amount of anisotropy applied, the more texels are sampled along the direction of greatest change (along the strongest point of view). A value of 1.0 represents normal texture filtering (called *isotropic* filtering). Bear in mind that anisotropic filtering is not free. The extra amount of work, including other texels, can sometimes result in substantial performance penalties. On modern hardware, this feature is getting quite fast and is becoming a standard feature of popular games, animation, and simulation programs.

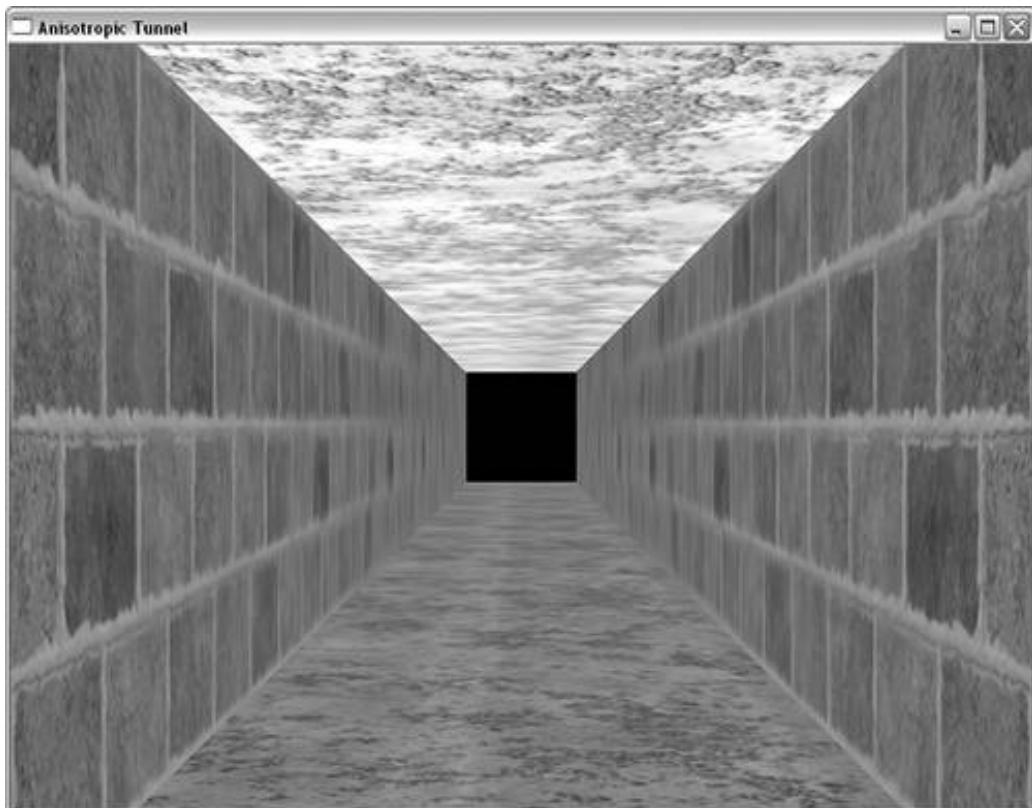
Finally, you set the amount of anisotropy you want applied using `glTexParameter` and the constant `GL_TEXTURE_MAX_ANISOTROPY_EXT`. For example, using the preceding code, if you want the maximum amount of anisotropy applied, you would call `glTexParameter` as follows:

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT, fLargest);
```

This modifier is applied per texture object just like the standard filtering parameters.

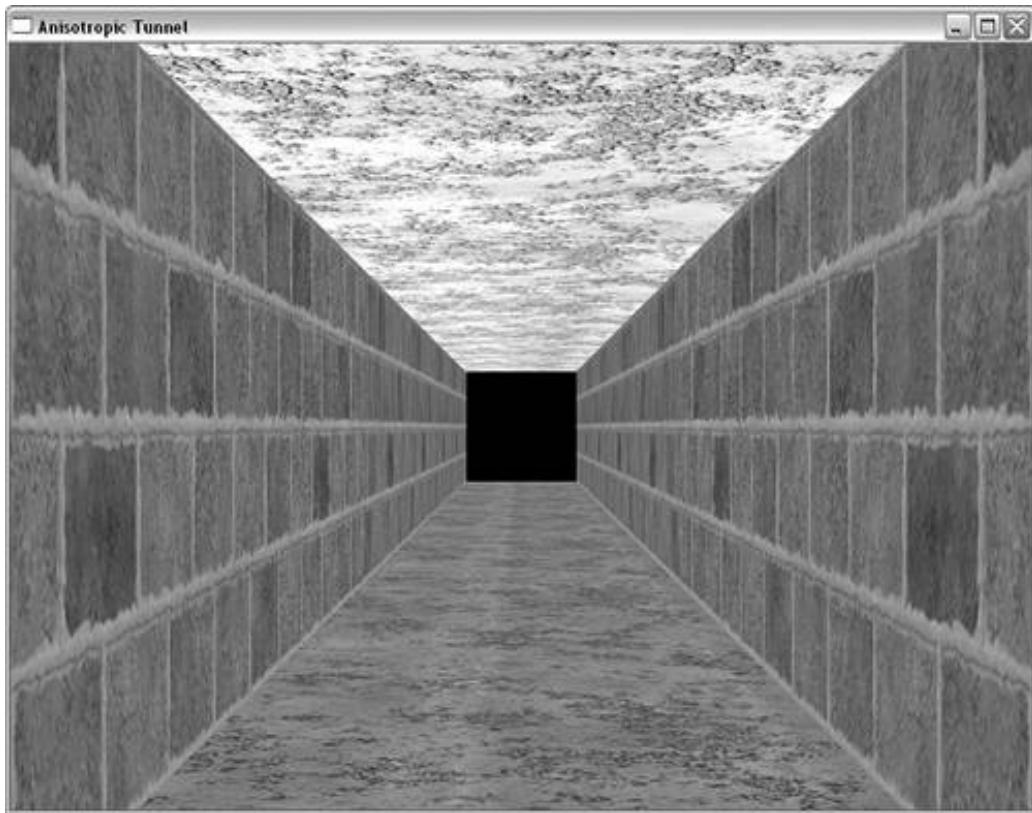
The sample program ANISOTROPIC provides a striking example of anisotropic texture filtering in action. This program displays a tunnel with walls, a floor, and ceiling geometry. The arrow keys move your point of view (or the tunnel) back and forth along the tunnel interior. A right mouse click brings up a menu that allows you to select from the various texture filters, and turn on and off anisotropic filtering. [Figure 9.5](#) shows the tunnel using trilinear filtered mipmapping. Notice how blurred the patterns become in the distance, particularly with the bricks.

Figure 9.5. ANISOTROPIC tunnel sample with trilinear filtering.



Now compare [Figure 9.5](#) with [Figure 9.6](#), where anisotropic filtering has been enabled. The mortar between the bricks is now clearly visible all the way to the end of the tunnel. In fact, anisotropic filtering can also greatly reduce the visible mipmap transition patterns for the `GL_LINEAR_MIPMAP_NEAREST` and `GL_NEAREST_MIPMAP_NEAREST` mipmapped filters.

Figure 9.6. ANISOTROPIC tunnel sample with anisotropic filtering.



Texture Compression

Texture mapping can add incredible realism to any 3D rendered scene, with a minimal cost in vertex processing. One drawback to using textures, however, is that they require a lot of memory to store and process. Early attempts at texture compression were crudely storing textures as JPG files and decompressing the textures when loaded before calling `glTexImage`. These attempts saved disk space or reduced the amount of time required to transmit the image over the network (such as the Internet), but did little to alleviate the storage requirements of texture images loaded into graphics hardware memory.

Native support for texture compression was added to OpenGL with version 1.3. Earlier versions of OpenGL may also support texture compression via extension functions of the same name. You can test for this extension by using the `GL_ARB_texture_compression` string.

Texture compression support in OpenGL hardware can go beyond simply allowing you to load a compressed texture; in most implementations, the texture data stays compressed even in the graphics hardware memory. This allows you to load more texture into less memory and can significantly improve texturing performance due to fewer texture swaps (moving textures around) and fewer memory accesses during texture filtering.

Compressing Textures

Texture data does not have to be initially compressed to take advantage of OpenGL support for compressed textures. You can request that OpenGL compress a texture image when loaded by using one of the values in [Table 9.1](#) for the `internalFormat` parameter of any of the `glTexImage` functions.

Table 9.1. Compressed Texture Formats

Compressed Format	Base Internal Format
GL_COMPRESSED_ALPHA	GL_ALPHA
GL_COMPRESSED_LUMINANCE	GL_LUMINANCE
GL_COMPRESSED_LUMINANCE_ALPHA	GL_LUMINANCE_ALPHA
GL_COMPRESSED_INTENSITY	GL_INTENSITY
GL_COMPRESSED_RGB	GL_RGB
GL_COMPRESSED_RGBA	GL_RGBA

Compressing images this way adds a bit of overhead to texture loads but can increase texture performance due to the more efficient usage of texture memory. If, for some reason, the texture cannot be compressed, OpenGL uses the base internal format listed instead and loads the texture uncompressed.

When you attempt to load and compress a texture in this way, you can find out whether the texture was successfully compressed by using `glGetTexLevelParameteriv` with `GL_TEXTURE_COMPRESSED` as the parameter name:

```
GLint compFlag;
. . .
glGetTexLevelParameteriv(GL_TEXTURE_2D, 0, GL_TEXTURE_COMPRESSED, &compFlag);
```

The `glGetTexLevelParameteriv` function accepts a number of new parameter names pertaining to compressed textures. These parameters are listed in [Table 9.2](#).

Table 9.2. Compressed Texture Parameters Retrieved with `glGetTexLevelParameter`

Parameter	Returns
<code>GL_TEXTURE_COMPRESSED</code>	The value 1 if the texture is compressed, 0 if not
<code>GL_TEXTURE_COMPRESSED_IMAGE_SIZE</code>	The size in bytes of the compressed texture
<code>GL_TEXTURE_INTERNAL_FORMAT</code>	The compression format used
<code>GL_NUM_COMPRESSED_TEXTURE_FORMATS</code>	The number of supported compressed texture formats
<code>GL_COMPRESSED_TEXTURE_FORMATS</code>	An array of constant values corresponding to each supported compressed texture format
<code>GL_TEXTURE_COMPRESSION_HINT</code>	The value of the texture compression hint (<code>GL_NICEST</code> / <code>GL_FASTEST</code>)

When textures are compressed using the values listed in [Table 9.1](#), OpenGL chooses the most appropriate texture compression format. You can use `glHint` to specify whether you want OpenGL to choose based on the fastest or highest quality algorithm:

```
glHint(GL_TEXTURE_COMPRESSION_HINT, GL_FASTEST);
glHint(GL_TEXTURE_COMPRESSION_HINT, GL_NICEST);
glHint(GL_TEXTURE_COMPRESSION_HINT, GL_DONT_CARE);
```

The exact compression format varies from implementation to implementation. You can obtain a count of compression formats and a list of the values by using `GL_NUM_COMPRESSED_TEXTURE_FORMATS` and `GL_COMPRESSED_TEXTURE_FORMATS`. To check for support for a specific set of compressed texture formats, you need to check for a specific extension for those formats. For example, one of the most popular (on both PC and Mac) is the `GL_EXT_texture_compression_s3tc` texture compression format. If this extension is supported, the compressed texture formats listed in [Table 9.3](#) are all supported (these constants are defined in `glext.h`), but only for two-dimensional textures.

Table 9.3. Compression Formats for `GL_EXT_texture_compression_s3tc`

Format	Description
<code>GL_COMPRESSED_RGB_S3TC_DXT1</code>	RGB data is compressed; alpha is always 1.0.
<code>GL_COMPRESSED_RGBA_S3TC_DXT1</code>	RGB data is compressed; alpha is either 1.0 or 0.0.
<code>GL_COMPRESSED_RGBA_S3TC_DXT3</code>	RGB data is compressed; alpha is stored as 4 bits.
<code>GL_COMPRESSED_RGBA_S3TC_DXT5</code>	RGB data is compressed; alpha is a weighted average of 8-bit values.

Loading Compressed Textures

Using the functions in the preceding section, you can have OpenGL compress textures in a natively supported format, retrieve the compressed data with the `glGetCompressedTexImage` function (identical to the `glGetTexImage` function in the preceding chapter), and save it to disk. On subsequent loads, the raw compressed data can be used, resulting in substantially faster texture loads.

To load precompressed texture data, use one of the following functions:

```
void glCompressedTexImage1D(GLenum target, GLint level, GLenum internalFormat,
                           GLsizei width,
                           GLint border, GLsizei imageSize, void *data);
void glCompressedTexImage2D(GLenum target, GLint level, GLenum internalFormat,
                           GLsizei width, GLsizei height,
                           GLint border, GLsizei imageSize, void *data);
void glCompressedTexImage3D(GLenum target, GLint level, GLenum internalFormat,
                           GLsizei width, GLsizei height, GLsizei depth,
                           GLint border, GLsizei imageSize, GLvoid *data);
```

These functions are virtually identical to the `glTexImage` functions from the preceding chapter. The only difference is that the `internalFormat` parameter must specify a supported compressed texture image. If the implementation supports the `GL_EXT_texture_compression_s3tc` extension, this would be one of the values from [Table 9.3](#). There is also a corresponding set of `glCompressedTexSubImage` functions for updating a portion or all of an already-loaded texture that mirrors the `glTexSubImage` functionality from the preceding chapter.

Texture Coordinate Generation

In [Chapter 8](#), you learned that textures are mapped to geometry using texture coordinates. Often, when you are loading models (see [Chapter 11](#), "It's All About the Pipeline: Faster Geometry Throughput"), texture coordinates are provided for you. If necessary, you can easily map texture coordinates manually to some surfaces such as spheres or flat planes. Sometimes, however, you may have a complex surface for which it is not so easy to manually derive the coordinates. OpenGL can automatically generate texture coordinates for you within certain limitations.

Texture coordinate generation is enabled on the S, T, R, and Q texture coordinates using `glEnable`:

```
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_GEN_R);
glEnable(GL_TEXTURE_GEN_Q);
```

When texture coordinate generation is enabled, any calls to `glTexCoord` are ignored, and OpenGL calculates the texture coordinates for each vertex for you. In the same manner that texture coordinate generation is turned on, you turn it off by using `glDisable`:

```
glDisable(GL_TEXTURE_GEN_S);
glDisable(GL_TEXTURE_GEN_T);
glDisable(GL_TEXTURE_GEN_R);
glDisable(GL_TEXTURE_GEN_Q);
```

You set the function or method used to generate texture coordinates with the following functions:

```
void glTexGenf(GLenum coord, GLenum pname, GLfloat param);
void glTexGenfv(GLenum coord, GLenum pname, GLfloat *param);
```

The first parameter, `coord`, specifies which texture coordinate this function sets. It must be either `GL_S`, `GL_T`, `GL_R`, or `GL_Q`. The second parameter, `pname`, must be either `GL_TEXTURE_GEN_MODE`, `GL_OBJECT_PLANE`, or `GL_EYE_PLANE`. The last parameter sets the values of the texture generation function or mode. Note that integer (`GLint`) and double (`GLdouble`) versions of these functions are also used.

The sample program `TEXGEN` is presented in [Listing 9.1](#). This program displays a torus that can be manipulated (rotated around) using the arrow keys. A right-click brings up a context menu that allows you to select from the first three texture generation modes we will discuss: Object Linear, Eye Linear, and Sphere Mapping.

Listing 9.1. Source Code for the `TEXGEN` Sample Program

```
#include "../../Common/OpenGLSB.h" // System and OpenGL Stuff
#include "../../Common/gltools.h" // gltools library
// Rotation amounts
static GLfloat xRot = 0.0f;
static GLfloat yRot = 0.0f;
GLuint toTextures[2]; // Two texture objects
int iRenderMode = 3; // Sphere Mapped is default
////////////////////////////// Reset flags as appropriate in response to menu selections
void ProcessMenu(int value)
{
    // Projection plane
    GLfloat zPlane[] = { 0.0f, 0.0f, 1.0f, 0.0f };
    // Store render mode
```

```

iRenderMode = value;
// Set up textgen based on menu selection
switch(value)
{
    case 1:
        // Object Linear
        glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
        glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
        glTexGenfv(GL_S, GL_OBJECT_PLANE, zPlane);
        glTexGenfv(GL_T, GL_OBJECT_PLANE, zPlane);
        break;
    case 2:
        // Eye Linear
        glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
        glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
        glTexGenfv(GL_S, GL_EYE_PLANE, zPlane);
        glTexGenfv(GL_T, GL_EYE_PLANE, zPlane);
        break;
    case 3:
    default:
        // Sphere Map
        glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
        glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
        break;
}
glutPostRedisplay();      // Redisplay
}

// Called to draw scene
void RenderScene(void)
{
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // Switch to orthographic view for background drawing
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    gluOrtho2D(0.0f, 1.0f, 0.0f, 1.0f);
    glMatrixMode(GL_MODELVIEW);
    glBindTexture(GL_TEXTURE_2D, toTextures[1]);      // Background texture
    // We will specify texture coordinates
    glDisable(GL_TEXTURE_GEN_S);
    glDisable(GL_TEXTURE_GEN_T);
    // No depth buffer writes for background
    glDepthMask(GL_FALSE);
    // Background image
    glBegin(GL_QUADS);
        glTexCoord2f(0.0f, 0.0f);
        glVertex2f(0.0f, 0.0f);

        glTexCoord2f(1.0f, 0.0f);
        glVertex2f(1.0f, 0.0f);
        glTexCoord2f(1.0f, 1.0f);
        glVertex2f(1.0f, 1.0f);
        glTexCoord2f(0.0f, 1.0f);
        glVertex2f(0.0f, 1.0f);
    glEnd();
    // Back to 3D land
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    glMatrixMode(GL_MODELVIEW);
}

```

```

// Turn texgen and depth writing back on
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glDepthMask(GL_TRUE);
// May need to switch to stripe texture
if(iRenderMode != 3)
    glBindTexture(GL_TEXTURE_2D, toTextures[0]);
// Save the matrix state and do the rotations
glPushMatrix();
glTranslatef(0.0f, 0.0f, -2.0f);
glRotatef(xRot, 1.0f, 0.0f, 0.0f);
glRotatef(yRot, 0.0f, 1.0f, 0.0f);
// Draw the tours
gltDrawTorus(0.35, 0.15, 61, 37);
// Restore the matrix state
glPopMatrix();
// Display the results
glutSwapBuffers();
}

///////////////////////////////
// This function does any needed initialization on the rendering
// context.
void SetupRC()
{
    GLbyte *pBytes;                      // Texture bytes
    GLint iComponents, iWidth, iHeight; // Texture sizes
    GLenum eFormat;                     // Texture format
    glEnable(GL_DEPTH_TEST);           // Hidden surface removal
    glFrontFace(GL_CCW);              // Counterclockwise polygons face out
    glEnable(GL_CULL_FACE);            // Do not calculate inside of jet
    // White background
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f );
    // Decal texture environment
    glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
    // Two textures
    glGenTextures(2, toTextures);
    /////////////////////////////////
    // Load the main texture
    glBindTexture(GL_TEXTURE_2D, toTextures[0]);
    pBytes = gltLoadTGA("stripes.tga", &iWidth, &iHeight,
                        &iComponents, &eFormat);
    glTexImage2D(GL_TEXTURE_2D, 0, iComponents, iWidth, iHeight, 0,
                eFormat, GL_UNSIGNED_BYTE, (void *)pBytes);
    free(pBytes);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glEnable(GL_TEXTURE_2D);

    /////////////////////////////////
    // Load environment map
    glBindTexture(GL_TEXTURE_2D, toTextures[1]);
    pBytes = gltLoadTGA("Environment.tga", &iWidth, &iHeight,
                        &iComponents, &eFormat);
    glTexImage2D(GL_TEXTURE_2D, 0, iComponents, iWidth, iHeight, 0,
                eFormat, GL_UNSIGNED_BYTE, (void *)pBytes);
    free(pBytes);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
}

```

```

glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glEnable(GL_TEXTURE_2D);
// Turn on texture coordinate generation
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
// Sphere Map will be the default
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
}

///////////
// Handle arrow keys
void SpecialKeys(int key, int x, int y)
{
    if(key == GLUT_KEY_UP)
        xRot-= 5.0f;
    if(key == GLUT_KEY_DOWN)
        xRot += 5.0f;
    if(key == GLUT_KEY_LEFT)
        yRot -= 5.0f;
    if(key == GLUT_KEY_RIGHT)
        yRot += 5.0f;
    if(key > 356.0f)
        xRot = 0.0f;

    if(key < -1.0f)
        xRot = 355.0f;
    if(key > 356.0f)
        yRot = 0.0f;
    if(key < -1.0f)
        yRot = 355.0f;
    // Refresh the Window
    glutPostRedisplay();
}

///////////
// Reset projection and light position
void ChangeSize(int w, int h)
{
    GLfloat fAspect;
    // Prevent a divide by zero
    if(h == 0)
        h = 1;
    // Set Viewport to window dimensions
    glViewport(0, 0, w, h);
    // Reset coordinate system
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    fAspect = (GLfloat) w / (GLfloat) h;
    gluPerspective(45.0f, fAspect, 1.0f, 225.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

///////////
// Program Entry Point
int main(int argc, char* argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(800,600);
    glutCreateWindow("Texture Coordinate Generation");
    glutReshapeFunc(ChangeSize);
    glutSpecialFunc(SpecialKeys);
}

```

```

glutDisplayFunc(RenderScene);
SetupRC();
// Create the Menu
glutCreateMenu(ProcessMenu);
glutAddMenuEntry("Object Linear",1);
glutAddMenuEntry("Eye Linear",2);
glutAddMenuEntry("Sphere Map",3);
glutAttachMenu(GLUT_RIGHT_BUTTON);
glutMainLoop();
// Don't forget the texture objects
glDeleteTextures(2, toTextures);
return 0;
}

```

Object Linear Mapping

When the texture generation mode is set to `GL_OBJECT_LINEAR`, texture coordinates are generated using the following function:

```
coord = P1*X + P2*Y + P3*Z + P4*W
```

The X, Y, Z, and W values are the vertex coordinates from the object being textured, and the P1–P4 values are the coefficients for a plane equation. The texture coordinates are then projected onto the geometry from the perspective of this plane. For example, to project texture coordinates for S and T from the plane Z = 0, we would use the following code from the `TEXGEN` sample program:

```

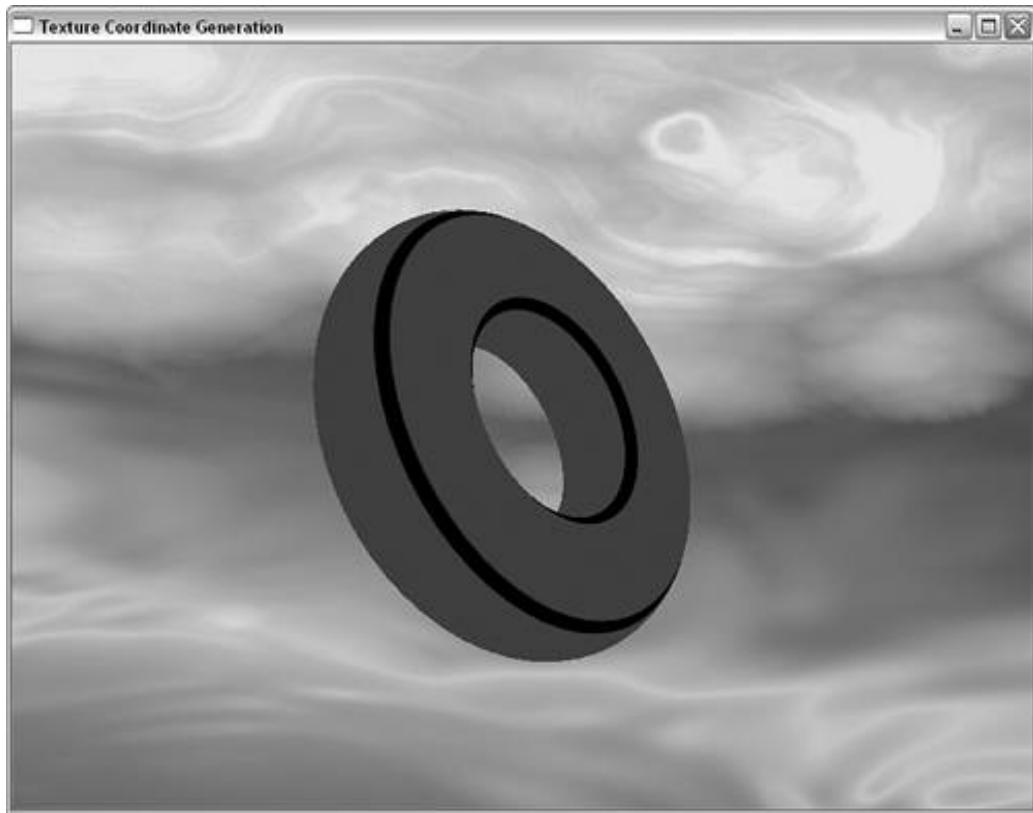
// Projection plane
GLfloat zPlane[] = { 0.0f, 0.0f, 1.0f, 0.0f };
. . .
. . .
// Object Linear
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGenfv(GL_S, GL_OBJECT_PLANE, zPlane);
glTexGenfv(GL_T, GL_OBJECT_PLANE, zPlane);

```

Note that the texture coordinate generation function can be different for each coordinate. Here, we simply use the same function for both the S and T coordinates.

This technique maps the texture to the object in object coordinates, regardless of any ModelView transformation in effect. [Figure 9.7](#) shows the output for `TEXGEN` when the Object Linear mode is selected. No matter how you reorient the torus, the mapping remains fixed to the geometry.

Figure 9.7. Torus mapped with object linear coordinates.



Eye Linear Mapping

When the texture generation mode is set to `GL_EYE_LINEAR`, texture coordinates are generated in a similar manner to `GL_OBJECT_LINEAR`. The coordinate generation looks the same, except that now the X, Y, Z, and W coordinates indicate the location of the point of view (where the camera or eye is located). The plane equation coefficients are also inverted before being applied to the equation.

The texture, therefore, is basically projected from the plane onto the geometry. As the geometry is transformed by the ModelView matrix, the texture will appear to slide across the surface. We set up this capability with the following code from the `TEXGEN` sample program:

```
// Projection plane
GLfloat zPlane[] = { 0.0f, 0.0f, 1.0f, 0.0f };
. . .
. . .
// Eye Linear
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
glTexGenfv(GL_S, GL_EYE_PLANE, zPlane);
glTexGenfv(GL_T, GL_EYE_PLANE, zPlane);
```

The output of the `TEXGEN` program when the Eye Linear menu option is selected is shown in [Figure 9.8](#). As you move the torus around with the arrow keys, note how the projected texture slides about on the geometry.

Figure 9.8. An example of eye linear texture mapping.



Sphere Mapping

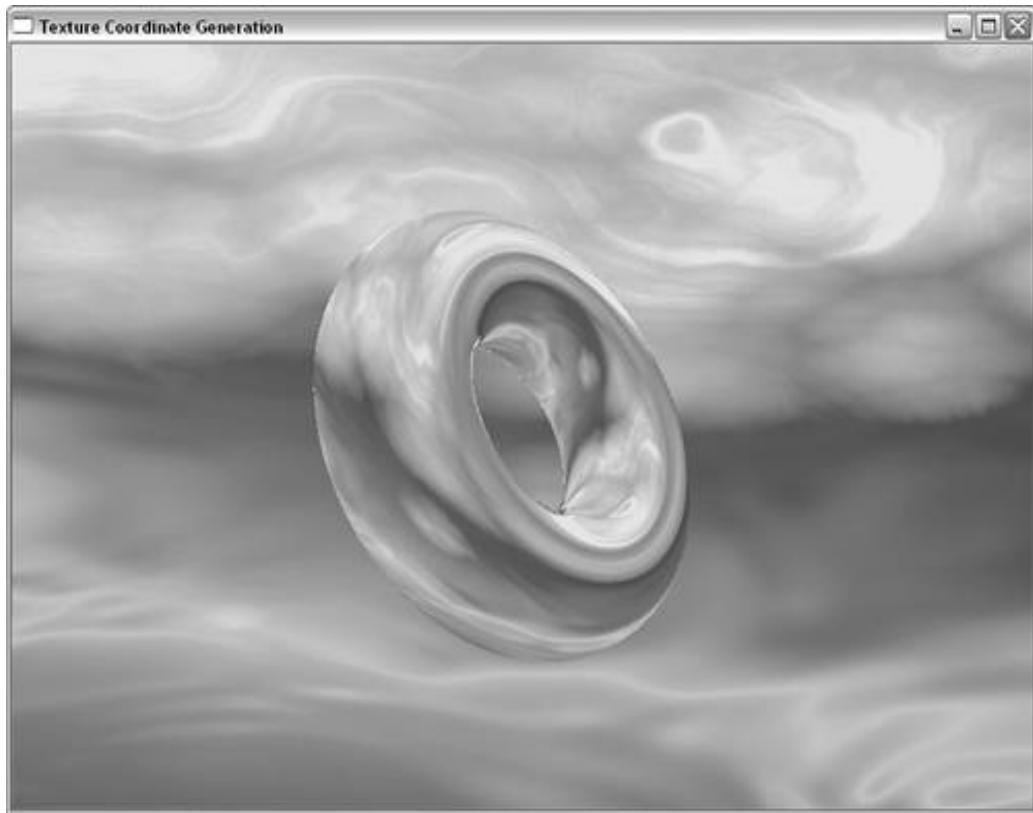
When the texture generation mode is set to `GL_SPHERE_MAP`, OpenGL calculates texture coordinates in such a way that the object appears to be reflecting the current texture map. This is the easiest mode to set up, with just these two lines from the TEXGEN sample program:

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
```

You usually can make a well-constructed texture by taking a photograph through a fish-eye lens. This texture then lends a convincing reflective quality to the geometry. For more realistic results, sphere mapping has largely been replaced by cube mapping (discussed next). However, sphere mapping still has some uses.

In particular, sphere mapping requires only a single texture instead of six, and if true reflectivity is not required, you can obtain adequate results from sphere mapping. Even without a well-formed texture taken through a fish-eye lens, you can also use sphere mapping for an approximate environment map. Many surfaces are shiny and reflect the light from their surroundings, but are not mirror-like in their reflective qualities. In the TEXGEN sample program, we use a suitable environment map for the background (all modes show this background), as well as the source for the sphere map. [Figure 9.9](#) shows the environment-mapped torus against a similarly colored background. Moving the torus around with the arrow keys produces a reasonable approximation of a reflective surface.

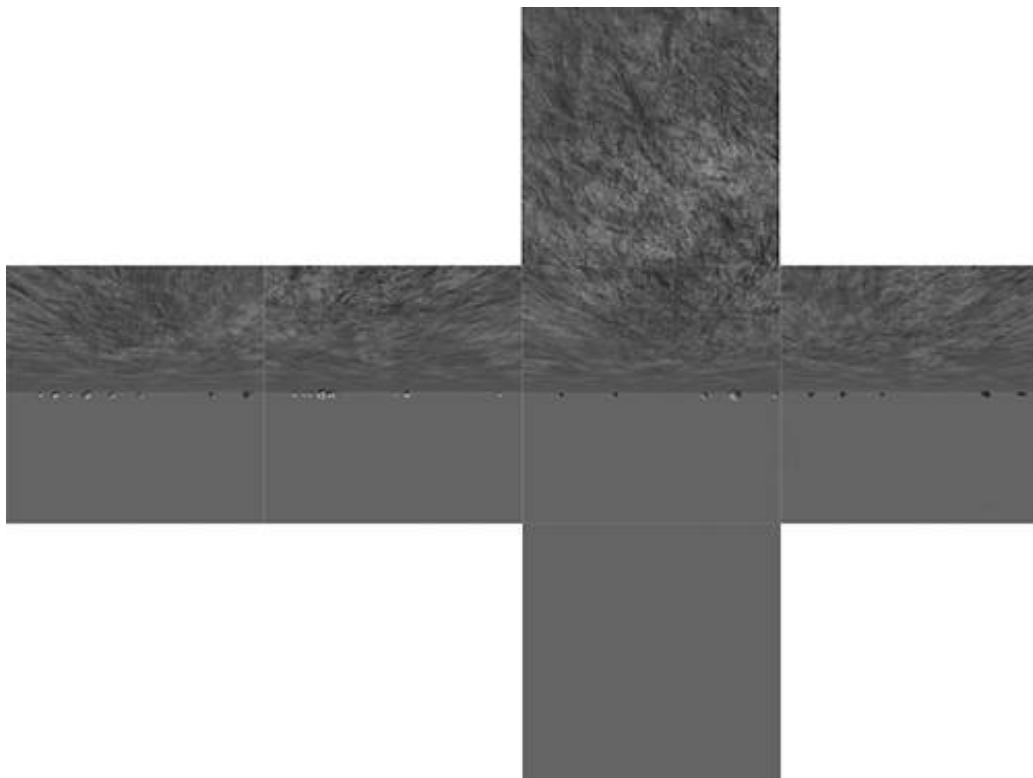
Figure 9.9. An environment map using sphere map.



Cube Mapping

The last two texture generation modes, `GL_REFLECTION_MAP` and `GL_NORMAL_MAP`, require the use of a new type of texture environment: the cube map. A cube map is not a single texture, but rather a set of six textures that make up the six sides of a cube. [Figure 9.10](#) shows the layout of six square textures comprising a cube map for the CUBEMAP sample program.

Figure 9.10. Cube map for SphereWorld in the CUBEMAP sample program.



These six tiles represent the view of SphereWorld from six different directions (forward, backward, left, right, up, and down). Using the texture generation mode `GL_REFLECTION_MAP`, you can then create a truly accurate reflective surface.

Loading Cube Maps

Cube maps add six new values that can be passed into `glTexImage2D`:

`GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`,
`GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`,
`GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, and `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`. These constants represent the direction in world coordinates of the cube face surrounding the object being mapped. For example, to load the map for the positive X direction, you might use a function that looks like this:

```
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0, GL_RGBA, iWidth, iHeight,
             0, GL_RGBA, GL_UNSIGNED_BYTE, pImage);
```

To take this example further, look at the following code segment from the CUBEMAP sample program. Here, we store the name and identifiers of the six cube maps in arrays and then use a loop to load all six images into a single texture object:

```
const char *szCubeFaces[6] = { "right.tga", "left.tga", "up.tga", "down.tga",
                               "backward.tga", "forward.tga" };
GLenum cube[6] = { GL_TEXTURE_CUBE_MAP_POSITIVE_X,
                    GL_TEXTURE_CUBE_MAP_NEGATIVE_X,
                    GL_TEXTURE_CUBE_MAP_POSITIVE_Y,
                    GL_TEXTURE_CUBE_MAP_NEGATIVE_Y,
                    GL_TEXTURE_CUBE_MAP_POSITIVE_Z,
                    GL_TEXTURE_CUBE_MAP_NEGATIVE_Z };
. . .
. . .
glBindTexture(GL_TEXTURE_CUBE_MAP, textureObjects[CUBE_MAP]);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_REPEAT);
// Load Cube Map images
for(i = 0; i < 6; i++)
{
    GLubyte *pBytes;
    GLint iWidth, iHeight, iComponents;
    GLenum eFormat;
    // Load this texture map
    // glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_GENERATE_MIPMAP, GL_TRUE);
    pBytes = gltLoadTGA(szCubeFaces[i], &iWidth, &iHeight,
                        &iComponents, &eFormat);
    glTexImage2D(cube[i], 0, iComponents, iWidth, iHeight, 0, eFormat,
                GL_UNSIGNED_BYTE, pBytes);
    free(pBytes);
}
```

Notice how the first parameter to `glBindTexture` is now `GL_TEXTURE_CUBE_MAP` instead of `GL_TEXTURE_2D`. The same value is also used in `glEnable` to enable cube mapping:

```
glEnable(GL_TEXTURE_CUBE_MAP);
```

If both `GL_TEXTURE_CUBE_MAP` and `GL_TEXTURE_2D` are enabled, `GL_TEXTURE_CUBE_MAP` has

precedence. Also, notice that the texture parameter values (set with `glTexParameter`) affect all six images in a single cube texture. For all intents and purposes, cube maps are treated like a single 3D texture map, using S, T, and R coordinates to interpolate texture coordinate values.

Using Cube Maps

The most common use of cube maps is to create an object that reflects its surroundings. The six images used for the CUBEMAP sample program were made from one of the SphereWorld samples with the torus and revolving sphere removed. The point of view was changed six times and captured (using a 90-degree field of view). These views were then assembled into a single cube map using the code from the preceding section, and appear something like that shown in [Figure 9.9](#).

The CUBEMAP sample sets the texture generation mode to `GL_REFLECTION_MAP` for all three texture coordinates:

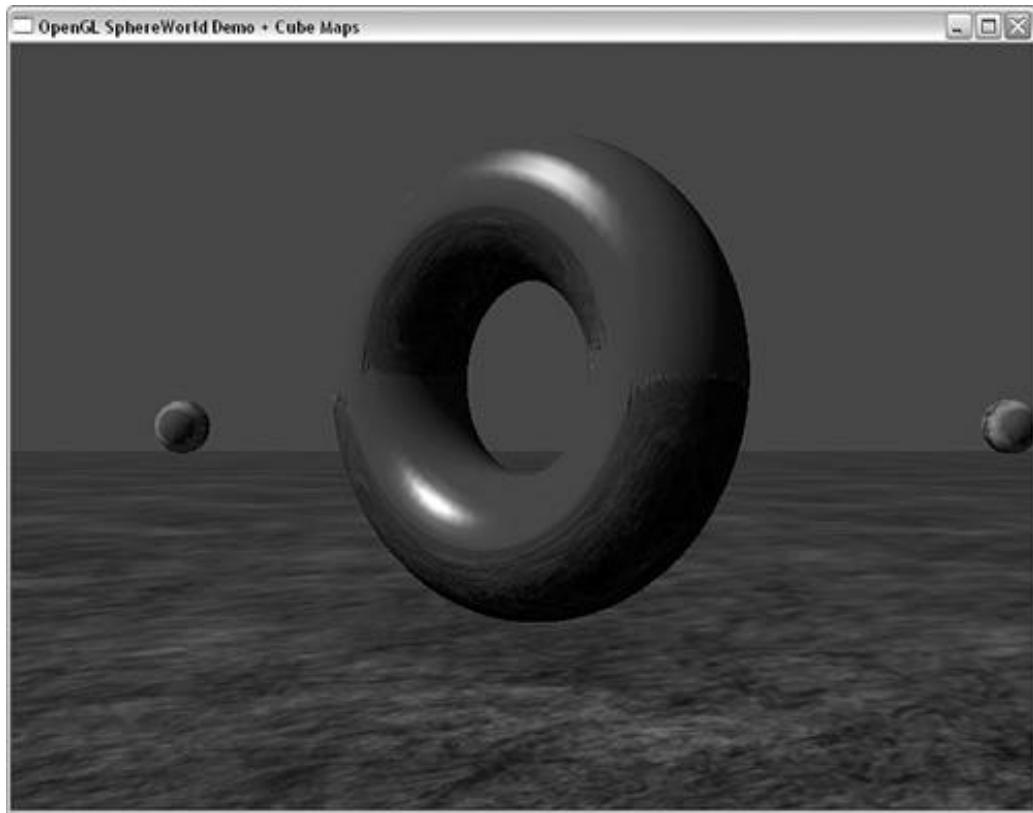
```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
```

Then, when the torus is drawn, 2D texturing is disabled and cube mapping is enabled. Texture coordinate generation is enabled and the torus is drawn. Afterward, the normal texturing state is restored:

```
glDisable(GL_TEXTURE_2D);
 glEnable(GL_TEXTURE_CUBE_MAP);
 glBindTexture(GL_TEXTURE_CUBE_MAP, textureObjects[CUBE_MAP]);
 glEnable(GL_TEXTURE_GEN_S);
 glEnable(GL_TEXTURE_GEN_T);
 glEnable(GL_TEXTURE_GEN_R);
 gltDrawTorus(0.35, 0.15, 61, 37);
 glDisable(GL_TEXTURE_GEN_S);
 glDisable(GL_TEXTURE_GEN_T);
 glDisable(GL_TEXTURE_GEN_R);
 glDisable(GL_TEXTURE_CUBE_MAP);
 glEnable(GL_TEXTURE_2D);
```

[Figure 9.11](#) shows the output of the CUBEMAP sample program. Notice how the ground is reflected correctly off the bottom surfaces of the torus, with the gray sky reflected on the upper surfaces. In the same manner, you can see the spheres scattered throughout SphereWorld reflected in the sides of the torus.

Figure 9.11. Output from the CUBEMAP sample program.



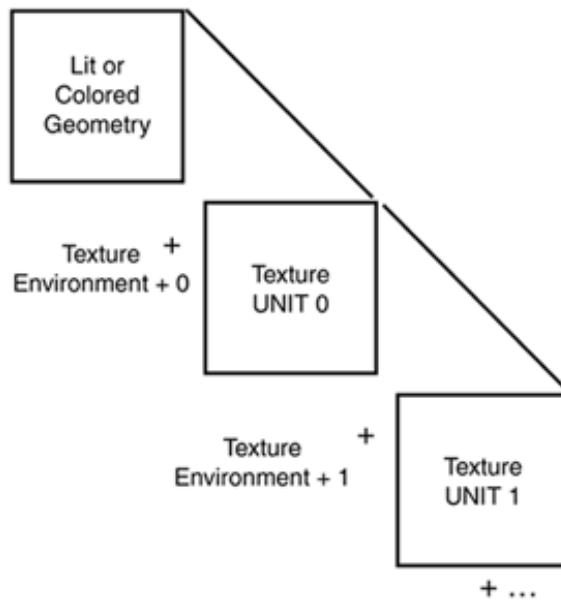
Multitexture

Most modern OpenGL hardware implementations support the ability to apply two or more textures to geometry simultaneously. All OpenGL implementations support at least a single texture being applied to geometry. If an implementation supports more than one texture unit, you can query with `GL_MAX_TEXTURE_UNITS` to see how many texture units are available:

```
GLint iUnits;  
glGetIntegerv(GL_MAX_TEXTURE_UNITS, &iUnits);
```

Textures are applied from the base texture unit (`GL_TEXTURE0`), up to the maximum number of texture units in use (`GL_TEXTUREn`, where *n* is the number of texture units in use). Each texture unit has its own texture environment that determines how fragments are combined with the previous texture unit. [Figure 9.12](#) shows three textures being applied to geometry, each with its own texture environment.

Figure 9.12. Multitexture order of operations.



By default, the first texture unit is the active texture unit. All texture commands, with the exception of `glTexCoord`, affect the currently active texture unit. You can change the current texture unit by calling `glActiveTexture` with the texture unit identifier as the argument. For example, to switch to the second texture unit and enable 2D texturing on that unit, you would call the following:

```
glActiveTexture(GL_TEXTURE1);
 glEnable(GL_TEXTURE_2D);
```

To disable texturing on the second texture unit and switch back to the first (base) texture unit, you would make these calls:

```
glDisable(GL_TEXTURE_2D);
 glActiveTexture(GL_TEXTURE0);
```

All calls to texture functions such as `glTexParameter`, `glTexEnv`, `glTexGen`, `glTexImage`, and `glBindTexture` are bound only to the current texture unit. When geometry is rendered, texture is applied from all enabled texture units using the texture environment and parameters previously specified. The only exception is texture coordinates.

Multiple Texture Coordinates

Occasionally, you might apply all active textures using the same texture coordinates for each texture, but this is rarely the case. When using multiple textures, you can still specify texture coordinates with `glTexCoord`; however, these texture coordinates are used only for the first texture unit (`GL_TEXTURE0`). To specify texture coordinates separately for each texture unit, you need a new texture coordinate function:

```
glMultiTexCoord2f(GLenum texUnit, GLfloat s, GLfloat t);
```

The `texUnit` parameter is `GL_TEXTURE0`, `GL_TEXTURE1`, and so on up to the maximum number of supported texturing units. In this version of `glMultiTexCoord`, you specify the `s` and `t` coordinates of a two-dimensional texture. Just like `glTexCoord`, many variations of `glMultiTexCoord` enable you to specify one-, two-, three-, and four-dimensional texture coordinates in a number of different data formats. You can also use texture coordinate generation on one or more texture units.

A Multitextured Example

Listing 9.2 presents the code for the sample program MULTITEXTURE. This program is similar to the CUBEMAP program, and only the changed functions are listed here. In the CUBEMAP program, we removed the wood texture from the torus and replaced it with a cube map, giving the appearance of a perfectly reflective surface. In MULTITEXTURE, we have put the wood texture back on the torus but moved the cube map to the second texture unit. We use the `GL_MODULATE` texture environment to blend the two textures together. We have also lightened the textures for the cube map to make the surface brighter and the wood easier to see. Figure 9.13 shows the output from the MULTITEXTURE program.

Listing 9.2. Source Code for the MULTITEXTURE Sample Program

```
#include "../../Common/OpenGLSB.h"      // System and OpenGL Stuff
#include "../../Common/GLTools.h"        // OpenGL toolkit
#include <math.h>
#define NUM_SPHERES      30
GLTFrame    spheres[NUM_SPHERES];
GLTFrame    frameCamera;
// Light and material Data
GLfloat fLightPos[4]   = { -100.0f, 100.0f, 50.0f, 1.0f }; // Point source
GLfloat fNoLight[] = { 0.0f, 0.0f, 0.0f, 0.0f };
GLfloat fLowLight[] = { 0.25f, 0.25f, 0.25f, 1.0f };
GLfloat fBrightLight[] = { 1.0f, 1.0f, 1.0f, 1.0f };
#define GROUND_TEXTURE  0
#define SPHERE_TEXTURE 1
#define WOOD_TEXTURE   2
#define CUBE_MAP        3
#define NUM_TEXTURES   4
GLuint  textureObjects[NUM_TEXTURES];
const char *szTextureFiles[] = {"grass.tga", "orb.tga", "wood.tga"};
const char *szCubeFaces[6] = { "right.tga", "left.tga", "up.tga",
                             "down.tga", "backward.tga", "forward.tga" };
GLenum  cube[6] = { GL_TEXTURE_CUBE_MAP_POSITIVE_X,
                    GL_TEXTURE_CUBE_MAP_NEGATIVE_X,
                    GL_TEXTURE_CUBE_MAP_POSITIVE_Y,
                    GL_TEXTURE_CUBE_MAP_NEGATIVE_Y,
                    GL_TEXTURE_CUBE_MAP_POSITIVE_Z,
                    GL_TEXTURE_CUBE_MAP_NEGATIVE_Z };

///////////////////////////////
// This function does any needed initialization on the rendering
// context.
void SetupRC()
{
    int iSphere;
    int i;
    // Grayish background
    glClearColor(fLowLight[0], fLowLight[1], fLowLight[2], fLowLight[3]);
    // Cull backs of polygons
    glCullFace(GL_BACK);
    glFrontFace(GL_CCW);
    glEnable(GL_CULL_FACE);
    glEnable(GL_DEPTH_TEST);
    // Setup light Parameters:
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, fNoLight);
    glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR);
    glLightfv(GL_LIGHT0, GL_AMBIENT, fLowLight);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, fBrightLight);
    glLightfv(GL_LIGHT0, GL_SPECULAR, fBrightLight);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
```

```

// Mostly use material tracking
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
glMateriali(GL_FRONT, GL_SHININESS, 128);
gltInitFrame(&frameCamera); // Initialize the camera
// Randomly place the sphere inhabitants
for(iSphere = 0; iSphere < NUM_SPHERES; iSphere++)
{
    gltInitFrame(&spheres[iSphere]); // Initialize the frame

    // Pick a random location between -20 and 20 at .1 increments
    spheres[iSphere].vLocation[0] = (float)((rand() % 400) - 200) * 0.1f;
    spheres[iSphere].vLocation[1] = 0.0f;
    spheres[iSphere].vLocation[2] = (float)((rand() % 400) - 200) * 0.1f;
}
// Set up texture maps
glEnable(GL_TEXTURE_2D);
glGenTextures(NUM_TEXTURES, textureObjects);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
// Load regular textures
for(i = 0; i < CUBE_MAP; i++)
{
    GLubyte *pBytes;
    GLint iWidth, iHeight, iComponents;
    GLenum eFormat;
    glBindTexture(GL_TEXTURE_2D, textureObjects[i]);
    // Load this texture map
    glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, GL_TRUE);
    pBytes = gltLoadTGA(szTextureFiles[i], &iWidth, &iHeight,
                        &iComponents, &eFormat);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_COMPRESSED_RGB, iWidth, iHeight,
                0, eFormat, GL_UNSIGNED_BYTE, pBytes);
    free(pBytes);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_LINEAR_MIPMAP_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
}

glActiveTexture(GL_TEXTURE1);
glDisable(GL_TEXTURE_2D);
glEnable(GL_TEXTURE_CUBE_MAP);
glBindTexture(GL_TEXTURE_CUBE_MAP, textureObjects[CUBE_MAP]);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_GEN_R);
glBindTexture(GL_TEXTURE_CUBE_MAP, textureObjects[CUBE_MAP]);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_REPEAT);
// Load Cube Map images
for(i = 0; i < 6; i++)
{
    GLubyte *pBytes;
    GLint iWidth, iHeight, iComponents;

```

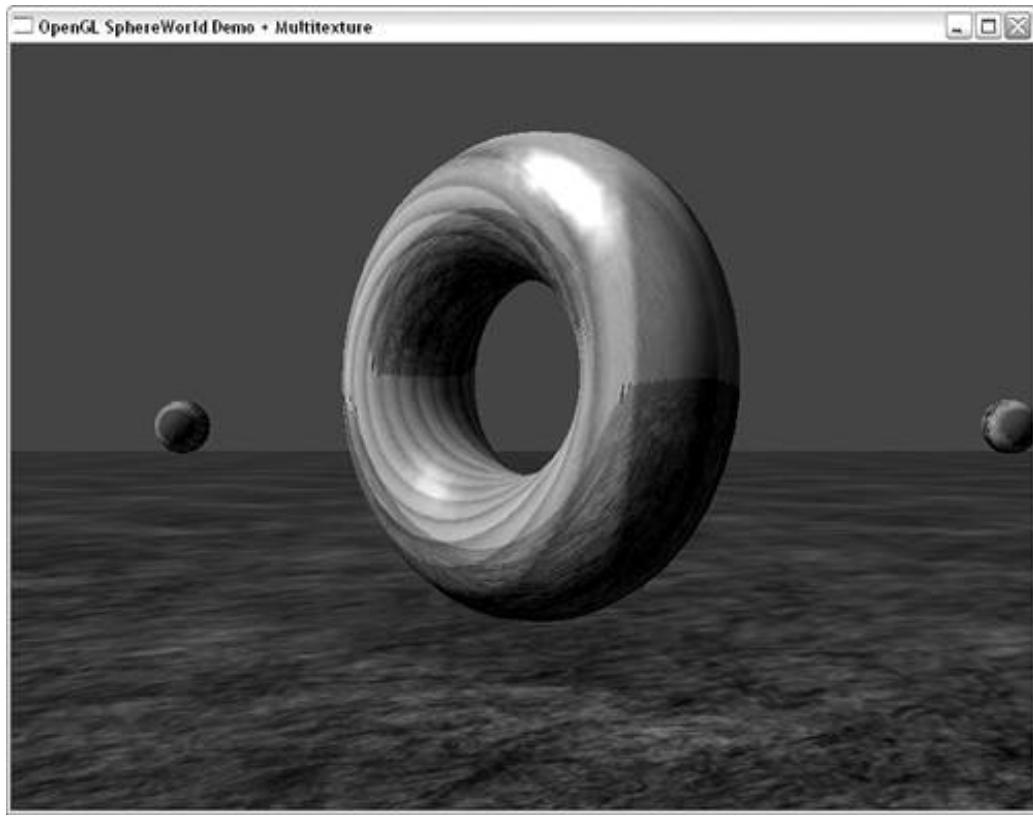
```

GLenum eFormat;
// Load this texture map
// glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_GENERATE_MIPMAP, GL_TRUE);
pBytes = gltLoadTGA(szCubeFaces[i], &iWidth, &iHeight,
                     &iComponents, &eFormat);
glTexImage2D(cube[i], 0, iComponents, iWidth, iHeight, 0,
              eFormat, GL_UNSIGNED_BYTE, pBytes);
free(pBytes);
}

///////////////////////////////
// Draw random inhabitants and the rotating torus/sphere duo
void DrawInhabitants(void)
{
    static GLfloat yRot = 0.0f;           // Rotation angle for animation
    GLint i;
    yRot += 0.5f;
    glColor4f(1.0f, 1.0f, 1.0f, 1.0f);
    // Draw the randomly located spheres
    glBindTexture(GL_TEXTURE_2D, textureObjects[SPHERE_TEXTURE]);
    for(i = 0; i < NUM_SPHERES; i++)
    {
        glPushMatrix();
        gltApplyActorTransform(&spheres[i]);
        gltDrawSphere(0.3f, 21, 11);
        glPopMatrix();
    }
    glPushMatrix();
    glTranslatef(0.0f, 0.1f, -2.5f);
    // Torus alone will be specular
    glMaterialfv(GL_FRONT, GL_SPECULAR, fBrightLight);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);
    // Bind to Wood, first texture
    glBindTexture(GL_TEXTURE_2D, textureObjects[WOOD_TEXTURE]);
    glActiveTexture(GL_TEXTURE1);
    glEnable(GL_TEXTURE_CUBE_MAP);
    glActiveTexture(GL_TEXTURE0);
    gltDrawTorus(0.35, 0.15, 41, 17);
    glActiveTexture(GL_TEXTURE1);
    glDisable(GL_TEXTURE_CUBE_MAP);
    glActiveTexture(GL_TEXTURE0);
    glMaterialfv(GL_FRONT, GL_SPECULAR, fNoLight);
    glPopMatrix();
}

```

Figure 9.13. Output from the MULTITEXTURE sample program.



Texture Combiners

In [Chapter 6](#), "More on Colors and Materials," you learned how to use the blending equation to control the way color fragments were blended together when multiple layers of geometry were drawn in the color buffer (typically back to front). OpenGL's texture combiners allow the same sort of control (only better) for the way multiple texture fragments are combined. By default, you can simply choose one of the texture environment modes ([GL_DECAL](#), [GL_REPLACE](#), [GL_MODULATE](#), or [GL_ADD](#)) for each texture unit, and the results of each texture application are then added to the next texture unit. These texture environments were covered in [Chapter 8](#).

Texture combiners add a new texture environment, [GL_COMBINE](#), that allows you to explicitly set the way texture fragments from each texture unit are combined. To use texture combiners, you call [glTexEnv](#) in the following manner:

```
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE);
```

Texture combiners are controlled entirely through the [glTexEnv](#) function. Next, you need to select which texture combiner function you want to use. The combiner function selector, which can be either [GL_COMBINE_RGB](#) or [GL_COMBINE_ALPHA](#), becomes the second argument to the [glTexEnv](#) function. The third argument becomes the texture environment function that you want to employ (for either RGB or alpha values). These functions are listed in [Table 9.4](#). For example, to select the [GL_REPLACE](#) combiner for RGB values, you would call the following function:

```
glTexEnvi(GL_TEXTURE_ENV, GL_COMBINE_RGB, GL_REPLACE);
```

This combiner does little more than duplicate the normal [GL_REPLACE](#) texture environment.

Table 9.4. Texture Combiner Functions

Constant	Function
<code>GL_REPLACE</code>	Arg0
<code>GL_MODULATE</code>	$\text{Arg0} * \text{Arg1}$
<code>GL_ADD</code>	$\text{Arg0} + \text{Arg1}$
<code>GL_ADD_SIGNED</code>	$\text{Arg0} + \text{Arg1} - 0.5$
<code>GL_INTERPOLATE</code>	$(\text{Arg0} * \text{Arg2}) + (\text{Arg1} * (1 - \text{Arg2}))$
<code>GL_SUBTRACT</code>	$\text{Arg0} - \text{Arg1}$
<code>GL_DOT3_RGB/GL_DOT3_RGBA</code>	$4 * ((\text{Arg0r} - 0.5) * (\text{Arg1r} - 0.5) + (\text{Arg0g} - 0.5) * (\text{Arg1g} - 0.5) + (\text{Arg0b} - 0.5) * (\text{Arg1b} - 0.5))$

The values of $\text{Arg0} - \text{Arg2}$ are from source and operand values set with more calls to `glTexEnv`. The values `GL_SOURCE x _RGB` and `GL_SOURCE x _ALPHA` are used to specify the RGB or alpha combiner function arguments, where x is 0, 1, or 2. The values for these sources are given in [Table 9.5](#).

Table 9.5. Texture Combiner Sources

Constant	Description
<code>GL_TEXTURE</code>	The texture bound to the current active texture unit
<code>GL_TEXTUREx</code>	The texture bound to texture unit x
<code>GL_CONSTANT</code>	The color (or alpha) value set by the texture environment variable <code>GL_TEXTURE_ENV_COLOR</code>
<code>GL_PRIMARY_COLOR</code>	The color (or alpha) value coming from the original geometry fragment
<code>GL_PREVIOUS</code>	The color (or alpha) value resulting from the previous texture unit's texture environment

For example, to select the texture from texture unit 0 for Arg0, you would make the following function call:

```
glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE0_RGB, GL_TEXTURE0);
```

You also have some additional control over what values are used from a given source for each argument. To set these operands, you use the constant `GL_OPERAND x _RGB` or `GL_OPERAND x _ALPHA`, where x is 0, 1, or 2. The valid operands and their meanings are given in [Table 9.6](#).

Table 9.6. Texture Combiner Operands

Constant	Meaning

<code>GL_SRC_COLOR</code>	The color values from the source. This may not be used with <code>GL_OPERANDx_ALPHA</code> .
<code>GL_ONE_MINUS_SRC_COLOR</code>	One's complement (1-value) of the color values from the source. This may not be used with <code>GL_OPERANDx_ALPHA</code> .
<code>GL_SRC_ALPHA</code>	The alpha values of the source.
<code>GL_ONE_MINUS_SRC_ALPHA</code>	One's complement (1-value) of the alpha values from the source.

For example, if you have two textures loaded on the first two texture units, and you want to multiply the color values from both textures during the texture application, you would set it up as follows:

```
// Tell OpenGL you want to use texture combiners
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE);
// Tell OpenGL which combiner you want to use (GL_MODULATE for RGB values)
glTexEnvi(GL_TEXTURE_ENV, GL_COMBINE_RGB, GL_MODULATE);
// Tell OpenGL to use texture unit 0's color values for Arg0
glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE0_RGB, GL_TEXTURE0);
glTexEnvi(GL_TEXTURE_ENV, GL_OPERAND0_RGB, GL_SRC_COLOR);
// Tell OpenGL to use texture unit 1's color values for Arg1
glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE0_RGB, GL_TEXTURE1);
glTexenvi(GL_TEXTURE_ENV, GL_OPERAND0_RGB, GL_SRC_COLOR);
```

Finally, with texture combiners, you can also specify a constant RGB or alpha scaling factor. The default parameters for these are as follows:

```
glTexEnvf(GL_TEXTURE_ENV, GL_RGB_SCALE, 1.0f);
glTexEnvf(GL_TEXTURE_ENV, GL_ALPHA_SCALE, 1.0f);
```

Summary

In this chapter, we took texture mapping beyond the simple basics of applying a texture to geometry. You saw how to get improved filtering, obtain better performance and memory efficiency through texture compression, and generate automatic texture coordinates for geometry. You also saw how to add plausible environment maps with sphere mapping and more realistic and correct reflections using cube maps.

Finally, we discussed multitexture and texture combiners. The ability to apply more than one texture at a time is the foundation for many special effects, including hardware support for bump mapping. Using texture combiners, you have a great deal of flexibility in specifying how up to three textures are combined. While fragment programs exposed through the new OpenGL shading language do give you ultimate control over texture application, you can quickly and easily take advantage of the capabilities described in this chapter.

Reference

glActiveTexture

Purpose: Sets the active texture unit.

Include File: `<gl.h>`

Syntax:

```
void glActiveTexture(GLenum texUnit);
```

Description: This function sets the active texture unit. The active texture unit is the target of all texture functions with the exception of `glTexCoord` and `glTexCoordPointer`.

Parameters:

`texUnit` `GLenum`: The texture unit to be made current. It may be `GL_TEXTURE0` through `GL_TEXTUREn`, where `n` is the number of supported texture units.

Returns: None.

See Also: `glClientActiveTexture`

glClientActiveTexture

Purpose: Sets the active texture unit for vertex arrays.

Include File: `<gl.h>`

Syntax:

```
void glClientActiveTexture(GLenum texUnit);
```

Description: This function sets the active texture unit for vertex array texture coordinate specification. The `glTexCoordPointer` vertex array function (see [Chapter 11](#)), by default, specifies a pointer for the first texture unit (`GL_TEXTURE0`). You use this function to change the texture unit that is the target of `glTexCoordPointer`. Calling this function multiple times allows you to specify multiple arrays of texture coordinates for multitextured vertex array operations.

Parameters:

`texUnit` `GLenum`: The texture unit to be made current. It may be `GL_TEXTURE0` through `GL_TEXTUREn`, where `n` is the number of supported texture units.

Returns: None.

See Also: `glActiveTexture`

glCompressedTexImage

Purpose: Loads a compressed texture image.

Include File: `<gl.h>`

Variations:

```

void glCompressedTexImage1D(GLenum target, GLint
  ↪ level, GLenum internalFormat,
    GLsizei width,
    GLint border, GLsizei
  ↪ imageSize, void *data);
void glCompressedTexImage2D(GLenum target, GLint
  ↪ level, GLenum internalFormat,
    GLsizei width, GLsizei height,
    GLint border, GLsizei
  ↪ imageSize, void *data);
void glCompressedTexImage3D(GLenum target, GLint
  ↪ level, GLenum internalFormat,
    GLsizei width, GLsizei
  ↪ height, GLsizei depth,
    GLint border, GLsizei
  ↪ imageSize, GLvoid *data);

```

Description: This function enables you to load a compressed texture image. This function is nearly identical in use to the `glTexImage` family of functions, with the exception that the `internalFormat` parameter must specify a supported compressed texture format.

Parameters:

`target` `GLenum`: It must be `GL_TEXTURE_1D` for `glCompressedTexImage1D`, `GL_TEXTURE_2D` for `glCompressedTexImage2D`, or `GL_TEXTURE_3D` for `glCompressedTexImage3D`. For 2D cube maps only, it may also be `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_CUBE_MAP_POSITIVE_Y`, `GL_CUBE_MAP_NEGATIVE_Y`, `GL_CUBE_MAP_POSITIVE_Z`, or `GL_CUBE_MAP_NEGATIVE_Z`.

`level` `GLint`: The level of detail. It is usually 0 unless mipmapping is used.

`internalFormat` `GLint`: The internal format of the compressed data. Specific values for texture compression formats will vary from implementation to implementation. [Table 9.3](#) lists the formats for one commonly supported compression format on the PC and Macintosh.

`width` `GLsizei`: The width of the one-, two-, or three-dimensional texture image. It must be a power of 2 but may include a border.

`height` `GLsizei`: The height of the two- or three-dimensional texture image. It must be a power of 2 but may include a border.

`depth` `GLsizei`: The depth of a three-dimensional texture image. It must be a power of 2 but may include a border.

`border` `GLint`: The width of the border. Not all texture compression formats support borders.

`imageSize` `GLsizei`: The size in bytes of the compressed data.

`data` `GLvoid *`: The compressed texture data.

Returns: None.

See Also: `glCompressedTexSubImage`, `glTexImage`

glCompressedTexSubImage

Purpose: Replaces a portion of an existing compressed texture map.

Include File: `<gl.h>`

Variations:

```
void glCompressedTexSubImage1D(GLenum target,
    ➔ GLint level,
                    GLint xOffset,
                    GLsizei width,
                    GLsizei imageSize
    ➔ , const void *data);
void glCompressedTexSubImage2D(GLenum target,
    ➔ GLint level,
                    GLint xOffset,
    ➔ GLint yOffset,
                    GLsizei width,
    ➔ GLsizei height,
                    GLsizei imageSize
    ➔ , const void *data);
void glCompressedTexSubImage3D(GLenum target,
    ➔ GLint level,
                    GLint xOffset,
    ➔ GLint yOffset, GLint zOffset,
                    GLsizei width,
    ➔ GLsizei height, GLsizei depth,
                    GLsizei imageSize
    ➔ , const void *data);
```

Description: This function replaces a portion of an existing one-, two-, or three-dimensional compressed texture map. Updating all or part of an existing texture map may be considerably faster than reloading a texture image with `glCompressedTexImage`. You cannot perform an initial texture load with this function; it is used only to update an existing texture.

Parameters:

<code>target</code>	<code>GLenum</code> : It must be <code>GL_TEXTURE_1D</code> , <code>GL_TEXTURE_2D</code> , or <code>GL_TEXTURE_3D</code> .
<code>level</code>	<code>GLint</code> : Mipmap level to be updated.
<code>xOffset</code>	<code>GLint</code> : Offset within the existing one-, two-, or three-dimensional texture in the x direction to begin the update.
<code>yOffset</code>	<code>GLint</code> : Offset within the existing two- or three-dimensional texture in the y direction to begin the update.
<code>zOffset</code>	<code>GLint</code> : Offset within the existing three-dimensional texture in the z direction to begin the update.
<code>width</code>	<code>GLsizei</code> : The width of the one-, two-, or three-dimensional texture data being updated.
<code>height</code>	<code>GLsizei</code> : The height of the two- or three-dimensional texture data being updated.

depth GLsizei: The depth of the three-dimensional texture data being updated.

imageSize GLsizei: The size in bytes of the compressed texture data.

data const void*: A pointer to the compressed texture data that is being used to update the texture target.

Returns: None.

See Also: [glCompressedTexImage](#), [glTexSubImage](#)

glGetCompressedTexImage

Purpose: Returns a compressed texture image.

Include File: `<gl.h>`

Syntax:

```
void glGetCompressedTexImage(GLenum target, GLint
→ level, void *pixels);
```

Description: This function enables you to fill a data buffer with the data that comprises the current compressed texture. This function performs the reverse operation of [glCompressedTexImage](#), which loads a compressed texture with a supplied data buffer. You cannot use this function to retrieve a compressed version of an uncompressed texture.

Parameters:

target GLenum: The texture target to be copied. It must be `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D`, `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, or `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`.

level GLint: The mipmap level to be read.

pixels void*: A pointer to a memory buffer to accept the compressed texture image.

Returns: None.

See Also: [glCompressedTexImage](#), [glCompressedTexSubImage](#), [glGetTexImage](#)

glMultiTexCoord

Purpose: Specifies the current texture image coordinate for the specified texture unit.

Include File: `<gl.h>`

Variations:

```

void glMultiTexCoord1d(GLenum texUnit, GLdouble s);
void glMultiTexCoord1dv(GLenum texUnit, GLdouble *v);
void glMultiTexCoord1i(GLenum texUnit, GLint s);
void glMultiTexCoord1liv(GLenum texUnit, GLint *v);
void glMultiTexCoord1s(GLenum texUnit, GLshort s);
void glMultiTexCoord1sv(GLenum texUnit, GLshort *v);
void glMultiTexCoord1f(GLenum texUnit, GLfloat s);
void glMultiTexCoord1fv(GLenum texUnit, GLfloat *v);
void glMultiTexCoord2i(GLenum texUnit, GLint s,
➥ GLint t);
void glMultiTexCoord2iv(GLenum texUnit, GLint *v);
void glMultiTexCoord2s(GLenum texUnit, GLshort s,
➥ GLshort t);
void glMultiTexCoord2sv(GLenum texUnit, GLshort *v);
void glMultiTexCoord2f(GLenum texUnit, GLfloat s,
➥ GLfloat t);
void glMultiTexCoord2fv(GLenum texUnit, GLfloat *v);
void glMultiTexCoord2d(GLenum texUnit, GLdouble s,
➥ GLdouble t);
void glMultiTexCoord2dv(GLenum texUnit, GLdouble *v);
void glMultiTexCoord3s(GLenum texUnit, GLshort s,
➥ GLshort t, GLshort r);
void glMultiTexCoord3sv(GLenum texUnit, GLshort *v);
void glMultiTexCoord3i(GLenum texUnit, GLint s,
➥ GLint t, GLint r);
void glMultiTexCoord3iv(GLenum texUnit, GLint *v);
void glMultiTexCoord3f(GLenum texUnit, GLfloat s,
➥ GLfloat t, GLfloat r);
void glMultiTexCoord3fv(GLenum texUnit, GLfloat *v);
void glMultiTexCoord3d(GLenum texUnit, GLdouble s,
➥ GLdouble t, GLdouble r);
void glMultiTexCoord3dv(GLenum texUnit, GLdouble *v);
void glMultiTexCoord4f(GLenum texUnit, GLfloat s,
➥ GLfloat t, GLfloat r,
➥ GLfloat q);
void glMultiTexCoord4fv(GLenum texUnit, GLfloat *v);
void glMultiTexCoord4d(GLenum texUnit, GLdouble s,
➥ GLdouble t, GLdouble r,
➥ GLdouble q);
void glMultiTexCoord4dv(GLenum texUnit, GLdouble *v);
void glMultiTexCoord4i(GLenum texUnit, GLint s,
➥ GLint t, GLint r, GLint q);
void glMultiTexCoord4iv(GLenum texUnit, GLint *v);
void glMultiTexCoord4s(GLenum texUnit, GLshort s,
➥ GLshort t, GLshort r,
➥ GLshort q);
void glMultiTexCoord4sv(GLenum texUnit, GLshort *v);

```

Description: These functions set the current texture image coordinate for a specific texture unit in one to four dimensions. Texture coordinates can be updated anytime between `glBegin` and `glEnd`, and correspond to the following `glVertex` call. The texture `q` coordinate is used to scale the `s`, `t`, and `r` coordinate values and, by default, is 1.0. Any valid matrix operation can be performed on texture coordinates by specifying `GL_TEXTURE` as the target of `glMatrixMode`.

Parameters:

`texUnit` `GLenum`: The texture unit this texture coordinate applies to. It may be any value from `GL_TEXTURE0` to `GL_TEXTUREn`, where `n` is the number of supported texture units.

`s` `GLdouble` or `GLfloat` or `GLint` or `GLshort`: The horizontal texture image coordinate.

`t` `GLdouble` or `GLfloat` or `GLint` or `GLshort`: The vertical texture image coordinate.

`r` `GLdouble` or `GLfloat` or `GLint` or `GLshort`: The texture image depth coordinate.

`q` `GLdouble` or `GLfloat` or `GLint` or `GLshort`: The texture image scaling value coordinate.

`v` `GLdouble*` or `GLfloat*` or `GLint*` or `GLshort*`:

Returns: None.

See Also: `glTexGen`, `glTexImage`, `glTexParameter`, `glTexCoord`

glSecondaryColor

Purpose: Specifies a secondary color.

Include File: `<gl.h>`

Variations:

```
void glSecondaryColor3b(GLbyte red, GLbyte green,
  GLbyte blue);
void glSecondaryColor3s(GLshort red, GLshort green
  , GLshort blue);
void glSecondaryColor3i(GLint red, GLint green,
  GLint blue);
void glSecondaryColor3f(GLclampf red, GLclampf
  green, GLclampf blue);
void glSecondaryColor3d(GLclampd red, GLclampd
  green, GLclampd blue);
void glSecondaryColor3ub(GLubyte red, GLubyte
  green, GLubyte blue);
void glSecondaryColor3us(GLushort red, GLushort
  green, GLushort blue);
void glSecondaryColor3ui(GLuint red, GLuint green,
  GLuint blue);
void glSecondaryColor3bv(GLbyte* values);
void glSecondaryColor3sv(GLshort* values);
void glSecondaryColor3iv(GLint* values);
void glSecondaryColor3fv(GLclampf* values);
void glSecondaryColor3dv(GLclampd* values);
```

```
void glSecondaryColor3ubv(GLubyte* values);
void glSecondaryColor3usv(GLushort* values);
void glSecondaryColor3uiv(GLuint* values);
```

Description: When objects are texture mapped, specular highlights are typically muted by the application of texture after lighting. You can compensate for this effect by setting the `GL_LIGHT_MODEL_COLOR_CONTROL` light model parameter, which uses the specular lighting and material values for a secondary color added after the texture. This function allows you to specify a separate secondary color value that is added to fragments when lighting is not enabled—for example, if you were doing your own lighting calculations. Calls to `glSecondaryColor` have no effect when lighting is enabled. The color sum operation must be enabled with `glEnable (GL_COLOR_SUM)`.

Parameters:

`red` Specifies the red component of the color.
`green` Specifies the green component of the color.
`blue` Specifies the blue component of the color.
`values` Specifies a pointer to the color component values.

Returns: None.

See Also: `glColor`

glTexGen

Purpose: Defines parameters for texture coordinate generation.

Include File: `<gl.h>`

Syntax:

```
void glTexGend(GLenum coord, GLenum pname,
  GLdouble param);
void glTexGenf(GLenum coord, GLenum pname, GLfloat
  param);
void glTexGeni(GLenum coord, GLenum pname, GLint
  param);
void glTexGendv(GLenum coord, GLenum pname,
  GLdouble *param);
void glTexGenfv(GLenum coord, GLenum pname,
  GLfloat *param);
void glTexGeniv(GLenum coord, GLenum pname, GLint
  *param);
```

Description: This function sets parameters for texture coordinate generation when one or more of `GL_TEXTURE_GEN_S`, `GL_TEXTURE_GEN_T`, `GL_TEXTURE_GEN_R`, or `GL_TEXTURE_GEN_Q` is enabled with `glEnable`. When `GL_TEXTURE_GEN_MODE` is set to `GL_OBJECT_LINEAR`, texture coordinates are generated by multiplying the current object (vertex) coordinates by the constant vector specified by `GL_OBJECT_PLANE`:

```
coordinate = v[0] * p[0] + v[1] * p[1] + v[2] *
    ↪ p[2] + v[3] * p[3]
```

For `GL_EYE_LINEAR`, the eye coordinates (object coordinates multiplied through the `GL_MODELVIEW` matrix) are used. When `GL_TEXTURE_GEN_MODE` is set to `GL_SPHERE_MAP`, coordinates are generated in a sphere about the current viewing position or origin. When `GL_TEXTURE_REFLECTION_MAP` is specified, texture coordinates are calculated as a reflection from the current cube map. Finally, `GL_TEXTURE_NORMAL_MAP` also works with cube maps but transforms the object's normal to eye coordinates (instead of vertex) and uses them for texture coordinates.

Parameters:

coord `GLenum`: The texture coordinate to map. It must be one of `GL_S`, `GL_T`, `GL_R`, or `GL_Q`.

pname `GLenum`: The parameter to set. It must be one of `GL_TEXTURE_GEN_MODE`, `GL_OBJECT_PLANE`, or `GL_EYE_PLANE`.

param The parameter value. For `GL_TEXTURE_GEN_MODE`, *param* is one of the following:

`GL_OBJECT_LINEAR`: Texture coordinates are calculated from object (vertex) coordinates.

`GL_EYE_LINEAR`: Texture coordinates are calculated by eye coordinates (object coordinates multiplied through the `GL_MODELVIEW` matrix).

`GL_SPHERE_MAP`: Texture coordinates are generated in a sphere around the viewing position.

`GL_REFLECTION_MAP`: Texture coordinates are generated using a cube map and are reflected from vertices.

`GL_NORMAL_MAP`: Texture coordinates are generated using a cube map and are based on an objects normals transformed into eye coordinates.

For `GL_OBJECT_PLANE` and `GL_EYE_PLANE`, *param* is a four-element array used as a multiplier for object or eye coordinates.

Returns: None.

See Also: `glTexCoord`, `glTexEnv`, `glTexImage1D`, `glTexImage2D`, `glTexParameter`

Chapter 10. Curves and Surfaces

by Richard S. Wright, Jr.

WHAT YOU'LL LEARN IN THIS CHAPTER:

How To	Functions You'll Use
Draw spheres, cylinders, and disks	<code>gluSphere/gluCylinder/gluDisk</code>
Use maps to render Bézier curves and surfaces	<code>glMap/glEvalCoord</code>
Use evaluators to simplify surface mapping	<code>glMapGrid/glEvalMesh</code>
Create NURBS surfaces	<code>gluNewNurbsRenderer/gluBeginSurface/ gluNurbsSurface/ gluEndSurface/ gluDeleteNurbsRendererf10</code>
Create trimming curves	<code>gluBeginTrim/gluPwlCurve/gluEndTrim</code>
Tessellate concave and convex polygons	<code>gluTessBeginPolygon/gluTessEndPolygon</code>

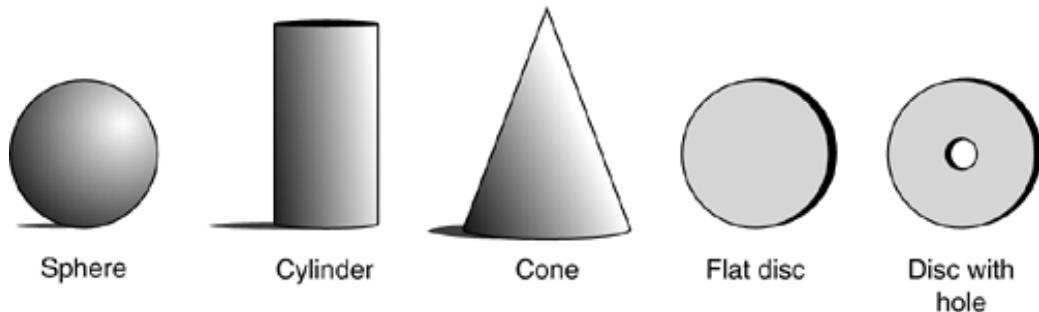
The practice of 3D graphics is little more than a computerized version of connect-the-dots. Vertices are laid out in 3D space and connected by flat primitives. Smooth curves and surfaces are approximated using flat polygons and shading tricks. The more polygons used, usually the more smooth and curved a surface may appear. OpenGL, of course, supports smooth curves and surfaces implicitly because you can specify as many vertices as you want and set any desired or calculated values for normals and color values.

OpenGL does provide some additional support, however, that makes the task of constructing more complex surfaces a bit easier. The easiest to use are some GLU functions that render spheres, cylinders, cones (special types of cylinders, as you will see), and flat, round disks, optionally with holes in them. OpenGL also provides top-notch support for complex surfaces that may be difficult to model with a simple mathematical equation: Bézier and NURB curves and surfaces. Finally, OpenGL can take large, irregular, and concave polygons and break them up into smaller, more manageable pieces.

Built-in Surfaces

The OpenGL Utility Library (GLU) that accompanies OpenGL contains a number of functions that render three quadratic surfaces. These quadric functions render spheres, cylinders, and disks. You can specify the radius of both ends of a cylinder. Setting one end's radius to 0 produces a cone. Disks, likewise, provide enough flexibility for you to specify a hole in the center (producing a washer-like surface). You can see these basic shapes illustrated graphically in [Figure 10.1](#).

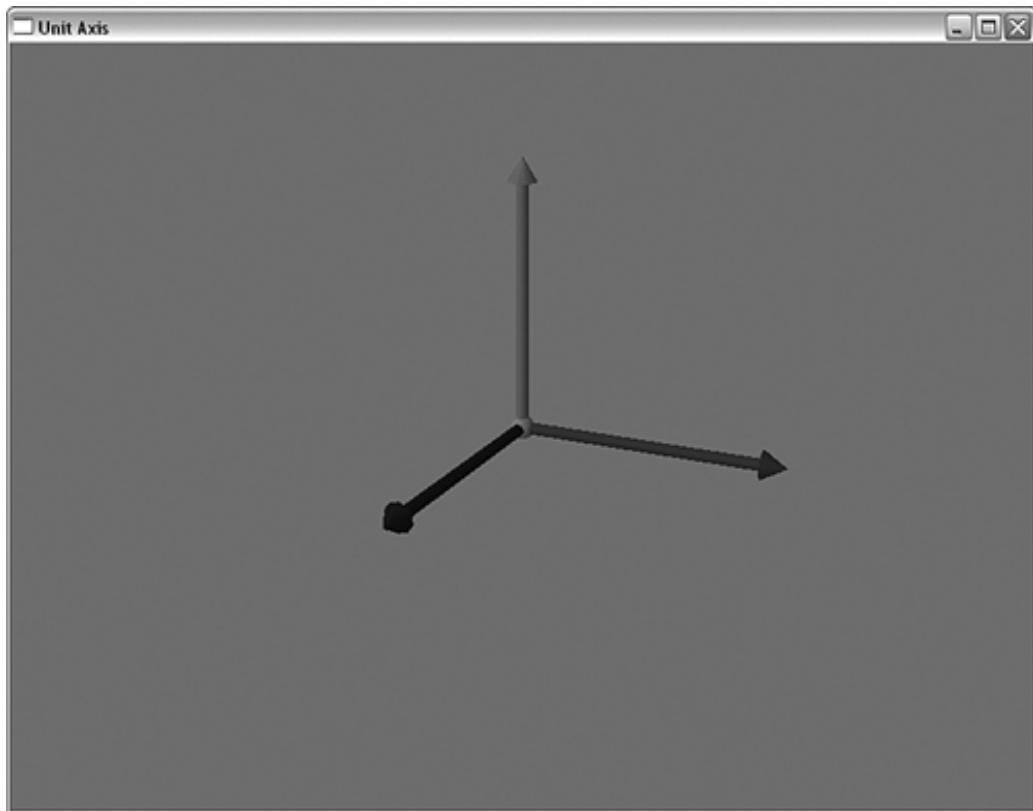
Figure 10.1. Possible quadric shapes.



These quadric objects can be arranged to create more complex models. For example, you could create a 3D molecular modeling program using just spheres and cylinders. [Figure 10.2](#) shows the 3D unit axes drawn with a sphere, three cylinders, three cones, and three disks. This model can be included into any of your own programs using the following `glTools` function:

```
void gltDrawUnitAxes(void)
```

Figure 10.2. The x,y,z-axes drawn with quadrics.



Setting Quadric States

The quadric surfaces can be drawn with some flexibility as to whether normals, texture coordinates, and so on are specified. Putting all these options into parameters to a sphere drawing function, for example, would create a function with an exceedingly long list of parameters that must be specified each time. Instead, the quadric functions use an object-oriented model. Essentially, you create a quadric object and set its rendering state with one or more state setting functions. Then you specify this object when drawing one of the surfaces, and its state determines how the surface is rendered. The following code segment shows how to create an empty quadric object and later delete it:

```
GLUquadricObj *pObj;
```

```

// . . .
pObj = gluNewQuadric(); // Create and initialize Quadric
// Set Quadric rendering Parameters
// Draw Quadric surfaces
// . . .
gluDeleteQuadric(pObj); // Free Quadric object

```

Note that you create a pointer to the `GLUQuadricObj` data type, not an instance of the data structure itself. The reason is that the `gluNewQuadric` function not only allocates space for it, but also initializes the structure members to reasonable default values.

There are four functions that you can use to modify the drawing state of the `GLUQuadricObj` object and, correspondingly, to any surfaces drawn with it. The first function sets the quadric draw style:

```
void gluQuadricDrawStyle(GLUquadricObj *obj, GLenum drawStyle);
```

The first parameter is the pointer to the quadric object to be set, and the `drawStyle` parameter is one of the values in [Table 10.1](#).

Table 10.1. Quadric Draw Styles

Constant	Description:
<code>GLU_FILL</code>	Quadric objects are drawn as solid objects.
<code>GLU_LINE</code>	Quadric objects are drawn as wireframe objects.
<code>GLU_POINT</code>	Quadric objects are drawn as a set of vertex points.
<code>GLU_SILHOUETTE</code>	This is similar to a wireframe, except adjoining faces of polygons are not drawn.

The next function specifies whether the quadric surface geometry would be generated with surface normals:

```
void gluQuadricNormals(GLUquadricObj *pbj, GLenum normals);
```

Quadrics may be drawn without normals (`GLU_NONE`), with smooth normals (`GLU_SMOOTH`), or flat normals (`GLU_FLAT`). The primary difference between smooth and flat normals is that for smooth normals, one normal is specified for each vertex of the surface, giving a smoothed-out appearance. For flat normals, one normal is used for all the vertices of any given facet (triangle) of the surface.

You can also specify whether the normals point out of the surface or inward. For example, looking at a lit sphere, you would want normals pointing outward from the surface of the sphere. However, if you were drawing the inside of a sphere--say, as part of a vaulted ceiling--you would want the normals and lighting to be applied to the inside of the sphere. The following function sets this parameter:

```
void gluQuadricOrientation(GLUquadricObj *obj, GLenum orientation);
```

Here, `orientation` can be either `GLU_OUTSIDE` or `GLU_INSIDE`. By default, quadric surfaces are wound counterclockwise, with the front faces facing the outsides of the surfaces. The outside of the surface is intuitive for spheres and cylinders; for disks, it is simply the side facing the positive

z-axis.

Finally, you can request that texture coordinates be generated for quadric surfaces with the following function:

```
void gluQuadricTexture(GLUquadricObj *obj, GLenum textureCoords);
```

Here, the *textureCoords* parameter can be either `GL_TRUE` or `GL_FALSE`. When texture coordinates are generated for quadric surfaces, they are wrapped around spheres and cylinders evenly; they are applied to disks using the center of the texture for the center of the disk, and the edges of the texture lining up with the edges of the disk.

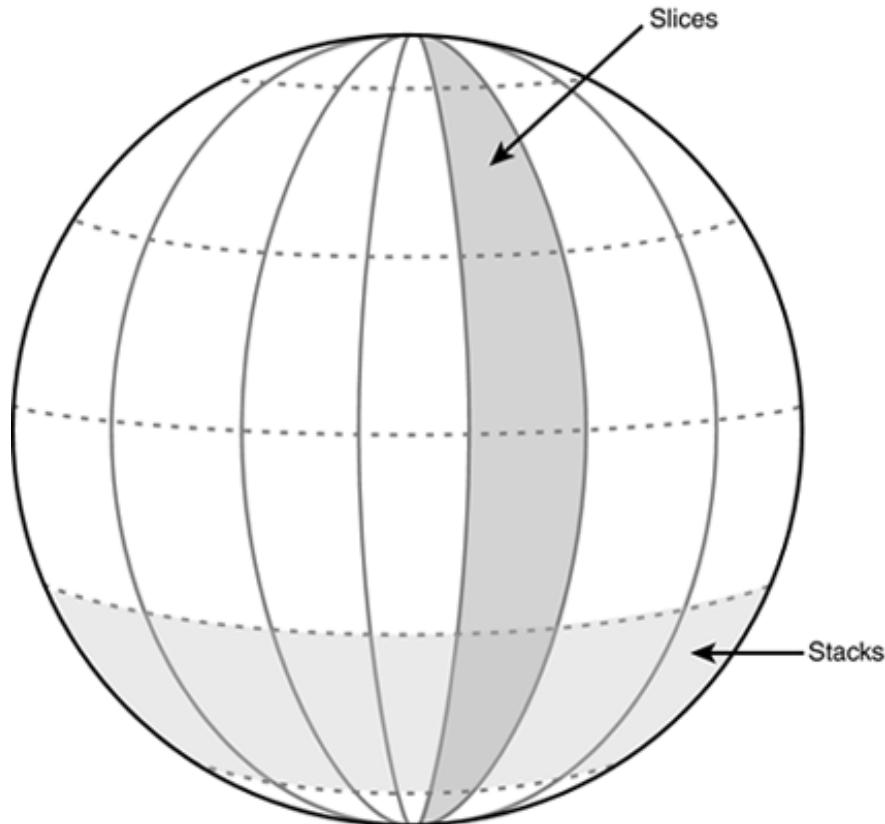
Drawing Quadrics

After the quadric object state has been set satisfactorily, each surface is drawn with a single function call. For example, to draw a sphere, you simply call the following function:

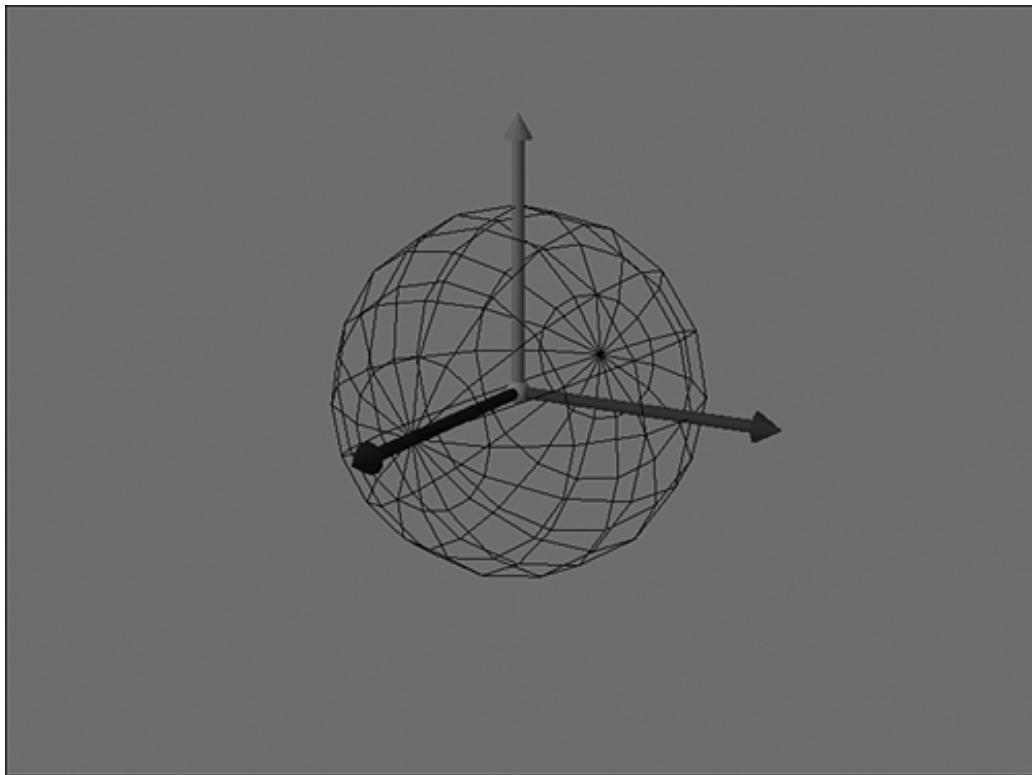
```
void gluSphere(GLUquadricObj *obj, GLdouble radius, GLint slices, GLint stacks);
```

The first parameter, *obj*, is just the pointer to the quadric object that was previously set up for the desired rendering state. The *radius* parameter is then the radius of the sphere, followed by the number of *slices* and *stacks*. Spheres are drawn with rings of triangle strips (or quad strips, depending on whose GLU library you're using) stacked from the bottom to the top, as shown in [Figure 10.3](#). The number of slices specifies how many triangle sets (or quads) are used to go all the way around the sphere. You could also think of this as the number of lines of latitude and longitude around a globe.

Figure 10.3. A quadric sphere's stacks and slices.



The quadric spheres are drawn on their sides with the positive z-axis pointing out the top of the spheres. [Figure 10.4](#) shows a wireframe quadric sphere drawn around the unit axes.

Figure 10.4. A quadric sphere's orientation.

Cylinders are also drawn along the positive z-axis and are composed of a number of stacked strips. The following function, which is similar to the `gluSphere` function, draws a cylinder:

```
void gluCylinder(GLUquadricObj *obj, GLdouble baseRadius,
                  GLdouble topRadius, GLdouble height,
                  GLint slices, GLint stacks);
```

With this function, you can specify both the base radius (near the origin) and the top radius (out along positive z-axis). The `height` parameter is simply the length of the cylinder. The orientation of the cylinder is shown in [Figure 10.5](#). [Figure 10.6](#) shows the same cylinder, but with the `topRadius` parameter set to 0, making a cone.

Figure 10.5. A quadric cylinder's orientation.



Figure 10.6. A quadric cone made from a cylinder.



The final quadric surface is the disk. Disks are drawn with loops of quads or triangle strips, divided into some number of slices. You use the following function to render a disk:

```
void gluDisk(GLUquadricObj *obj, GLdouble innerRadius,  
            GLdouble outerRadius, GLint slices, GLint loops);
```

To draw a disk, you specify both an inner radius and an outer radius. If the inner radius is 0, you

get a solid disk like the one shown in [Figure 10.7](#). A nonzero radius gives you a disk with a hole in it, as shown in [Figure 10.8](#). The disk is drawn in the xy plane.

Figure 10.7. A quadric disk showing loops and slices.

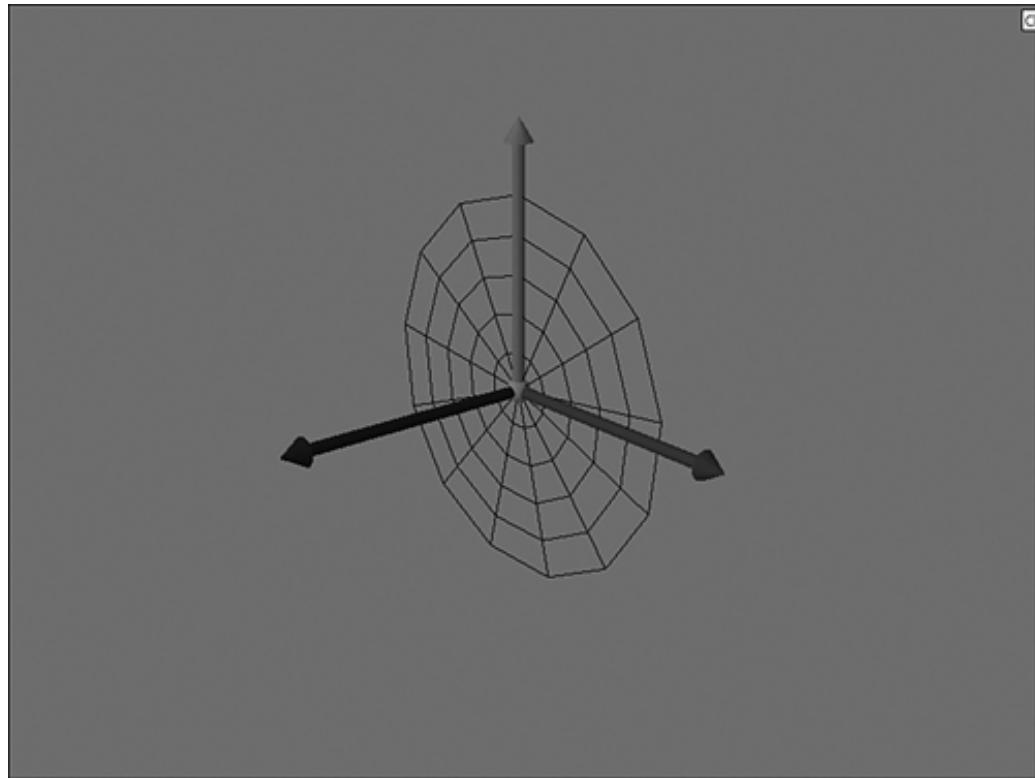


Figure 10.8. A quadric disk with a hole in the center.



In the sample program SNOWMAN, all the quadric objects are used to piece together a crude model of a snowman. White spheres make up the three sections of the body. Two small black spheres make up the eyes, and an orange cone is drawn for the carrot nose. A cylinder is used for the body of a black top hat, and two disks, one closed and one open, provide the top and the rim of the hat. The output from SNOWMAN is shown in [Figure 10.9](#). [Listing 10.1](#) shows the rendering code that draws the snowman by simply transforming the various quadric surfaces into their respective positions.

Listing 10.1. Rendering Code for the SNOWMAN Example

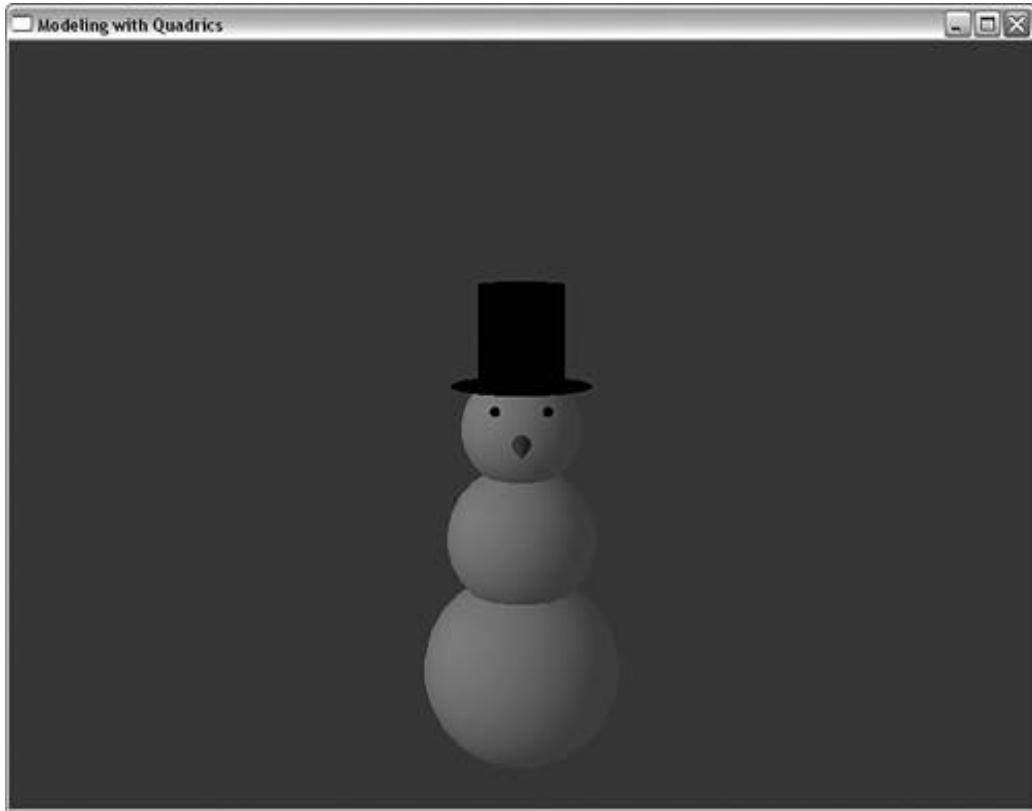
```

void RenderScene(void)
{
    GLUquadricObj *pObj;      // Quadric Object
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // Save the matrix state and do the rotations
    glPushMatrix();
        // Move object back and do in place rotation
        glTranslatef(0.0f, -1.0f, -5.0f);
        glRotatef(xRot, 1.0f, 0.0f, 0.0f);
        glRotatef(yRot, 0.0f, 1.0f, 0.0f);
        // Draw something
        pObj = gluNewQuadric();
        gluQuadricNormals(pObj, GLU_SMOOTH);
        // Main Body
        glPushMatrix();
            glColor3f(1.0f, 1.0f, 1.0f);
            gluSphere(pObj, .40f, 26, 13); // Bottom
            glTranslatef(0.0f, .550f, 0.0f); // Mid section
            gluSphere(pObj, .3f, 26, 13);

            glTranslatef(0.0f, 0.45f, 0.0f); // Head
            gluSphere(pObj, 0.24f, 26, 13);
            // Eyes
            glColor3f(0.0f, 0.0f, 0.0f);
            glTranslatef(0.1f, 0.1f, 0.21f);
            gluSphere(pObj, 0.02f, 26, 13);
            glTranslatef(-0.2f, 0.0f, 0.0f);
            gluSphere(pObj, 0.02f, 26, 13);
            // Nose
            glColor3f(1.0f, 0.3f, 0.3f);
            glTranslatef(0.1f, -0.12f, 0.0f);
            gluCylinder(pObj, 0.04f, 0.0f, 0.3f, 26, 13);
        glPopMatrix();
        // Hat
        glPushMatrix();
            glColor3f(0.0f, 0.0f, 0.0f);
            glTranslatef(0.0f, 1.17f, 0.0f);
            glRotatef(-90.0f, 1.0f, 0.0f, 0.0f);
            gluCylinder(pObj, 0.17f, 0.17f, 0.4f, 26, 13);
            // Hat brim
            glDisable(GL_CULL_FACE);
            gluDisk(pObj, 0.17f, 0.28f, 26, 13);
            glEnable(GL_CULL_FACE);
            glTranslatef(0.0f, 0.0f, 0.40f);
            gluDisk(pObj, 0.0f, 0.17f, 26, 13);
        glPopMatrix();
        // Restore the matrix state
        glPopMatrix();
        // Buffer swap
        glutSwapBuffers();
}

```

}

Figure 10.9. A snowman rendered from quadric objects.

Bézier Curves and Surfaces

Quadrics provide built-in support for some very simple surfaces easily modeled with algebraic equations. Suppose, however, you want to create a curve or surface, and you don't have an algebraic equation to start with. It's far from a trivial task to figure out your surface in reverse, starting from what you visualize as the result and working down to a second- or third-order polynomial. Taking a rigorous mathematical approach is time consuming and error prone, even with the aid of a computer. You can also forget about trying to do it in your head.

Recognizing this fundamental need in the art of computer-generated graphics, Pierre Bézier, an automobile designer for Renault in the 1970s, created a set of mathematical models that could represent curves and surfaces by specifying only a small set of control points. In addition to simplifying the representation of curved surfaces, the models facilitated interactive adjustments to the shape of the curve or surface.

Other types of curves and surfaces and indeed a whole new vocabulary for computer-generated surfaces soon evolved. The mathematics behind this magic show are no more complex than the matrix manipulations in [Chapter 4, "Geometric Transformation: The Pipeline,"](#) and an intuitive understanding of these curves is easy to grasp. As we did in [Chapter 4](#), we take the approach that you can do a lot with these functions without a deep understanding of their mathematics.

Parametric Representation

A curve has a single starting point, a length, and an endpoint. It's really just a line that squiggles about in 3D space. A surface, on the other hand, has width and length and thus a surface area. We begin by showing you how to draw some smooth curves in 3D space and then extend this concept to surfaces. First, let's establish some common vocabulary and math fundamentals.

When you think of straight lines, you might think of this famous equation:

$$y = mx + b$$

Here, m equals the slope of the line, and b is the y intercept of the line (the place where the line crosses the y -axis). This discussion might take you back to your eighth-grade algebra class, where you also learned about the equations for parabolas, hyperbolas, exponential curves, and so on. All these equations expressed y (or x) in terms of some function of x (or y).

Another way of expressing the equation for a curve or line is as a parametric equation. A parametric equation expresses both x and y in terms of another variable that varies across some predefined range of values that is not explicitly a part of the geometry of the curve. Sometimes in physics, for example, the x , y , and z coordinates of a particle might be in terms of some functions of time, where time is expressed in seconds. In the following, $f()$, $g()$, and $h()$ are unique functions that vary with time (t):

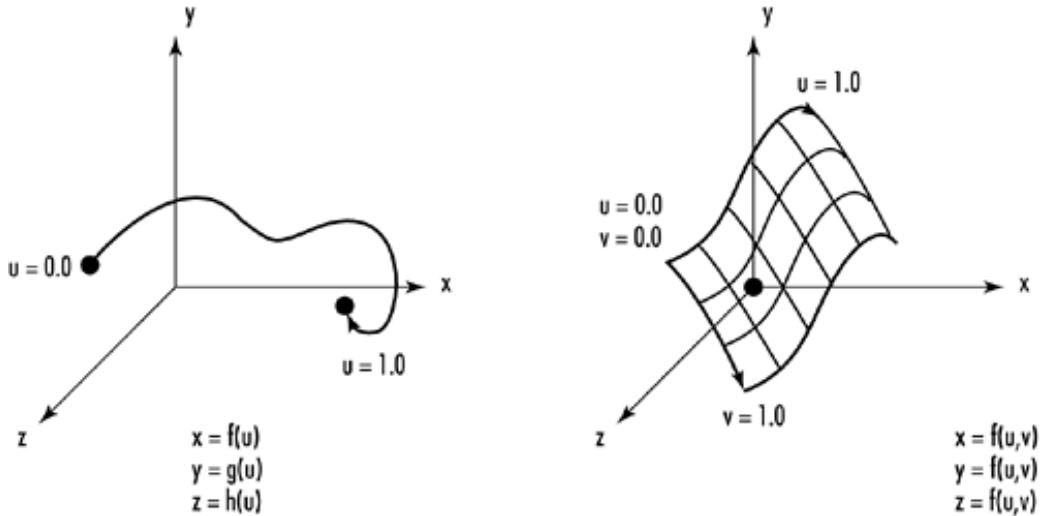
$$x = f(t)$$

$$y = g(t)$$

$$z = h(t)$$

When we define a curve in OpenGL, we also define it as a parametric equation. The parametric parameter of the curve, which we call u , and its range of values is the domain of that curve. Surfaces are described using two parametric parameters: u and v . [Figure 10.10](#) shows both a curve and a surface defined in terms of u and v domains. The important point to realize here is that the parametric parameters (u and v) represent the extents of the equations that describe the curve; they do not reflect actual coordinate values.

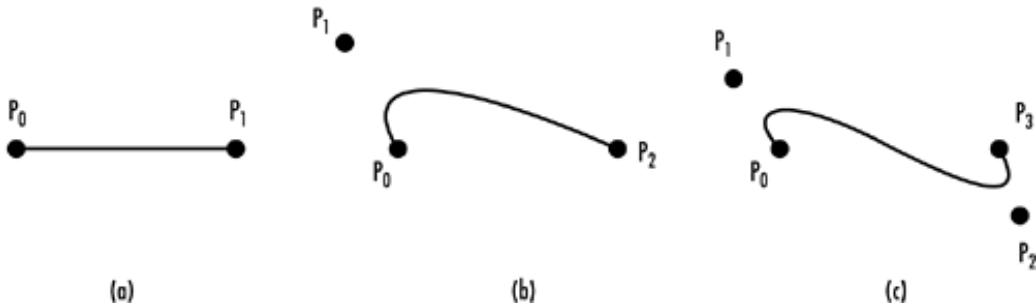
Figure 10.10. Parametric representation of curves and surfaces.



Control Points

A curve is represented by a number of control points that influence the shape of the curve. For a Bézier curve, the first and last control points are actually part of the curve. The other control points act as magnets, pulling the curve toward them. [Figure 10.11](#) shows some examples of this concept, with varying numbers of control points.

Figure 10.11. How control points affect curve shape.



The *order* of the curve is represented by the number of control points used to describe its shape. The *degree* is one less than the order of the curve. The mathematical meaning of these terms pertains to the parametric equations that exactly describe the curve, with the order being the number of coefficients and the degree being the highest exponent of the parametric parameter. If you want to read more about the mathematical basis of Bézier curves, see [Appendix A](#), "Further Reading."

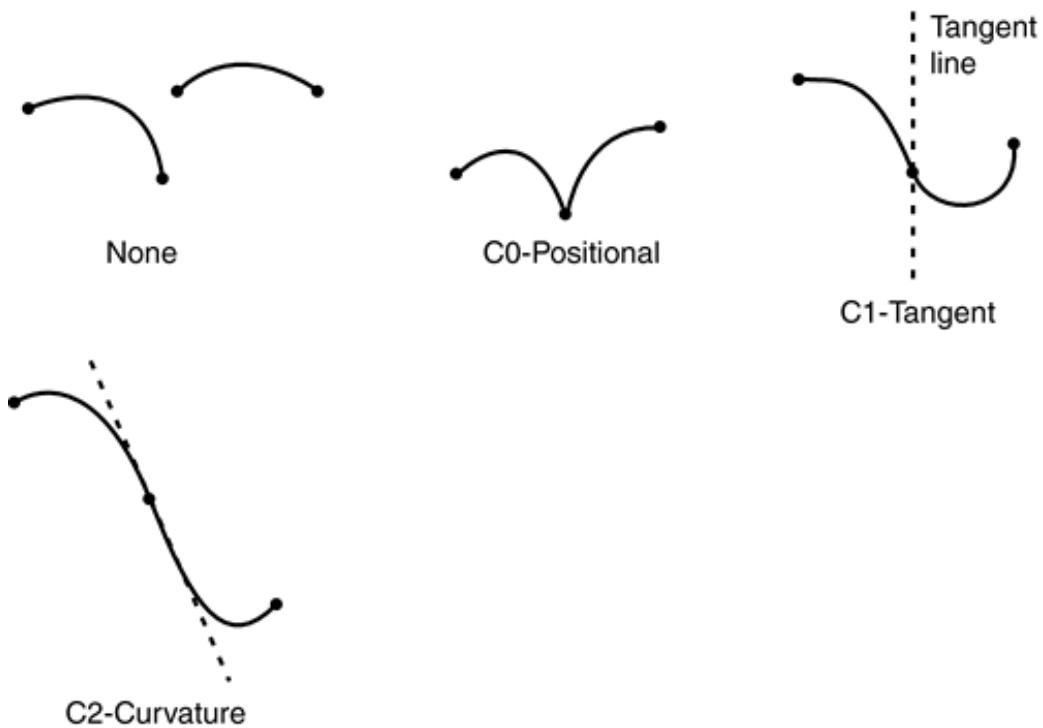
The curve in [Figure 10.11\(b\)](#) is called a *quadratic* curve (degree 2), and [Figure 10.11\(c\)](#) is called a *cubic* (degree 3). Cubic curves are the most typical. Theoretically, you could define a curve of any order, but higher order curves start to oscillate uncontrollably and can vary wildly with the slightest change to the control points.

Continuity

If two curves placed side by side share an endpoint (called the *breakpoint*), they together form a *piecewise* curve. The continuity of these curves at this breakpoint describes how smooth the transition is between them. The four categories of continuity are none, positional (C0), tangential (C1), and curvature (C2).

As you can see in [Figure 10.12](#), no continuity occurs when the two curves don't meet at all. Positional continuity is achieved when the curves at least meet and share a common endpoint. Tangential continuity occurs when the two curves have the same tangent at the breakpoint. Finally, curvature continuity means the two curves' tangents also have the same rate of change at the breakpoint (thus an even smoother transition).

Figure 10.12. Continuity of piecewise curves.



When assembling complex surfaces or curves from many pieces, you usually strive for tangential or curvature continuity. You'll see later that some parameters for curve and surface generation can be chosen to produce the desired continuity.

Evaluators

OpenGL contains several functions that make it easy to draw Bézier curves and surfaces. To draw them, you specify the control points and the range for the parametric u and v parameters. Then, by calling the appropriate evaluation function (the evaluator), OpenGL generates the points that make up the curve or surface. We start with a 2D example of a Bézier curve and then extend it to three dimensions to create a Bézier surface.

A 2D Curve

The best way to start is to go through an example, explaining it line by line. [Listing 10.2](#) shows some code from the sample program BEZIER in this chapter's subdirectory on the CD. This program specifies four control points for a Bézier curve and then renders the curve using an evaluator. The output from [Listing 10.2](#) is shown in [Figure 10.13](#).

Listing 10.2. Code from BEZIER That Draws a Bézier Curve with Four Control Points

```

// The number of control points for this curve
GLint nNumPoints = 4;
GLfloat ctrlPoints[4][3] = {{ -4.0f, 0.0f, 0.0f}, // End Point
                           { -6.0f, 4.0f, 0.0f}, // Control Point
                           { 6.0f, -4.0f, 0.0f}, // Control Point
                           { 4.0f, 0.0f, 0.0f }}; // End Point
// This function is used to superimpose the control points over the curve
void DrawPoints(void)
{
    int i; // Counting variable
    // Set point size larger to make more visible
    glPointSize(5.0f);
    // Loop through all control points for this example
    glBegin(GL_POINTS);
        for(i = 0; i < nNumPoints; i++)

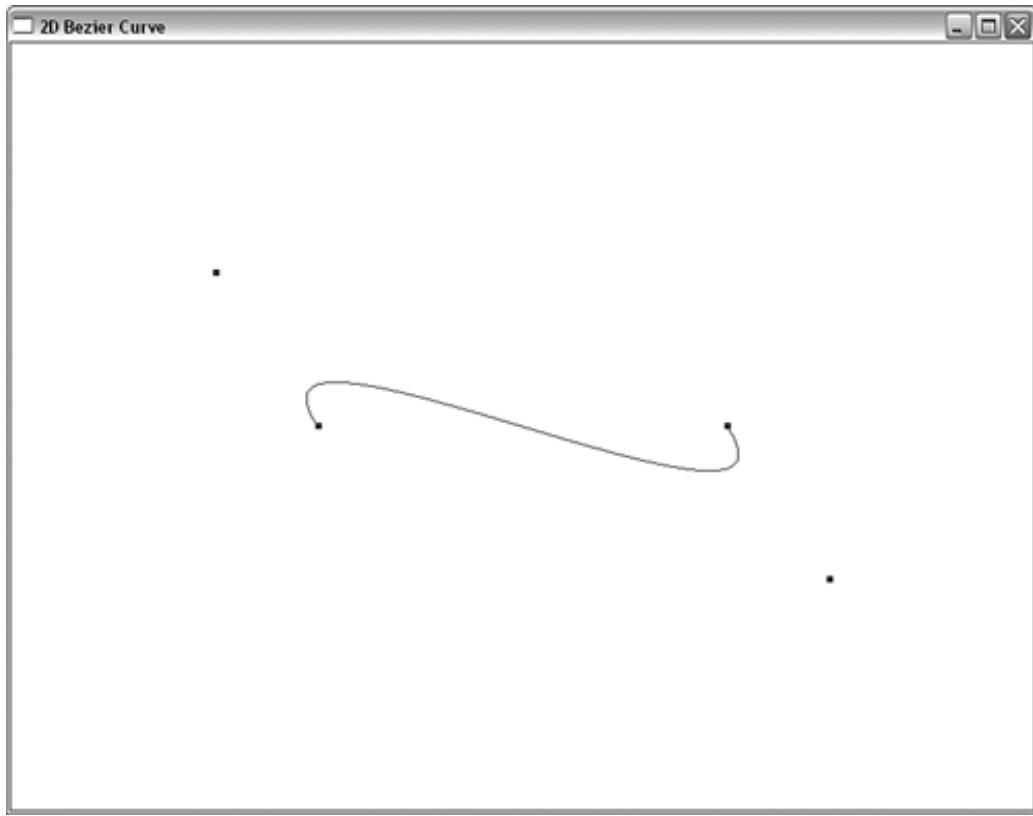
```

```

        glVertex2fv(ctrlPoints[i]);
    glEnd();
}
// Called to draw scene
void RenderScene(void)
{
    int i;
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT);
    // Sets up the bezier
    // This actually only needs to be called once and could go in
    // the setup function
    glMap1f(GL_MAP1_VERTEX_3, // Type of data generated
    0.0f,                      // Lower u range
    100.0f,                     // Upper u range
    3,                          // Distance between points in the data
    nNumPoints,                 // number of control points
    &ctrlPoints[0][0]);        // array of control points
    // Enable the evaluator
    glEnable(GL_MAP1_VERTEX_3);
    // Use a line strip to "connect-the-dots"
    glBegin(GL_LINE_STRIP);
        for(i = 0; i <= 100; i++)
        {
            // Evaluate the curve at this point
            glEvalCoord1f((GLfloat) i);
        }
    glEnd();
    // Draw the Control Points
    DrawPoints();
    // Flush drawing commands
    glutSwapBuffers();
}
// This function does any needed initialization on the rendering
// context.
void SetupRC()
{
    // Clear Window to white
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f );
    // Draw in Blue
    glColor3f(0.0f, 0.0f, 1.0f);
}
///////////////////////////////
// Set 2D Projection
void ChangeSize(int w, int h)
{
    // Prevent a divide by zero
    if(h == 0)
        h = 1;
    // Set Viewport to window dimensions
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-10.0f, 10.0f, -10.0f, 10.0f);
    // Modelview matrix reset
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

```

Figure 10.13. Output from the BEZIER sample program.



The first thing we do in [Listing 10.2](#) is define the control points for our curve:

```
// The number of control points for this curve
GLint nNumPoints = 4;
GLfloat ctrlPoints[4][3] = {{ -4.0f, 0.0f, 0.0f},      // Endpoint
                           { -6.0f, 4.0f, 0.0f},      // Control point
                           { 6.0f, -4.0f, 0.0f},      // Control point
                           { 4.0f, 0.0f, 0.0f } };    // Endpoint
```

We defined global variables for the number of control points and the array of control points. To experiment, you can change them by adding more control points or just modifying the position of these points.

The `DrawPoints` function is reasonably straightforward. We call this function from our rendering code to display the control points along with the curve. This capability also is useful when you're experimenting with control-point placement. Our standard `ChangeSize` function establishes a 2D orthographic projection that spans from -10 to $+10$ in the x and y directions.

Finally, we get to the rendering code. The `RenderScene` function first calls `glMap1f` (after clearing the screen) to create a mapping for our curve:

```
// Called to draw scene
void RenderScene(void)
{
    int i;
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT);
    // Sets up the Bezier
    // This actually only needs to be called once and could go in
    // the setup function
    glMap1f(GL_MAP1_VERTEX_3,      // Type of data generated
            0.0f,                // Lower u range
            100.0f);             // Upper u range
```

```

3,                                // Distance between points in the data
nNumPoints,                         // Number of control points
&ctrlPoints[0][0]);                // Array of control points
...
...

```

The first parameter to `glMap1f`, `GL_MAP1_VERTEX_3`, sets up the evaluator to generate vertex coordinate triplets (x, y, and z). You can also have the evaluator generate other values, such as texture coordinates and color information. See the reference section at the end of this chapter for details.

The next two parameters specify the lower and upper bounds of the parametric u value for this curve. The lower value specifies the first point on the curve, and the upper value specifies the last point on the curve. All the values in between correspond to the other points along the curve. Here, we set the range to 0–100.

The fourth parameter to `glMap1f` specifies the number of floating-point values between the vertices in the array of control points. Each vertex consists of three floating-point values (for x, y, and z), so we set this value to 3. This flexibility allows the control points to be placed in an arbitrary data structure, as long as they occur at regular intervals.

The last parameter is a pointer to a buffer containing the control points used to define the curve. Here, we pass a pointer to the first element of the array. After creating the mapping for the curve, we enable the evaluator to make use of this mapping. This capability is maintained through a state variable, and the following function call is all that is needed to enable the evaluator to produce points along the curve:

```

// Enable the evaluator
 glEnable(GL_MAP1_VERTEX_3);

```

The `glEvalCoord1f` function takes a single argument: a parametric value along the curve. This function then evaluates the curve at this value and calls `glVertex` internally for that point. By looping through the domain of the curve and calling `glEvalCoord` to produce vertices, we can draw the curve with a simple line strip:

```

// Use a line strip to "connect the dots"
 glBegin(GL_LINE_STRIP);
 for(i = 0; i <= 100; i++)
 {
    // Evaluate the curve at this point
    glEvalCoord1f((GLfloat) i);
 }
 glEnd();

```

Finally, we want to display the control points themselves:

```

// Draw the control points
DrawPoints();

```

Evaluating a Curve

OpenGL can make things even easier than what we've done so far. We set up a grid with the `glMapGrid` function, which tells OpenGL to create an evenly spaced grid of points over the u domain (the parametric argument of the curve). Then we call `glEvalMesh` to "connect the dots" using the primitive specified (`GL_LINE` or `GL_POINTS`). The two function calls

```

// Use higher level functions to map to a grid, then evaluate the
// entire thing.

```

```
// Map a grid of 100 points from 0 to 100
glMapGrid1d(100,0.0,100.0);
// Evaluate the grid, using lines
glEvalMesh1(GL_LINE,0,100);
```

completely replace the code

```
// Use a line strip to "connect-the-dots"
glBegin(GL_LINE_STRIP);
for(i = 0; i <= 100; i++)
{
    // Evaluate the curve at this point
    glEvalCoord1f((GLfloat) i);
}
glEnd();
```

As you can see, this approach is more compact and efficient, but its real benefit comes when evaluating surfaces rather than curves.

A 3D Surface

Creating a 3D Bézier surface is much like creating the 2D version. In addition to defining points along the u domain, we must define them along the v domain. [Listing 10.3](#) contains code from our next sample program, BEZ3D, and displays a wire mesh of a 3D Bézier surface. The first change from the preceding example is that we have defined three more sets of control points for the surface along the v domain. To keep this surface simple, we've kept the same control points except for the z value. This way, we create a uniform surface, as if we simply extruded a 2D Bézier along the z-axis.

Listing 10.3. BEZ3D Code to Create a Bézier Surface

```
// The number of control points for this curve
GLint nNumPoints = 3;
GLfloat ctrlPoints[3][3][3] = {{{{-4.0f, 0.0f, 4.0f},
                                  {-2.0f, 4.0f, 4.0f},
                                  {4.0f, 0.0f, 4.0f}}},
                                  {{{-4.0f, 0.0f, 0.0f},
                                  {-2.0f, 4.0f, 0.0f},
                                  {4.0f, 0.0f, 0.0f}}},
                                  {{{-4.0f, 0.0f, -4.0f},
                                  {-2.0f, 4.0f, -4.0f},
                                  {4.0f, 0.0f, -4.0f}}}};
// This function is used to superimpose the control points over the curve
void DrawPoints(void)
{
    int i,j;      // Counting variables
    // Set point size larger to make more visible
    glPointSize(5.0f);
    // Loop through all control points for this example
    glBegin(GL_POINTS);
    for(i = 0; i < nNumPoints; i++)
        for(j = 0; j < 3; j++)
            glVertex3fv(ctrlPoints[i][j]);
    glEnd();
}
// Called to draw scene
void RenderScene(void)
{
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT);
    // Save the modelview matrix stack
```

```

glMatrixMode(GL_MODELVIEW);
glPushMatrix();
// Rotate the mesh around to make it easier to see
glRotatef(45.0f, 0.0f, 1.0f, 0.0f);
glRotatef(60.0f, 1.0f, 0.0f, 0.0f);
// Sets up the Bezier
// This actually only needs to be called once and could go in
// the setup function
glMap2f(GL_MAP2_VERTEX_3,           // Type of data generated
0.0f,                                // Lower u range
10.0f,                               // Upper u range
3,                                     // Distance between points in the data
3,                                     // Dimension in u direction (order)
0.0f,                                // Lower v range
10.0f,                               // Upper v range
9,                                     // Distance between points in the data
3,                                     // Dimension in v direction (order)
&ctrlPoints[0][0][0]);                // array of control points
// Enable the evaluator
glEnable(GL_MAP2_VERTEX_3);
// Use higher level functions to map to a grid, then evaluate the
// entire thing.
// Map a grid of 10 points from 0 to 10
glMapGrid2f(10,0.0f,10.0f,10,0.0f,10.0f);
// Evaluate the grid, using lines
glEvalMesh2(GL_LINE,0,10,0,10);
// Draw the Control Points
DrawPoints();
// Restore the modelview matrix
glPopMatrix();
// Display the image
glutSwapBuffers();
}

```

Our rendering code is different now, too. In addition to rotating the figure for a better visual effect, we call `glMap2f` instead of `glMap1f`. This call specifies control points along two domains (u and v) instead of just one (u):

```

// Sets up the Bezier
// This actually only needs to be called once and could go in
// the setup function
glMap2f(GL_MAP2_VERTEX_3,           // Type of data generated
0.0f,                                // Lower u range
10.0f,                               // Upper u range
3,                                     // Distance between points in the data
3,                                     // Dimension in u direction (order)
0.0f,                                // Lower v range
10.0f,                               // Upper v range
9,                                     // Distance between points in the data
3,                                     // Dimension in v direction (order)
&ctrlPoints[0][0][0]);                // Array of control points

```

We must still specify the lower and upper range for u, and the distance between points in the u domain is still 3. Now, however, we must also specify the lower and upper range in the v domain. The distance between points in the v domain is now nine values because we have a three-dimensional array of control points, with each span in the u domain being three points of three values each ($3 \times 3 = 9$). Then we tell `glMap2f` how many points in the v direction are specified for each u division, followed by a pointer to the control points themselves.

The two-dimensional evaluator is enabled just like the one-dimensional version, and we call

`glMapGrid2f` with the number of divisions in the u and v direction:

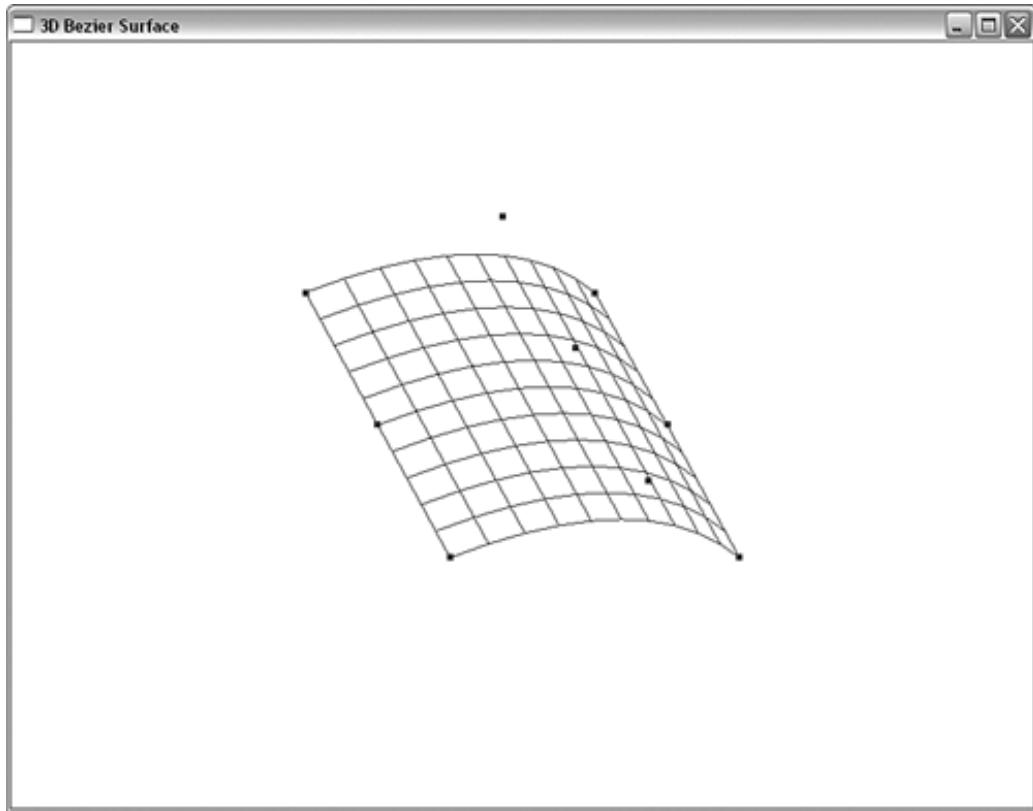
```
// Enable the evaluator
glEnable(GL_MAP2_VERTEX_3);
// Use higher level functions to map to a grid, then evaluate the
// entire thing.
// Map a grid of 10 points from 0 to 10
glMapGrid2f(10,0.0f,10.0f,10,0.0f,10.0f);
```

After the evaluator is set up, we can call the two-dimensional (meaning u and v) version of `glEvalMesh` to evaluate our surface grid. Here, we evaluate using lines and specify the u and v domains' values to range from 0 to 10:

```
// Evaluate the grid, using lines
glEvalMesh2(GL_LINE,0,10,0,10);
```

The result is shown in [Figure 10.14](#).

Figure 10.14. Output from the BEZ3D program.



Lighting and Normal Vectors

Another valuable feature of evaluators is the automatic generation of surface normals. By simply changing this code

```
// Evaluate the grid, using lines
glEvalMesh2(GL_LINE,0,10,0,10);
```

to this

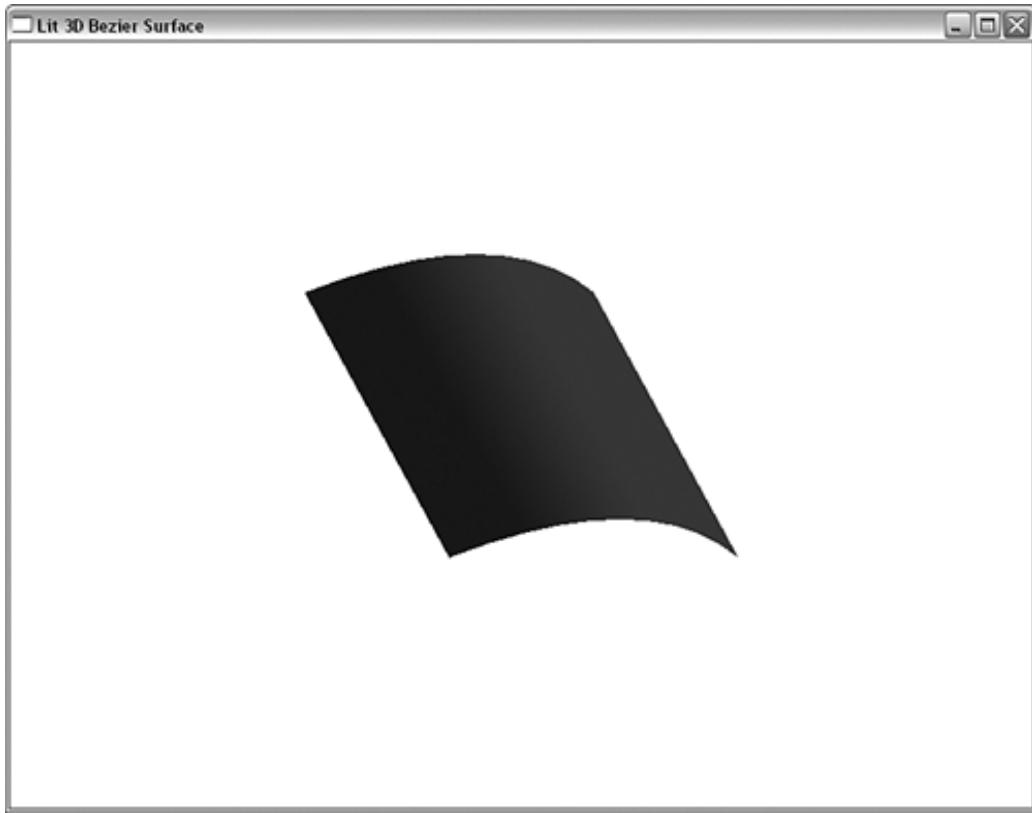
```
// Evaluate the grid, using lines
glEvalMesh2(GL_FILL,0,10,0,10);
```

and then calling

```
glEnable(GL_AUTO_NORMAL);
```

in our initialization code, we enable easy lighting of surfaces generated by evaluators. [Figure 10.15](#) shows the same surface as [Figure 10.14](#), but with lighting enabled and automatic normalization turned on. The code for this program appears in the BEZLIT sample in the CD subdirectory for this chapter. The program is only slightly modified from BEZ3D.

Figure 10.15. Output from the BEZLIT program.

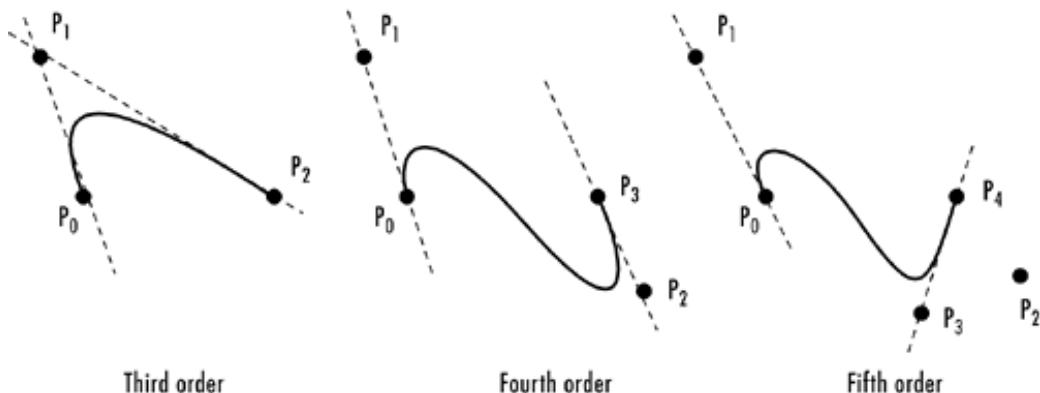


NURBS

You can use evaluators to your heart's content to evaluate Bézier surfaces of any degree, but for more complex curves, you have to assemble your Béziers piecewise. As you add more control points, creating a curve that has good continuity becomes difficult. A higher level of control is available through the GLU library's NURBS functions. NURBS stands for *non-uniform rational B-spline*. Mathematicians out there might know immediately that this is just a more generalized form of curves and surfaces that can produce Bézier curves and surfaces, as well as some other kinds (mathematically speaking). These functions allow you to tweak the influence of the control points you specified for the evaluators to produce smoother curves and surfaces with larger numbers of control points.

From Bézier to B-Splines

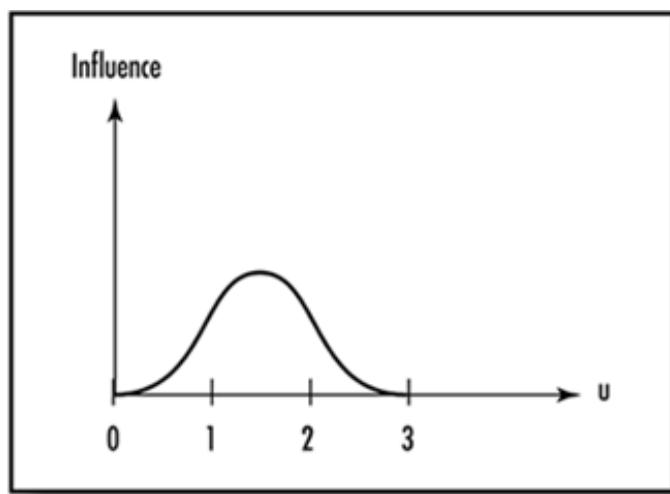
A Bézier curve is defined by two points that act as endpoints and any number of other control points that influence the shape of the curve. The three Bézier curves in [Figure 10.16](#) have three, four, and five control points specified. The curve is tangent to a line that connects the endpoints with their adjacent control points. For quadratic (three points) and cubic (four points) curves, the resulting Béziers are quite smooth, usually with a continuity of C2 (curvature). For higher numbers of control points, however, the smoothness begins to break down as the additional control points pull and tug on the curve.

Figure 10.16. Bézier continuity as the order of the curve increases.

B-splines (bi-cubic splines), on the other hand, work much as the Bézier curves do, but the curve is broken down into segments. The shape of any given segment is influenced only by the nearest four control points, producing a piecewise assemblage of a curve with each segment exhibiting characteristics much like a fourth-order Bézier curve. A long curve with many control points is inherently smoother, with the junction between each segment exhibiting C₂ continuity. It also means that the curve does not necessarily have to pass through any of the control points.

Knots

The real power of NURBS is that you can tweak the influence of the four control points for any given segment of a curve to produce the smoothness needed. This control is handled via a sequence of values called *knots*. Two knot values are defined for every control point. The range of values for the knots matches the u or v parametric domain and must be nondescending. The knot values determine the influence of the control points that fall within that range in u/v space. [Figure 10.17](#) shows a curve demonstrating the influence of control points over a curve having four units in the u parametric domain. Points in the middle of the u domain have a greater pull on the curve, and only points between 0 and 3 have any effect on the shape of the curve.

Figure 10.17. Control point influence along the u parameter.

The key here is that one of these influence curves exists at each control point along the u/v parametric domain. The knot sequence then defines the strength of the influence of points within this domain. If a knot value is repeated, points near this parametric value have even greater influence. The repeating of knot values is called *knot multiplicity*. Higher knot multiplicity

decreases the curvature of the curve or surface within that region.

Creating a NURBS Surface

The GLU NURBS functions provide a useful high-level facility for rendering surfaces. You don't have to explicitly call the evaluators or establish the mappings or grids. To render a NURBS, you first create a NURBS object that you reference whenever you call the NURBS-related functions to modify the appearance of the surface or curve.

The `gluNewNurbsRenderer` function creates a renderer for the NURB, and `gluDeleteNurbsRenderer` destroys it. The following code fragments demonstrate these functions in use:

```
// NURBS object pointer
GLUnurbsObj *pNurb = NULL;
...
...
// Setup the NURBS object
pNurb = gluNewNurbsRenderer();
...
// Do your NURBS things...
...
...
// Delete the NURBS object if it was created
if(pNurb)
    gluDeleteNurbsRenderer(pNurb);
```

NURBS Properties

After you have created a NURBS renderer, you can set various high-level NURBS properties for the NURB:

```
// Set sampling tolerance
gluNurbsProperty(pNurb, GLU_SAMPLING_TOLERANCE, 25.0f);
// Fill to make a solid surface (use GLU_OUTLINE_POLYGON to create a
// polygon mesh)
gluNurbsProperty(pNurb, GLU_DISPLAY_MODE, (GLfloat)GLU_FILL);
```

You typically call these functions in your setup routine rather than repeatedly in your rendering code. In this example, `GLU_SAMPLING_TOLERANCE` defines the fineness of the mesh that defines the surface, and `GLU_FILL` tells OpenGL to fill in the mesh instead of generating a wireframe.

Defining the Surface

The surface definition is passed as arrays of control points and knot sequences to the `gluNurbsSurface` function. As shown here, this function is also bracketed by calls to `gluBeginSurface` and `gluEndSurface`:

```
// Render the NURB
// Begin the NURB definition
gluBeginSurface(pNurb);
// Evaluate the surface
gluNurbsSurface(pNurb, // Pointer to NURBS renderer
    8, Knots,           // No. of knots and knot array u direction
    8, Knots,           // No. of knots and knot array v direction
    4 * 3,              // Distance between control points in u dir.
    3,                  // Distance between control points in v dir.
    &ctrlPoints[0][0][0], // Control points
```

```

4, 4,           // u and v order of surface
GL_MAP2_VERTEX_3); // Type of surface
// Done with surface
gluEndSurface(pNurb);

```

You can make more calls to `gluNurbsSurface` to create any number of NURBS surfaces, but the properties you set for the NURBS renderer are still in effect. Often, this is desired; you rarely want two surfaces (perhaps joined) to have different fill styles (one filled and one a wire mesh).

Using the control points and knot values shown in the next code segment, we produced the NURBS surface shown in [Figure 10.18](#). You can find this NURBS program in this chapter's subdirectory on the CD:

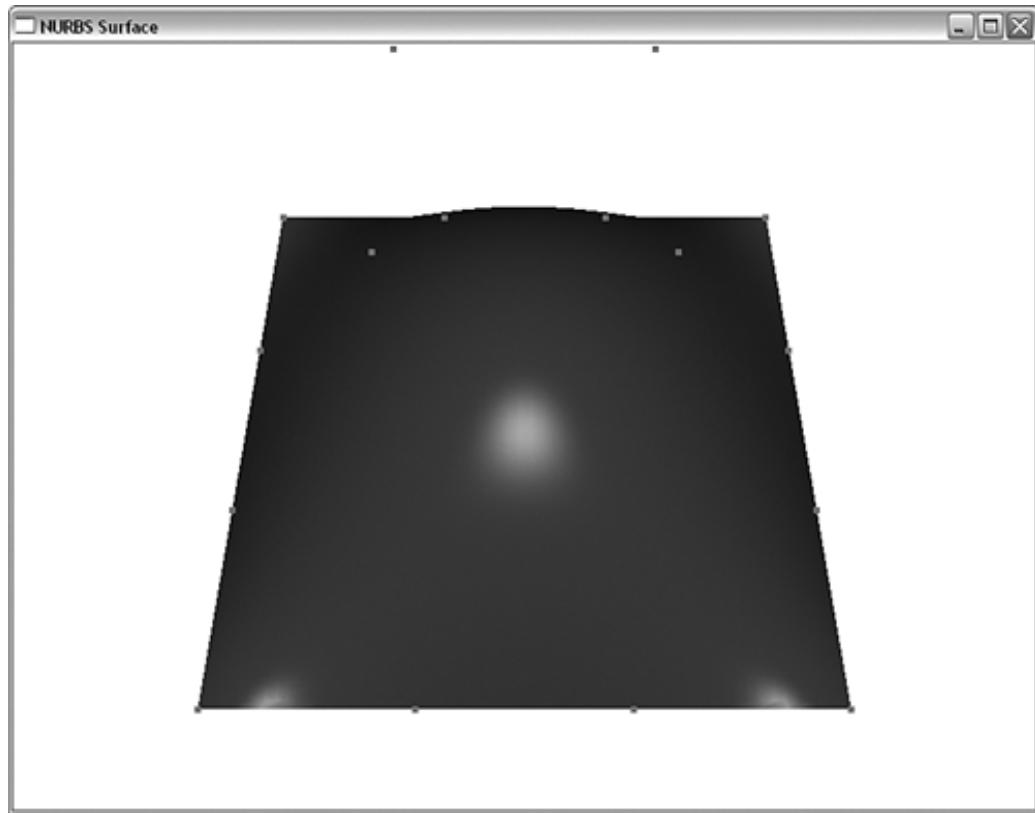
```

// Mesh extends four units -6 to +6 along x and y axis
// Lies in Z plane
//      u  v  (x,y,z)
GLfloat ctrlPoints[4][4][3]= { {{ { -6.0f, -6.0f, 0.0f},      // u = 0,   v = 0
{ -6.0f, -2.0f, 0.0f},      //           v = 1
{ -6.0f,  2.0f, 0.0f},      //           v = 2
{ -6.0f,  6.0f, 0.0f} },     //           v = 3
{ { -2.0f, -6.0f, 0.0f},      // u = 1     v = 0
{ -2.0f, -2.0f, 8.0f},      //           v = 1
{ -2.0f,  2.0f, 8.0f},      //           v = 2
{ -2.0f,  6.0f, 0.0f} },     //           v = 3
{ { 2.0f, -6.0f, 0.0f },     // u = 2     v = 0
{ 2.0f, -2.0f, 8.0f },     //           v = 1
{ 2.0f,  2.0f, 8.0f },     //           v = 2
{ 2.0f,  6.0f, 0.0f } },     //           v = 3
{ { 6.0f, -6.0f, 0.0f},      // u = 3     v = 0
{ 6.0f, -2.0f, 0.0f},      //           v = 1
{ 6.0f,  2.0f, 0.0f},      //           v = 2
{ 6.0f,  6.0f, 0.0f} } };    //           v = 3

// Knot sequence for the NURB
GLfloat Knots[8] = { 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f };

```

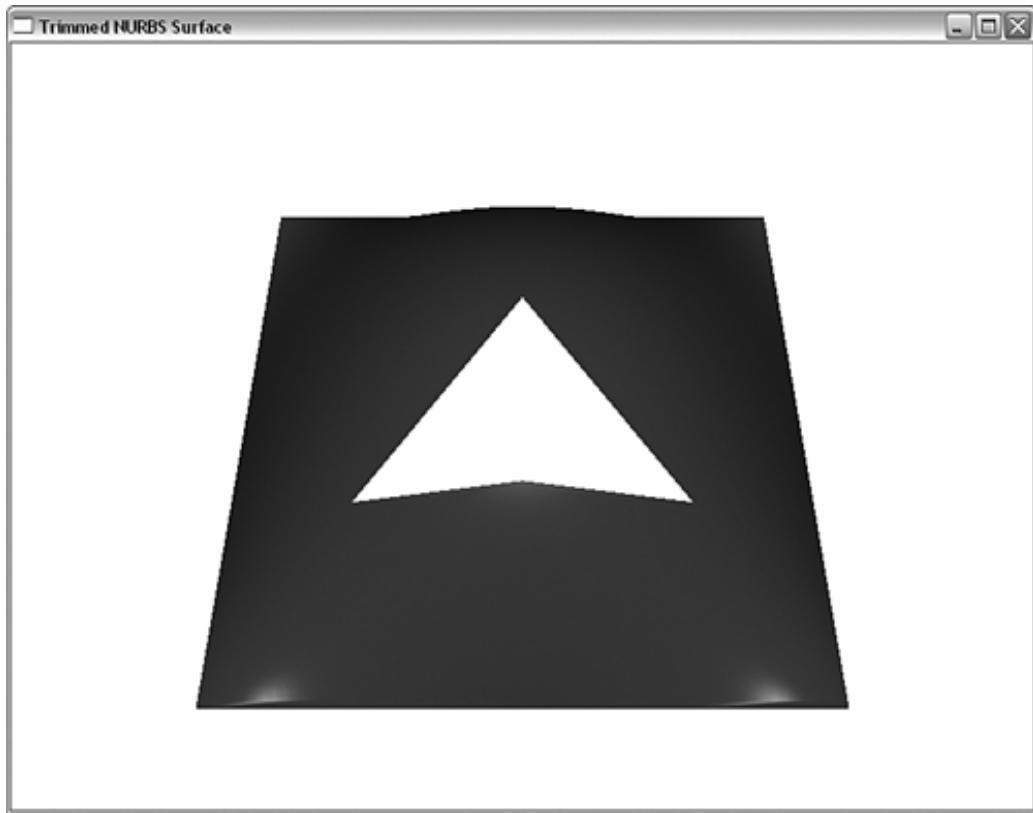
Figure 10.18. Output from the NURBS program.



Trimming

Trimming means creating cutout sections from NURBS surfaces. This capability is often used for literally trimming sharp edges of a NURBS surface. You can also create holes in your surface just as easily. The output from the NURBT program is shown in [Figure 10.19](#). This is the same NURBS surface used in the preceding sample (without the control points shown), with a triangular region removed. This program, too, is on the CD.

Figure 10.19. Output from the NURBT program.



[Listing 10.4](#) shows the code added to the NURBS sample program to produce this trimming effect. Within the `gluBeginSurface/gluEndSurface` delimiters, we call `gluBeginTrim`, specify a trimming curve with `gluPwlCurve`, and finish the trimming curve with `gluEndTrim`.

Listing 10.4. Modifications to NURBS to Produce Trimming

```

// Outside trimming points to include entire surface
GLfloat outsidePts[5][2] = /* counterclockwise */
    {{0.0f, 0.0f}, {1.0f, 0.0f}, {1.0f, 1.0f}, {0.0f, 1.0f}, {0.0f, 0.0f}};
// Inside trimming points to create triangle shaped hole in surface
GLfloat insidePts[4][2] = /* clockwise */
    {{0.25f, 0.25f}, {0.5f, 0.5f}, {0.75f, 0.25f}, {0.25f, 0.25f}};
...
...
...
// Render the NURB
// Begin the NURB definition
gluBeginSurface(pNurb);
// Evaluate the surface
gluNurbsSurface(pNurb, // Pointer to NURBS renderer
    8, Knots, // No. of knots and knot array u direction
    8, Knots, // No. of knots and knot array v direction
    4 * 3, // Distance between control points in u dir.
    3, // Distance between control points in v dir.
    &ctrlPoints[0][0][0], // Control points
    4, 4, // u and v order of surface
    GL_MAP2_VERTEX_3); // Type of surface
// Outer area, include entire curve
gluBeginTrim (pNurb);
gluPwlCurve (pNurb, 5, &outsidePts[0][0], 2, GLU_MAP1_TRIM_2);
gluEndTrim (pNurb);
// Inner triangular area
gluBeginTrim (pNurb);
gluPwlCurve (pNurb, 4, &insidePts[0][0], 2, GLU_MAP1_TRIM_2);

```

```

gluEndTrim (pNurb);
// Done with surface
gluEndSurface(pNurb);

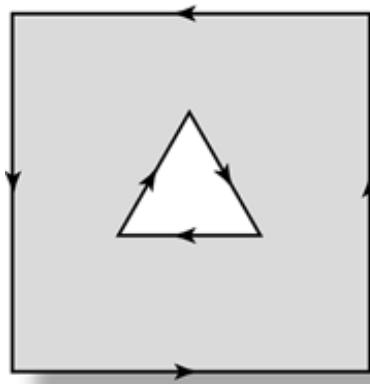
```

Within the `gluBeginTrim`/`gluEndTrim` delimiters, you can specify any number of curves as long as they form a closed loop in a piecewise fashion. You can also use `gluNurbsCurve` to define a trimming region or part of a trimming region. These trimming curves must, however, be in terms of the unit parametric u and v space. This means the entire u/v domain is scaled from 0.0 to 1.0.

`gluPwlCurve` defines a piecewise linear curve--nothing more than a list of points connected end to end. In this scenario, the inner trimming curve forms a triangle, but with many points, you could create an approximation of any curve needed.

Trimming a curve trims away surface area that is to the right of the curve's winding. Thus, a clockwise-wound trimming curve discards its interior. Typically, an outer trimming curve is specified, which encloses the entire NURBS parameter space. Then smaller trimming regions are specified within this region with clockwise winding. [Figure 10.20](#) illustrates this relationship.

Figure 10.20. An area inside clockwise-wound curves is trimmed away.



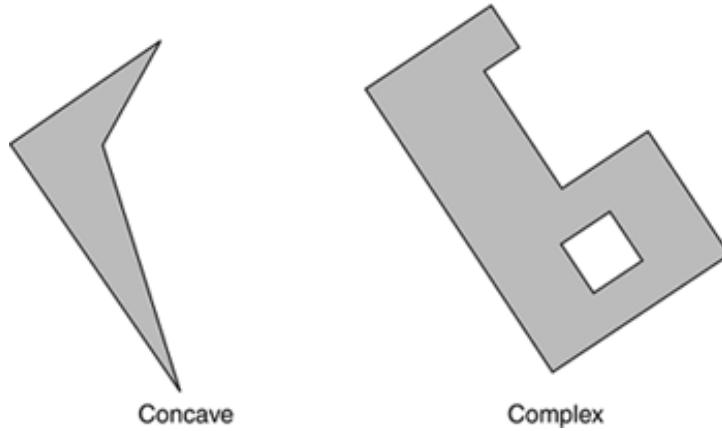
NURBS Curves

Just as you can have Bézier surfaces and curves, you can also have NURBS surfaces and curves. You can even use `gluNurbsCurve` to do NURBS surface trimming. By this point, we hope you have the basics down well enough to try trimming surfaces on your own. However, another sample, NURBC, is included on the CD if you want a starting point to play with.

Tessellation

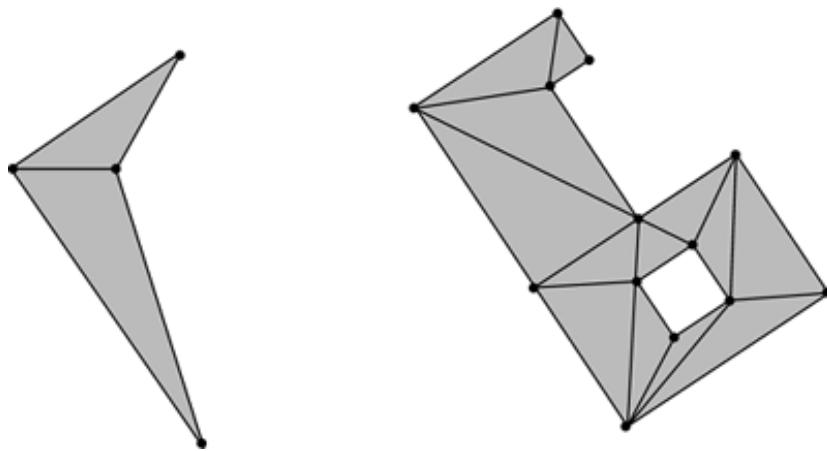
To keep OpenGL as fast as possible, all geometric primitives must be convex. We made this point in [Chapter 3](#), "Drawing in Space: Geometric Primitives and Buffers." However, many times we have vertex data for a concave or more complex shape that we want to render with OpenGL. These shapes fall into two basic categories, as shown in [Figure 10.21](#). A simple concave polygon is shown on the left, and a more complex polygon with a hole in it is shown on the right. For the shape on the left, you might be tempted to try using `GL_POLYGON` as the primitive type, but the rendering would fail because OpenGL algorithms are optimized for convex polygons. As for the figure on the right...well, there is little hope for that shape at all!

Figure 10.21. Some nonconvex polygons.



The intuitive solution to both of these problems is to break down the shape into smaller convex polygons or triangles that can be fit together to create the final overall shape. [Figure 10.22](#) shows one possible solution to breaking the shapes in [Figure 10.21](#) into more manageable triangles.

Figure 10.22. Complex shapes broken down into triangles.



Breaking down the shapes by hand is tedious at best and possibly error prone. Fortunately, the OpenGL Utility Library contains functions to help you break concave and complex polygons into smaller, valid OpenGL primitives. The process of breaking down these polygons is called *tessellation*.

The Tessellator

Tessellation works through a tessellator object that must be created and destroyed much in the same way that we did for quadric state objects:

```
GLUTesselator *pTess;
pTess = gluNewTess();
. . .
// Do some tessellation
. . .
gluDeleteTess(pTess);
```

All the tessellation functions use the tessellator object as the first parameter. This allows you to have more than one tessellation object active at a time or interact with libraries or other code that also uses tessellation. The tessellation functions change the tessellator's state and behavior, and this allows you to make sure your changes affect only the object you are currently working with. Alas, yes, **GLUTesselator** has only one *l* and is thus misspelled!

The tessellator breaks up a polygon and renders it appropriately when you perform the following steps:

1. Create the tessellator object.
2. Set tessellator state and callbacks.
3. Start a polygon.
4. Start a contour.
5. Feed the tessellator the vertices that specify the contour.
6. End the contour.
7. Go back to step 4 if there are more contours.
8. End the polygon.

Each polygon consists of one or more contours. The polygon to the left in [Figure 10.21](#) contains one contour, simply the path around the outside of the polygon. The polygon on the right, however, has two contours: the outside edge and the edge around the inner hole. Polygons may contain any number of contours (several holes) or even nested contours (holes within holes). The actual work of tessellating the polygon does not occur until step 8. This task can sometimes be very time consuming, and if the geometry is static, it may be best to store these function calls in a display list (the next chapter discusses display lists).

Tessellator Callbacks

During tessellation, the tessellator calls a number of callback functions that you must provide. You use these callbacks to actually specify the vertex information and begin and end the primitives. The following function registers the callback functions:

```
void gluTessCallback(GLUTesselator *tobj, GLenum which, void (*fn)());
```

The first parameter is the tessellation object. The second specifies the type of callback being registered, and the last is the pointer to the callback function itself. You can specify various callbacks, which are listed in [Table 10.3](#) in the reference section. As an example, examine the following lines of code:

```
// Just call glBegin at beginning of triangle batch
gluTessCallback(pTess, GLU_TESS_BEGIN, glBegin);
// Just call glEnd at end of triangle batch
gluTessCallback(pTess, GLU_TESS_END, glEnd);
// Just call glVertex3dv for each vertex
gluTessCallback(pTess, GLU_TESS_VERTEX, glVertex3dv);
```

The `GLU_TESS_BEGIN` callback specifies the function to call at the beginning of each new primitive. Specifying `glBegin` simply tells the tessellator to call `glBegin` to begin a primitive batch. This may seem pointless, but you can also specify your own function here to do additional processing whenever a new primitive begins. For example, suppose you want to find out how many triangles are used in the final tessellated polygon.

The `GLU_TESS_END` callback, again, simply tells the tessellator to call `glEnd` and that you have no other specific code you want to inject into the process. Finally, the `GLU_TESS_VERTEX` call drops in a call to `glVertex3dv` to specify the tessellated vertex data. Tessellation requires that vertex data

be specified as double precision, and always uses three component vertices. Again, you could substitute your own function here to do some additional processing (such as adding color, normal, or texture coordinate information).

For an example of specifying your own callback (instead of cheating and just using existing OpenGL functions), the following code shows the registration of a function to report any errors that may occur during tessellation:

```
///////////////////////////////
// Tessellation error callback
void tessError(GLenum error)
{
    // Get error message string
    const char *szError = gluErrorString(error);
    // Set error message as window caption
    glutSetWindowTitle(szError);
}
. . .
. . .
// Register error callback
gluTessCallback(pTess, GLU_TESS_ERROR, tessError);
```

Specifying Vertex Data

To begin a polygon (this corresponds to step 3 shown earlier), you call the following function:

```
void gluTessBeginPolygon(GLUTesselator *tobj, void *data);
```

You first pass in the tessellator object and then a pointer to any user-defined data that you want associated with this tessellation. This data can be sent back to you during tessellation using the callback functions listed in [Table 10.3](#). Often, this is just `NULL`. To finish the polygon (step 8) and begin tessellation, call this function:

```
void gluTessEndPolygon(GLUTesselator *tobj);
```

Nested within the beginning and ending of the polygon, you specify one or more contours using the following pair of functions (steps 4 and 6):

```
void gluTessBeginContour(GLUTesselator *tobj);
void gluTessEndContour(GLUTesselator *tobj);
```

Finally, within the contour, you must add the vertices that make up that contour (step 5). The following function feeds the vertices, one at a time, to the tessellator:

```
void gluTessVertex(GLUTesselator *tobj, GLdouble v[3], void *data);
```

The `v` parameter contains the actual vertex data used for tessellator calculations. The `data` parameter is a pointer to the vertex data passed to the callback function specified by `GLU_VERTEX`. Why two different arguments to specify the same thing? Because the pointer to the vertex data may also point to additional information about the vertex (color, normals, and so on). If you specify your own function for `GLU_VERTEX` (instead of our cheat), you can access this additional vertex data in the callback routine.

Putting It All Together

Now let's look at an example that takes a complex polygon and performs tessellation to render a solid shape. The sample program FLORIDA contains the vertex information to draw the crude, but

recognizable, shape of the state of Florida. The program has three modes of rendering, accessible via the context menu: Line Loops, Concave Polygon, and Complex Polygon. The basic shape with Line Loops is shown in [Figure 10.23](#).

Figure 10.23. Basic outline of Florida.



[Listing 10.5](#) shows the vertex data and the rendering code that draws the outlines for the state and Lake Okeechobee.

Listing 10.5. Vertex Data and Drawing Code for State Outline

```
// Coast Line Data
#define COAST_POINTS 24
GLdouble vCoast[COAST_POINTS][3] = {{-70.0, 30.0, 0.0 },
{ -50.0, 30.0, 0.0 },
{ -50.0, 27.0, 0.0 },
{ -5.0, 27.0, 0.0 },
{ 0.0, 20.0, 0.0 },
{ 8.0, 10.0, 0.0 },
{ 12.0, 5.0, 0.0 },
{ 10.0, 0.0, 0.0 },
{ 15.0,-10.0, 0.0 },
{ 20.0,-20.0, 0.0 },
{ 20.0,-35.0, 0.0 },
{ 10.0,-40.0, 0.0 },
{ 0.0,-30.0, 0.0 },
{ -5.0,-20.0, 0.0 },
{ -12.0,-10.0, 0.0 },
{ -13.0, -5.0, 0.0 },
{ -12.0, 5.0, 0.0 },
{ -20.0, 10.0, 0.0 },
{ -30.0, 20.0, 0.0 },
{ -40.0, 15.0, 0.0 },
{ -50.0, 15.0, 0.0 },
{ -55.0, 20.0, 0.0 },
{ -60.0, 25.0, 0.0 },
{ -70.0, 25.0, 0.0 }};
```

```

// Lake Okeechobee
#define LAKE_POINTS 4
GLdouble vLake[LAKE_POINTS][3] = {{ 10.0, -20.0, 0.0 },
                                  { 15.0, -25.0, 0.0 },
                                  { 10.0, -30.0, 0.0 },
                                  { 5.0, -25.0, 0.0 }};
. . .
. . .

case DRAW_LOOPS:           // Draw line loops
{
    glColor3f(0.0f, 0.0f, 0.0f); // Just black outline
    // Line loop with coastline shape
    glBegin(GL_LINE_LOOP);
    for(i = 0; i < COAST_POINTS; i++)
        glVertex3dv(vCoast[i]);
    glEnd();
    // Line loop with shape of interior lake
    glBegin(GL_LINE_LOOP);
    for(i = 0; i < LAKE_POINTS; i++)
        glVertex3dv(vLake[i]);
    glEnd();
}
break;

```

For the Concave Polygon rendering mode, only the outside contour is drawn. This results in a solid filled shape, despite the fact that the polygon is clearly concave. This result is shown in [Figure 10.24](#), and the tessellation code is shown in [Listing 10.6](#).

Listing 10.6. Drawing a Convex Polygon

```

case DRAW_CONCAVE:           // Tessellate concave polygon
{
    // Tessellator object
    GLUTesselator *pTess;
    // Green polygon
    glColor3f(0.0f, 1.0f, 0.0f);
    // Create the tessellator object
    pTess = gluNewTess();
    // Set callback functions
    // Just call glBegin at beginning of triangle batch
    gluTessCallback(pTess, GLU_TESS_BEGIN, glBegin);
    // Just call glEnd at end of triangle batch
    gluTessCallback(pTess, GLU_TESS_END, glEnd);
    // Just call glVertex3dv for each vertex
    gluTessCallback(pTess, GLU_TESS_VERTEX, glVertex3dv);
    // Register error callback
    gluTessCallback(pTess, GLU_TESS_ERROR, tessError);
    // Begin the polygon
    gluTessBeginPolygon(pTess, NULL);
    // Begin the one and only contour
    gluTessBeginContour(pTess);
    // Feed in the list of vertices
    for(i = 0; i < COAST_POINTS; i++)
        gluTessVertex(pTess, vCoast[i], vCoast[i]); // Can't be NULL
    // Close contour and polygon
    gluTessEndContour(pTess);
    gluTessEndPolygon(pTess);
    // All done with tessellator object
    gluDeleteTess(pTess);
}

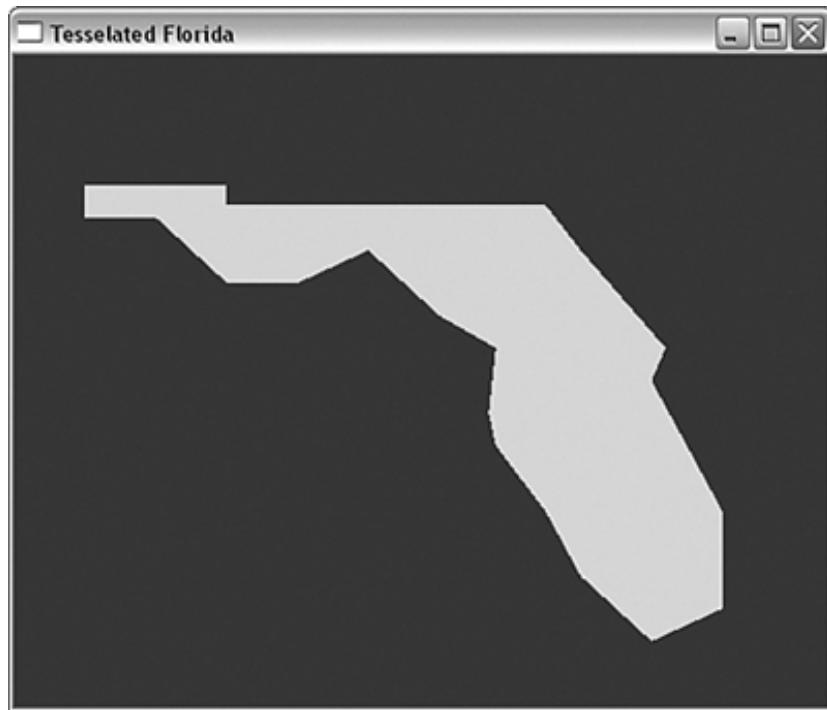
```

```

}
break;

```

Figure 10.24. A solid convex polygon.



Finally, we present a more complex polygon, one with a hole in it. The Complex Polygon drawing mode draws the solid state, but with a hole representing Lake Okeechobee (a large lake in south Florida, typically shown on maps). The output is shown in [Figure 10.25](#), and the relevant code is presented in [Listing 10.7](#).

Listing 10.7. Tessellating a Complex Polygon with Multiple Contours

```

case DRAW_COMPLEX:           // Tessellate, but with hole cut out
{
    // Tessellator object
    GLUtesselator *pTess;
    // Green polygon
    glColor3f(0.0f, 1.0f, 0.0f);
    // Create the tessellator object
    pTess = gluNewTess();
    // Set callback functions
    // Just call glBegin at beginning of triangle batch
    gluTessCallback(pTess, GLU_TESS_BEGIN, glBegin);
    // Just call glEnd at end of triangle batch
    gluTessCallback(pTess, GLU_TESS_END, glEnd);
    // Just call glVertex3dv for each vertex
    gluTessCallback(pTess, GLU_TESS_VERTEX, glVertex3dv);
    // Register error callback
    gluTessCallback(pTess, GLU_TESS_ERROR, tessError);
    // How to count filled and open areas
    gluTessProperty(pTess, GLU_TESS_WINDING_RULE, GLU_TESS_WINDING_ODD);
    // Begin the polygon
    gluTessBeginPolygon(pTess, NULL); // No user data
    // First contour, outline of state
    gluTessBeginContour(pTess);
    for(i = 0; i < COAST_POINTS; i++)

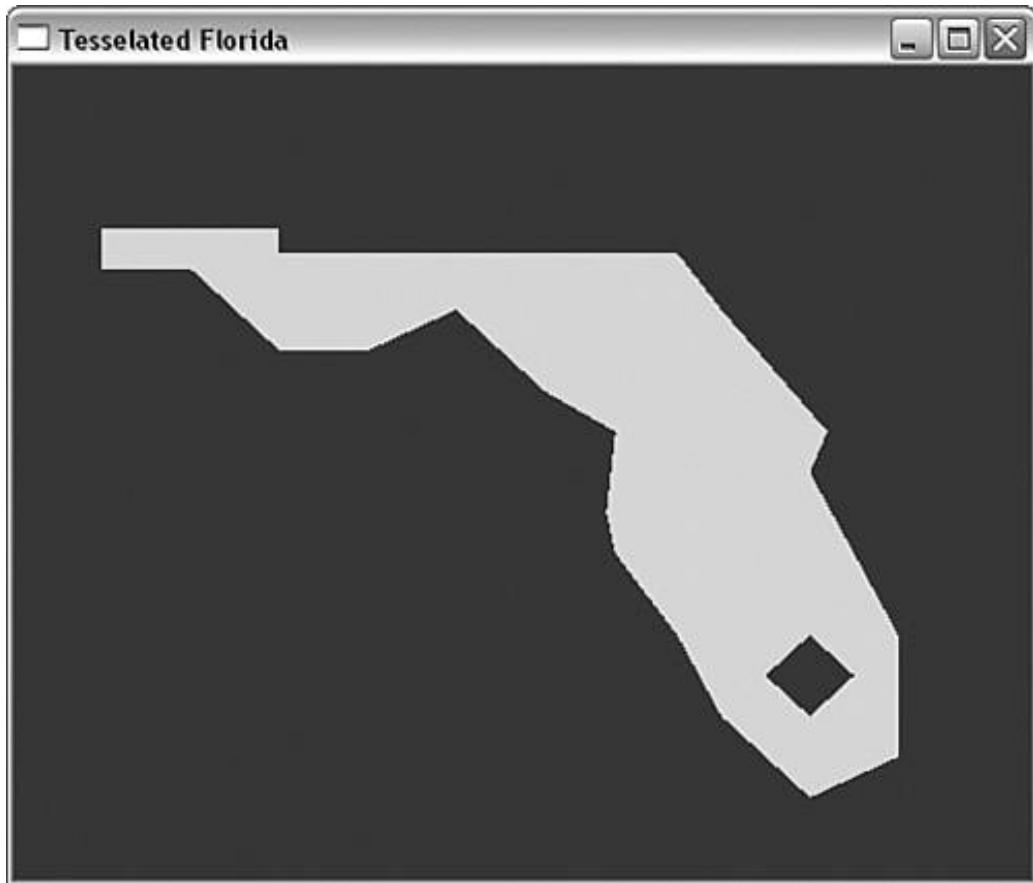
```

```

gluTessVertex(pTess, vCoast[i], vCoast[i]);
gluTessEndContour(pTess);
// Second contour, outline of lake
gluTessBeginContour(pTess);
for(i = 0; i < LAKE_POINTS; i++)
    gluTessVertex(pTess, vLake[i], vLake[i]);
gluTessEndContour(pTess);
// All done with polygon
gluTessEndPolygon(pTess);
// No longer need tessellator object
gluDeleteTess(pTess);
}
break;

```

Figure 10.25. The solid polygon, but with a hole.



This code contained a new function call:

```

// How to count filled and open areas
gluTessProperty(pTess, GLU_TESS_WINDING_RULE, GLU_TESS_WINDING_ODD);

```

This call tells the tessellator how to decide what areas to fill in and which areas to leave empty when there are multiple contours. The value `GLU_TESS_WINDING_ODD` is actually the default, and we could have skipped this function. However, you should understand how the tessellator handles nested contours. By specifying `ODD`, we are saying that any given point inside the polygon is filled in if it is enclosed in an odd number of contours. The area inside the lake (inner contour) is surrounded by two (an even number) contours and is left unfilled. Points outside the lake but inside the state boundary are enclosed by only one contour (an odd number) and are drawn filled.

Summary

The quadrics library makes creating a few simple surfaces (spheres, cylinders, disks, and cones) child's play. Expanding on this concept into more advanced curves and surfaces could have made this chapter the most intimidating in the entire book. As you have seen, however, the concepts behind these curves and surfaces are not very difficult to understand. [Appendix A](#) suggests further reading if you want in-depth mathematical information or tips on creating NURBS-based models.

Other examples from this chapter give you a good starting point for experimenting with NURBS. Adjust the control points and knot sequences to create warped or rumpled surfaces. Also, try some quadratic surfaces and some with higher order than the cubic surfaces. Watch out: One pitfall to avoid as you play with these curves is trying too hard to create one complex surface out of a single NURB. You can find greater power and flexibility if you compose complex surfaces out of several smaller and easy-to-handle NURBS or Bézier surfaces.

Finally, in this chapter, we saw OpenGL's powerful support for automatic polygon tessellation. You learned that you can draw complex surfaces, shapes, and patterns with only a few points that specify the boundaries. You also learned that concave regions and even regions with holes can be broken down into simpler convex primitives using the GLU library's tessellator object.

Reference

glEvalCoord

Purpose: Evaluates 1D and 2D maps that have been previously enabled.

Include File: `<gl.h>`

Variations:

```
void glEvalCoord1d(GLdouble u);
void glEvalCoord1f(GLfloat u);
void glEvalCoord2d(GLdouble u, GLdouble v);
void glEvalCoord2f(GLfloat u, GLfloat v);
void glEvalCoord1dv(const GLdouble *u);
void glEvalCoord1fv(const GLfloat *u);
void glEvalCoord2dv(const GLdouble *u);
void glEvalCoord2fv(const GLfloat *u);
```

Description: This function uses a previously enabled evaluator (set up with [glMap](#)) to produce vertex, color, normal, or texture values based on the parametric *u/v* values. The types of data and function calls simulated are specified by the [glMap1](#) and [glMap2](#) functions.

Parameters:

u, v These parameters specify the *u* and *v* parametric value that is to be evaluated along the curve or surface.

Returns: None.

See Also: [glEvalMesh](#), [glEvalPoint](#), [glMap1](#), [glMap2](#), [glMapGrid](#)

glEvalMesh**Purpose:** Computes a 1D or 2D grid of points or lines.**Include File:** `<gl.h>`**Variations:**

```
void glEvalMesh1(GLenum mode, GLint i1, GLint i2);
void glEvalMesh2(GLenum mode, GLint i1, GLint i2,
    ➔ GLint j1, GLint j2);
```

Description: You use this function with `glMapGrid` to efficiently create a mesh of evenly spaced *u* and *v* domain values. `glEvalMesh` actually evaluates the mesh and produces the points, line segments, or filled polygons.**Parameters:**

mode `GLdouble`: Specifies whether the mesh should be computed as points (`GL_POINT`), lines (`GL_LINE`), or filled meshes for surfaces (`GL_FILL`).

i1, i2 `GLint`: Specifies the first and last integer values for the *u* domain.

j1, j2 `GLint`: Specifies the first and last integer values for the *v* domain.

Returns: None.**See Also:** `glBegin`, `glEvalCoord`, `glEvalPoint`, `glMap1`, `glMap2`, `glMapGrid`**glEvalPoint****Purpose:** Generates and evaluates a single point in a mesh.**Include File:** `<gl.h>`**Variations:**

```
void glEvalPoint1(GLint i);
void glEvalPoint2(GLint i, GLint j);
```

Description: You can use this function in place of `glEvalMesh` to evaluate a domain at a single point. The evaluation produces a single primitive, `GL_POINTS`. The first variation (`glEvalPoint1`) is used for curves, and the second (`glEvalPoint2`) is for surfaces.**Parameters:**

i, j `GLint`: Specifies the *u* and *v* domain parametric values.

Returns: None.**See Also:** `glEvalCoord`, `glEvalMesh`, `glMap1`, `glMap2`, `glMapGrid`

glGetMap**Purpose:** Returns evaluator parameters.**Include File:** `<gl.h>`**Variations:**

```
void glGetMapdv(GLenum target, GLenum query,
  GLdouble *v);
void glGetMapfv(GLenum target, GLenum query,
  GLfloat *v);
void glGetMapiv(GLenum target, GLenum query, GLint
  *v);
```

Description: This function retrieves map settings that were set by the `glMap` functions. See `glMap1` and `glMap2` in this section for explanations of the types of maps.**Parameters:**

`target` `GLenum`: The name of the map; the following maps are defined:
`GL_MAP1_COLOR_4`, `GL_MAP1_INDEX`, `GL_MAP1_NORMAL`,
`GL_MAP1_TEXTURE_COORD_1`, `GL_MAP1_TEXTURE_COORD_2`,
`GL_MAP1_TEXTURE_COORD_3`, `GL_MAP1_TEXTURE_COORD_4`, `GL_MAP1_VERTEX_3`,
`GL_MAP1_VERTEX_4`, `GL_MAP2_COLOR_4`, `GL_MAP2_INDEX`, `GL_MAP2_NORMAL`,
`GL_MAP2_TEXTURE_COORD_1`, `GL_MAP2_TEXTURE_COORD_2`,
`GL_MAP2_TEXTURE_COORD_3`, `GL_MAP2_TEXTURE_COORD_4`, `GL_MAP2_VERTEX_3`, and
`GL_MAP2_VERTEX_4`. See `glMap` in this section for an explanation of these map types.

`query` `GLenum`: Specifies which map parameter to return in `*v`. It may be one of the following values:

- `GL_COEFF`: Returns an array containing the control points for the map. Coordinates are returned in row-major order. 1D maps return order control points, and 2D maps return u-order times the v-order control points.
- `GL_ORDER`: Returns the order of the evaluator function. For 1D maps, this is a single value. For 2D maps, two values are returned (an array) that contain first the u-order and then the v-order.
- `GL_DOMAIN`: Returns the linear parametric mapping parameters. For 1D evaluators, this is the lower and upper `u` value. For 2D maps, it's the lower and upper `u` followed by the lower and upper `v`.

`*v` Pointer to storage that will contain the requested parameter. The data type of this storage should match the function used (double, float, or integer).

Returns: None.

See Also: `glEvalCoord`, `glMap1`, `glMap2`

glMap

Purpose: Defines a 1D or 2D evaluator.

Include File: <gl.h>

Variations:

```
void glMap1d(GLenum target, GLdouble u1, GLdouble u2,
             GLint
             ↪ ustride, GLint uorder,
             ↪ const
             ↪ GLdouble *points);
void glMap1f(GLenum target, GLfloat u1, GLfloat u2,
             GLint
             ↪ ustride, GLint uorder,
             ↪ const
             ↪ GLfloat *points);
void glMap2d(GLenum target, GLdouble u1, GLdouble u2,
             GLint
             ↪ ustride, GLint uorder,
             ↪ GLdouble v1, GLdouble v2,
             ↪ GLint
             ↪ vstride, GLint vorder,
             ↪ const
             ↪ GLdouble *points);
void glMap2f(GLenum target, GLfloat u1, GLfloat u2,
             GLint
             ↪ ustride, GLint uorder,
             ↪ GLfloat
             ↪ v1, GLfloat v2,
             ↪ GLint
             ↪ vstride, GLint vorder,
             ↪ const
             ↪ GLfloat *points);
```

Description: These functions define 1D or 2D evaluators. The `glMap1x` functions are used for 1D evaluators (curves), and the `glMap2x` functions are used for 2D evaluators (surfaces). Evaluators produce vertex or other information (see the `target` parameter) evaluated along one or two dimensions of a parametric range (`u` and `v`).

Parameters:

`target` **GLenum:** Specifies what kinds of values are produced by the evaluator. Valid values for 1D and 2D evaluators are as follows:

`GL_MAP1_VERTEX_3` (or `GL_MAP2_VERTEX_3`): Control points are three floats that represent x, y, and z coordinate values. `glVertex3` commands are generated internally when the map is evaluated.

`GL_MAP1_VERTEX_4` (or `GL_MAP2_VERTEX_4`): Control points are four floats that represent x, y, z, and w coordinate values. `glVertex4` commands are generated internally when the map is evaluated.

`GL_MAP1_INDEX` (or `GL_MAP2_INDEX`): The generated control points are single floats that represent a color index value. `glIndex` commands are generated internally when the map is evaluated. Note: The current color index is not changed as it would be if `glIndex` were called directly.

GL_MAP1_COLOR_4 (or **GL_MAP2_COLOR_4**): The generated control points are four floats that represent red, green, blue, and alpha components. `glColor4` commands are generated internally when the map is evaluated. Note: The current color is not changed as it would be if `glColor4f` were called directly.

GL_MAP1_NORMAL (or **GL_MAP2_NORMAL**): The generated control points are three floats that represent the x, y, and z components of a normal vector. `glNormal` commands are generated internally when the map is evaluated. Note: The current normal is not changed as it would be if `glNormal` were called directly.

GL_MAP1_TEXTURE_COORD_1 (or **GL_MAP2_TEXTURE_COORD_1**): The generated control points are single floats that represent the s texture coordinate. `glTexCoord1` commands are generated internally when the map is evaluated. Note: The current texture coordinates are not changed as they would be if `glTexCoord` were called directly.

GL_MAP1_TEXTURE_COORD_2 (or **GL_MAP2_TEXTURE_COORD_2**): The generated control points are two floats that represent the s and t texture coordinates. `glTexCoord2` commands are generated internally when the map is evaluated. Note: The current texture coordinates are not changed as they would be if `glTexCoord` were called directly.

GL_MAP1_TEXTURE_COORD_3 (or **GL_MAP2_TEXTURE_COORD_3**): The generated control points are three floats that represent the s, t, and r texture coordinates. `glTexCoord3` commands are generated internally when the map is evaluated. Note: The current texture coordinates are not changed as they would be if `glTexCoord` were called directly.

GL_MAP1_TEXTURE_COORD_4 (or **GL_MAP2_TEXTURE_COORD_4**): The generated control points are four floats that represent the s, t, r, and q texture coordinates. `glTexCoord4` commands are generated internally when the map is evaluated. Note: The current texture coordinates are not changed as they would be if `glTexCoord` were called directly.

u1, u2

Specifies the linear mapping of the parametric *u* parameter.

v1, v2

Specifies the linear mapping of the parametric *v* parameter. This parameter is used only for 2D maps.

ustride, vstride

Specifies the number of floats or doubles between control points in the **points* data structure. The coordinates for each point occupy consecutive memory locations, but this parameter allows the points to be spaced as needed to let the data come from an arbitrary data structure.

uorder, vorder

Specifies the number of control points in the u and v direction.

****points***

A memory pointer that points to the control points. It can be a 2D or 3D array or any arbitrary data structure.

Returns:

None.

See Also:

`glBegin`, `glColor`, `glEnable`, `glEvalCoord`, `glEvalMesh`, `glEvalPoint`, `glMapGrid`, `glNormal`, `glTexCoord`, `glVertex`

glMapGrid**Purpose:** Defines a 1D or 2D mapping grid.**Include File:** `<gl.h>`**Variations:**

```
void glMapGrid1d(GLint un, GLdouble u1, GLdouble u2);
void glMapGrid1f(GLint un, GLfloat u1, GLfloat u2);
void glMapGrid2d(GLint un, GLdouble u1, GLdouble u2,
                 GLint vn, GLdouble v1,
                 GLdouble v2);
void glMapGrid2f(GLint un, GLfloat u1, GLfloat u2,
                 GLint vn,
                 GLfloat v1, GLfloat v2);
```

Description: This function establishes a 1D or 2D mapping grid. It is used with `glMap` and `glEvalMesh` to efficiently evaluate a mapping and create a mesh of coordinates.**Parameters:**

`un, vn` **GLint**: Specifies the number of grid subdivisions in the u or v direction.

`u1, u2` Specifies the lower and upper grid domain values in the u direction.

`v1, v2` Specifies the lower and upper grid domain values in the v direction.

Returns: None.**See Also:** `glEvalCoord`, `glEvalMesh`, `glEvalPoint`, `glMap1`, `glMap2`**gluBeginCurve****Purpose:** Begins a NURBS curve definition.**Include File:** `<glu.h>`**Syntax:**

```
void gluBeginCurve(GLUnurbsObj *nObj);
```

Description: You use this function with `gluEndCurve` to delimit a NURBS curve definition.**Parameters:**

`nObj` **GLUnurbsObj** *: Specifies the NURBS object.

Returns: None.**See Also:** `gluEndCurve`**gluBeginSurface**

Purpose: Begins a NURBS surface definition.

Include File: `<glu.h>`

Syntax:

```
void gluBeginSurface(GLUnurbsObj *nObj);
```

Description: You use this function with `gluEndSurface` to delimit a NURBS surface definition.

Parameters:

nObj `GLUnurbsObj *:` Specifies the NURBS object.

Returns: None.

See Also: `gluEndSurface`

gluBeginTrim

Purpose: Begins a NURBS trimming loop definition.

Include File: `<glu.h>`

Syntax:

```
void gluBeginTrim(GLUnurbsObj *nObj);
```

Description: You use this function with `gluEndTrim` to delimit a trimming curve definition. A trimming curve is a curve or set of joined curves defined with `gluNurbsCurve` or `gluPwlCurve`. The `gluBeginTrim` and `gluEndTrim` functions must reside inside the `gluBeginSurface/gluEndSurface` delimiters. When you use trimming, the direction of the curve specifies which portions of the surface are trimmed. Surface area to the left of the traveling direction of the trimming curve is left untrimmed. Thus, clockwise-wound trimming curves eliminate the area inside them, and counterclockwise-wound trimming curves eliminate the area outside them.

Parameters:

nObj `GLUnurbsObj *:` Specifies the NURBS object.

Returns: None.

See Also: `gluEndTrim`

gluCylinder

Purpose: Draws a quadric cylinder.

Include File: `<glu.h>`

Syntax:

```
void gluCylinder(GLUquadricObj *obj, GLdouble
  ↪ baseRadius,
  ↪ GLdouble topRadius,
  ↪ GLdouble height,
  ↪ GLint slices, GLint stacks);
```

Description: This function draws a hollow cylinder with no ends along the z-axis. If `topRadius` or `bottomRadius` is 0, a cone is drawn instead. The cylinder is projected `height` units along the positive z-axis. The `slices` argument controls the number of sides along the cylinder. The `stacks` argument controls the number of segments generated along the z-axis across the cylinder.

Parameters:

`obj` `GLUquadricObj *`: The quadric state information to use for rendering.
`baseRadius` `GLdouble`: The radius of the base ($z = 0$) of the cylinder.
`topRadius` `GLdouble`: The radius of the top ($z = height$) of the cylinder.
`height` `GLdouble`: The height or length of the cylinder along the z-axis.
`slices` `GLint`: The number of sides on the cylinder.
`stacks` `GLint`: The number of segments in the cylinder along the z-axis.

Returns: None.

See Also: `gluDeleteQuadric`, `gluNewQuadric`, `gluQuadricCallback`,
`gluQuadricDrawStyle`, `gluQuadricNormals`, `gluQuadricOrientation`,
`gluQuadricTexture`

gluDeleteNurbsRenderer

Purpose: Destroys a NURBS object.

Include File: `<glu.h>`

Syntax:

```
void gluDeleteNurbsRenderer(GLUnurbsObj *nobj);
```

Description: This function deletes the NURBS object specified and frees any memory associated with it.

Parameters:

`nObj` `GLUnurbsObj *`: Specifies the NURBS object to delete.

Returns: None.

See Also: `gluNewNurbsRenderer`

gluDeleteQuadric

Purpose: Deletes a quadric state object.

Include File: `<glu.h>`

Syntax:

```
void gluDeleteQuadric(GLUquadricObj *obj);
```

Description: This function deletes a quadric state object. After an object has been deleted, it cannot be used for drawing again.

Parameters:

obj `GLUquadricObj *:` The quadric state object to delete.

Returns: None.

See Also: `gluNewQuadric`, `gluQuadricCallback`, `gluQuadricDrawStyle`,
`gluQuadricNormals`, `gluQuadricOrientation`, `gluQuadricTexture`

gluDeleteTess

Purpose: Deletes a tessellator object.

Include File: `<glu.h>`

Syntax:

```
void gluDeleteTess(GLUtesselator *tobj);
```

Description: This function frees all memory associated with a tessellator object.

Parameters:

tobj `GLUtesselator *:` The tessellator object to delete.

Returns: None.

See Also: `gluNewTess`

gluDisk

Purpose: Draws a quadric disk.

Include File: `<glu.h>`

Syntax:

```
void gluDisk(GLUquadricObj *obj, GLdouble innerRadius,
            GLdouble outerRadius,
            ➔ GLint slices, GLint loops);
```

Description: This function draws a disk perpendicular to the z-axis. If *innerRadius* is 0, a solid (filled) circle is drawn instead of a washer. The *slices* argument controls the number of sides on the disk. The *loops* argument controls the number of rings generated out from the z-axis.

Parameters:

obj `GLUquadricObj *`: The quadric state information to use for rendering.

innerRadius `GLdouble`: The inside radius of the disk.

outerRadius `GLdouble`: The outside radius of the disk.

slices `GLint`: The number of sides on the cylinder.

loops `GLint`: The number of rings out from the z-axis.

Returns: None.

See Also: `gluDeleteQuadric`, `gluNewQuadric`, `gluQuadricCallback`,
`gluQuadricDrawStyle`, `gluQuadricNormals`, `gluQuadricOrientation`,
`gluQuadricTexture`

gluEndCurve

Purpose: Ends a NURBS curve definition.

Include File: `<glu.h>`

Syntax:

```
void gluEndCurve(GLUnurbsObj *nobj);
```

Description: You use this function with `gluBeginCurve` to delimit a NURBS curve definition.

Parameters:

nObj `GLUnurbsObj *`: Specifies the NURBS object.

Returns: None.

See Also: `gluBeginCurve`

gluEndSurface

Purpose: Ends a NURBS surface definition.

Include File: `<glu.h>`

Syntax:

```
void gluEndSurface(GLUnurbsObj *nObj);
```

Description: You use this function with `gluBeginSurface` to delimit a NURBS surface definition.

Parameters:

nObj `GLUnurbsObj *:` Specifies the NURBS object.

Returns: None.

See Also: `gluBeginSurface`

gluEndTrim

Purpose: Ends a NURBS trimming loop definition.

Include File: `<glu.h>`

Syntax:

```
void gluEndTrim(GLUnurbsObj *nObj);
```

Description: You use this function with `gluBeginTrim` to mark the end of a NURBS trimming loop. See `gluBeginTrim` for more information on trimming loops.

Parameters:

nObj `GLUnurbsObj *:` Specifies the NURBS object.

Returns: None.

See Also: `gluBeginTrim`

gluGetNurbsProperty

Purpose: Retrieves a NURBS property.

Include File: `<gl.h>`

Syntax:

```
void gluGetNurbsProperty(GLUnurbsObj *nObj, GLenum
    ↪ property, GLfloat *value);
```

Description: This function retrieves the NURBS property specified for a particular NURBS object. See `gluNurbsProperty` for an explanation of the various properties.

Parameters:

nObj `GLUnurbsObj *:` Specifies the NURBS object.

property `GLenum *:` The NURBS property to be retrieved. Valid properties are `GLU_SAMPLING_TOLERANCE`, `GLU_DISPLAY_MODE`, `GLU_CULLING`, `GLU_AUTO_LOAD_MATRIX`, `GLU_PARAMETRIC_TOLERANCE`, `GLU_SAMPLING_METHOD`, `GLU_U_STEP`, and `GLU_V_STEP`. See the `gluNurbsProperty` function for details on these properties.

value `GLfloat *:` A pointer to the location into which the value of the named property is to be copied.

Returns: None.

See Also: [gluNewNurbsRenderer](#), [gluNurbsProperty](#)

gluLoadSamplingMatrices

Purpose: Loads NURBS sampling and culling matrices.

Include File: `<gl.h>`

Syntax:

```
void gluLoadSamplingMatrices(GLUnurbsObj *nObj,
➥ const GLfloat modelMatrix[16],
➥                                     const GLfloat
➥ projMatrix[16],
➥                                     const GLint
➥ viewport[4]);
```

Description: You use this function to recompute the sampling and culling matrices for a NURBS surface. The sampling matrix enables you to determine how finely the surface must be tessellated to satisfy the sampling tolerance. The culling matrix enables you to determine whether the surface should be culled before rendering. Usually, this function does not need to be called, unless the [GLU_AUTO_LOAD_MATRIX](#) property is turned off. This might be the case when using selection and feedback modes.

Parameters:

`nObj` `GLUnurbsObj *:` Specifies the NURBS object.

`modelMatrix` `GLfloat[16]:` Specifies the modelview matrix.

`projMatrix` `GLfloat[16]:` Specifies the projection matrix.

`viewport` `GLint[4]:` Specifies a viewport.

Returns: None.

See Also: [gluNewNurbsRenderer](#), [gluNurbsProperty](#)

gluNewNurbsRenderer

Purpose: Creates a NURBS object.

Include File: `<glu.h>`

Syntax:

```
GLUnurbsObj* gluNewNurbsRenderer(void);
```

Description: This function creates a NURBS rendering object. This object is used to control the behavior and characteristics of NURBS curves and surfaces. The functions that allow the NURBS properties to be set all require this pointer. You must delete this object with [gluDeleteNurbsRenderer](#) when you are finished rendering your NURBS.

Returns: A pointer to a new NURBS object. This object is used when you call the rendering and control functions.

See Also: [gluDeleteNurbsRenderer](#)

gluNewQuadric

Purpose: Creates a new quadric state object.

Include File: [<glu.h>](#)

Syntax:

```
GLUquadricObj *gluNewQuadric(void);
```

Description: This function creates a new opaque quadric state object to be used for drawing. The quadric state object contains specifications that determine how subsequent images will be drawn.

Parameters: None.

Returns: `GLUquadricObj *: NULL` if no memory is available; otherwise, a valid quadric state object pointer.

See Also: [gluDeleteQuadric](#), [gluQuadricCallback](#), [gluQuadricDrawStyle](#), [gluQuadricNormals](#), [gluQuadricOrientation](#), [gluQuadricTexture](#)

gluNewTess

Purpose: Creates a tessellator object.

Include File: [<glu.h>](#)

Syntax:

```
GLUTriangulatorObj *gluNewTess(void);
```

Description: This function creates a tessellator object.

Parameters: None.

Returns: `GLUTriangulatorObj *: The new tessellator object.`

See Also: [gluDeleteTess](#)

gluNurbsCallback

Purpose: Defines a callback for a NURBS function.

Include File: [<glu.h>](#)

Syntax:

```
void gluNurbsCallback(GLUnurbsObj *nObj, GLenum
➥ which, void(*fn)( ));
```

Description: This function sets a NURBS callback function. The only supported callback prior to GLU version 1.3 is `GLU_ERROR`. When an error occurs, this function is called with an argument of type `GLenum`. One of 37 NURBS errors can be specified by the constants `GLU_NURBS_ERROR1` through `GLU_NURBS_ERROR37`. You can retrieve a character string definition of the error with the function `gluErrorString`. These error codes are listed in [Table 10.2](#). For GLU version 1.3 and later, `GLU_ERROR` has been superceded by `GLU_NURBS_ERROR` and a number of other callbacks listed under "Parameters."

Parameters:

`nObj` `GLUnurbsObj *:` Specifies the NURBS object.

`which` `GLenum:` Specifies the callback being defined. Prior to GLU version 1.3, the only valid value was `GLU_ERROR`. For GLU version 1.3 and later (currently, only MacOS X) `GLU_ERROR` has been superceded by `GLU_NURBS_ERROR` and any of the following other callbacks: `GLU_NURBS_BEGIN`, `GLU_NURBS_VERTEX`, `GLU_NURBS_NORMAL`, `GLU_NURBS_COLOR`, `GLU_NURBS_TEXTURE_COORD`, `GLU_NURBS_END`, `GLU_NURBS_BEGIN_DATA`, `GLU_NURBS_VERTEX_DATA`, `GLU_NURBS_NORMAL_DATA`, `GLU_NURBS_COLOR_DATA`, `GLU_NURBS_TEXTURE_COORD_DATA`, and `GLU_NURBS_END_DATA`.

`fn` `void *():` Specifies the function that should be called for the callback. The following prototypes are used for the different callbacks:

```
GLU_NURBS_BEGIN: void *(GLenum type);
GLU_NURBS_BEGIN_DATA: void *(GLenum type, void
➥ *userData)
GLU_NURBS_VERTEX: void *(GLfloat *vertex);
GLU_NURBS_VERTEX_DATA: void *(GLfloat *vertex, void
➥ *userData)
GLU_NURBS_NORMAL: void *(GLfloat *normal);
GLU_NURBS_NORMAL_DATA: void *(GLfloat *normal,
➥ void *userData);
GLU_NURBS_COLOR: void *(GLfloat *color);
GLU_NURBS_COLOR_DATA: void *(GLfloat *color, void
➥ *userData);
GLU_NURBS_TEXTURE_COORD: void *(GLfloat *texCoord);
GLU_NURBS_TEXTURE_COORD_DATA: void *(GLfloat
➥ *texCoord, void *userData);
GLU_NURBS_END: void *(void);
GLU_NURBS_END_DATA: void *(void userData);
GLU_NURBS_ERROR: void *(GLenum error);
```

Returns: None.

See Also: `gluErrorString`

Table 10.2. NURBS Error Codes

Error Code	Definition
GLU_NURBS_ERROR1	Spline order unsupported.
GLU_NURBS_ERROR2	Too few knots.
GLU_NURBS_ERROR3	Valid knot range is empty.
GLU_NURBS_ERROR4	Decreasing knot sequence knot.
GLU_NURBS_ERROR5	Knot multiplicity greater than order of spline.
GLU_NURBS_ERROR6	<i>endcurve</i> must follow <i>bgncurve</i> .
GLU_NURBS_ERROR7	<i>bgncurve</i> must precede <i>endcurve</i> .
GLU_NURBS_ERROR8	Missing or extra geometric data.
GLU_NURBS_ERROR9	Can't draw <i>pwlcurves</i> .
GLU_NURBS_ERROR10	Missing or extra domain data.
GLU_NURBS_ERROR11	Missing or extra domain data.
GLU_NURBS_ERROR12	<i>endtrim</i> must precede <i>endsurface</i> .
GLU_NURBS_ERROR13	<i>bgnsurface</i> must precede <i>endsurface</i> .
GLU_NURBS_ERROR14	Curve of improper type passed as trim curve.
GLU_NURBS_ERROR15	<i>bgnsurface</i> must precede <i>bgntrim</i> .
GLU_NURBS_ERROR16	<i>endtrim</i> must follow <i>bgntrim</i> .
GLU_NURBS_ERROR17	<i>bgntrim</i> must precede <i>endtrim</i> .
GLU_NURBS_ERROR18	Invalid or missing trim curve.
GLU_NURBS_ERROR19	<i>bgntrim</i> must precede <i>pwlcurve</i> .
GLU_NURBS_ERROR20	<i>pwlcurve</i> referenced twice.
GLU_NURBS_ERROR21	<i>pwlcurve</i> and <i>nurbscurve</i> mixed.
GLU_NURBS_ERROR22	Improper usage of trim data type.
GLU_NURBS_ERROR23	<i>nurbscurve</i> referenced twice.
GLU_NURBS_ERROR24	<i>nurbscurve</i> and <i>pwlcurve</i> mixed.
GLU_NURBS_ERROR25	<i>nurbssurface</i> referenced twice.
GLU_NURBS_ERROR26	Invalid property.
GLU_NURBS_ERROR27	<i>endsurface</i> must follow <i>bgnsurface</i> .
GLU_NURBS_ERROR28	Intersecting or misoriented trim curves.
GLU_NURBS_ERROR29	Intersecting trim curves.
GLU_NURBS_ERROR30	Unused.
GLU_NURBS_ERROR31	Unconnected trim curves.
GLU_NURBS_ERROR32	Unknown knot error.
GLU_NURBS_ERROR33	Negative vertex count encountered.
GLU_NURBS_ERROR34	Negative byte-stride encountered.

GLU_NURBS_ERROR35 Unknown type descriptor.
GLU_NURBS_ERROR36 Null control point reference.
GLU_NURBS_ERROR37 Duplicate point on *pwlcurve*.

gluNurbsCurve

Purpose: Defines the shape of a NURBS curve.

Include File: `<glu.h>`

Syntax:

```
void gluNurbsCurve(GLUnurbsObj *nObj, GLint nknots
→ , GLfloat *knot,
→ , GLint stride, GLfloat *ctlArray
→ , GLint order, GLenum type);
```

Description: This function defines the shape of a NURBS curve. The definition of this curve must be delimited by `gluBeginCurve` and `gluEndCurve`.

Parameters:

nObj `GLUnurbsObj *:` A pointer to the NURBS object (created with `gluNewNurbsRenderer`).

nknots `GLInt:` The number of knots in **knots*. This is the number of control points plus *order*.

knot `GLfloat *:` An array of knot values in nondescending order.

stride `GLInt:` The offset, as a number of single-precision floating-point values, between control points.

ctlArray `GLfloat *:` A pointer to an array or data structure containing the control points for the NURBS surface.

order `GLInt:` The order of the NURBS surface. The order is 1 more than the degree.

type `GLenum:` The type of surface. It can be any of the two-dimensional evaluator types: `GL_MAP2_VERTEX_3`, `GL_MAP2_VERTEX_4`, `GL_MAP2_INDEX`, `GL_MAP2_COLOR_4`, `GL_MAP2_NORMAL`, `GL_MAP2_TEXTURE_COORD_1`, `GL_MAP2_TEXTURE_COORD_2`, `GL_MAP2_TEXTURE_COORD_3`, and `GL_MAP2_TEXTURE_COORD_4`.

Returns: None.

See Also: `gluBeginCurve`, `gluEndCurve`, `gluNurbsSurface`

gluNurbsProperty

Purpose: Sets a NURBS property.

Include File: `<glu.h>`

Syntax:

```
void gluNurbsProperty(GLUnurbsObj *nObj, GLenum
➥ property, GLfloat value);
```

Description: This function sets the properties of the NURBS object. Valid properties are as follows:

GLU_SAMPLING_TOLERANCE: Sets the maximum length in pixels to use when using the **GLU_PATH_LENGTH** sampling method. The default is 50.0.

GLU_DISPLAY_MODE: Defines how the NURBS surface is rendered. The **value** parameter can be **GLU_FILL** to use filled and shaded polygons, **GLU_OUTLINE_POLYGON** to draw just the outlines of the polygons (after tessellation), or **GLU_OUTLINE_PATCH** to draw just the outlines of user-defined patches and trim curves. The default is **GLU_FILL**.

GLU_CULLING: Interprets the **value** parameter as a Boolean value that indicates whether the NURBS curve should be discarded if its control points are outside the viewport.

GLU_PARAMETRIC_TOLERANCE: Sets the maximum pixel distance used when the sampling method is set to **GLU_PARAMETRIC_ERROR**. The default is 0.5. This property was introduced in GLU version 1.1.

GLU_SAMPLING_METHOD: Specifies how to tessellate the NURBS surface. This property was introduced in GLU version 1.1. The following values are valid:

GLU_PATH_LENGTH specifies that surfaces rendered with the maximum pixel length of the edges of the tessellation polygons are not greater than the value specified by **GLU_SAMPLING_TOLERANCE**. **GLU_PARAMETRIC_ERROR** specifies that the surface is rendered with the value of **GLU_PARAMETRIC_TOLERANCE** designating the maximum distance, in pixels, between the tessellation polygons and the surfaces they approximate. **GLU_DOMAIN_DISTANCE** specifies, in parametric coordinates, how many sample points per unit length to take in the **u** and **v** dimensions. The default is **GLU_PATH_LENGTH**.

GLU_U_STEP: Sets the number of sample points per unit length taken along the **u** dimension in parametric coordinates. This value is used when **GLU_SAMPLING_METHOD** is set to **GLU_DOMAIN_DISTANCE**. The default is 100. This property was introduced in GLU version 1.1.

GLU_V_STEP: Sets the number of sample points per unit length taken along the **v** dimension in parametric coordinates. This value is used when **GLU_SAMPLING_METHOD** is set to **GLU_DOMAIN_DISTANCE**. The default is 100. This property was introduced in GLU version 1.1.

GLU_AUTO_LOAD_MATRIX: Interprets the **value** parameter as a Boolean value. When it is set to **GL_TRUE**, it causes the NURBS code to download the projection matrix, the modelview matrix, and the viewport from the OpenGL server to compute sampling and culling matrices for each NURBS curve. Sampling and culling matrices are required to determine the tessellation of a NURBS surface into line segments or polygons and to cull a NURBS surface if it lies outside the viewport. If this mode is set to **GL_FALSE**, the user needs to provide these matrices and a viewport for the NURBS renderer to use in constructing sampling

and culling matrices. This can be done with the `gluLoadSamplingMatrices` function. The default value for this mode is `GL_TRUE`. Changing this mode does not affect the sampling and culling matrices until `gluLoadSamplingMatrices` is called.

Parameters:

`nObj` `GLUnurbsObj *`: The NURB object that is having a property modified. (It is created by calling `glNewNurbsRenderer`.)

`property` `GLenum`: The property to be set or modified. It may be any of the following values: `GLU_SAMPLING_TOLERANCE`, `GLU_DISPLAY_MODE`, `GLU_CULLING`, `GLU_AUTO_LOAD_MATRIX`, `GLU_PARAMETRIC_TOLERANCE`, `GLU_SAMPLING_METHOD`, `GLU_U_STEP`, and `GLU_V_STEP`.

`value` `GLfloat`: The value to which the indicated property is being set.

Returns: None.

See Also: `gluGetNurbsProperty`, `gluGetString`, `gluLoadSamplingMatrices`, `gluNewNurbsRenderer`, `gluNewNurbsRenderer`, `gluNurbsCurve`, `gluPwlCurve`

gluNurbsSurface

Purpose: Defines the shape of a NURBS surface.

Include File: `<glu.h>`

Syntax:

```
void gluNurbsSurface(GLUnurbsObj *nObj, GLint
  ↪ uknotCount, GLfloat *uknot,
  ↪           GLint vknotCount, GLfloat
  ↪ *vknot, GLint uStride,
  ↪           GLint vStride, GLfloat
  ↪ *ctlArray, GLint uorder,
  ↪           GLint vorder, GLenum type);
```

Description: This function defines the shape of a NURBS surface. It must be delimited by `gluBeginSurface` and `gluEndSurface`. The shape of the surface is defined before any trimming takes place. You can trim a NURBS surface by using `gluBeginTrim`/`gluEndTrim` and `gluNurbsCurve` or `gluPwlCurve` to do the trimming.

Parameters:

`nObj` `GLUnurbsObj *`: A pointer to the NURBS object. (It is created with `gluNewNurbsRenderer`.)

`uknotCount` `GLint`: The number of knots in the parametric u direction.

`uknot` `GLfloat *`: An array of knot values that represent the knots in the u direction. These values must be nondescending. The length of the array is specified in `uknotCount`.

`vknotCount` `GLint`: The number of knots in the parametric v direction.

<i>vknot</i>	<code>GLfloat*</code> : An array of knot values that represent the knots in the v direction. These values must be nondescending. The length of the array is specified in <i>vknotCount</i> .
<i>uStride</i>	<code>GLint</code> : The offset, as a number of single-precision, floating-point values, between successive control points in the parametric u direction in <i>ctlArray</i> .
<i>vStride</i>	<code>GLint</code> : The offset, as a number of single-precision, floating-point values, between successive control points in the parametric v direction in <i>ctlArray</i> .
<i>ctlArray</i>	<code>GLfloat *</code> : A pointer to an array containing the control points for the NURBS surface. The offsets between successive control points in the parametric u and v directions are given by <i>uStride</i> and <i>vStride</i> .
<i>uorder</i>	<code>GLint</code> : The order of the NURBS surface in the parametric u direction. The order is 1 more than the degree; hence, a surface that is cubic in u has a u order of 4.
<i>vorder</i>	<code>GLint</code> : The order of the NURBS surface in the parametric v direction. The order is 1 more than the degree; hence, a surface that is cubic in v has a v order of 4.
<i>type</i>	<code>GLenum</code> : The type of surface. It can be any of the 2D evaluator types: <code>GL_MAP2_VERTEX_3, GL_MAP2_VERTEX_4, GL_MAP2_INDEX, GL_MAP2_COLOR_4, GL_MAP2_NORMAL, GL_MAP2_TEXTURE_COORD_1, GL_MAP2_TEXTURE_COORD_2, GL_MAP2_TEXTURE_COORD_3, and GL_MAP2_TEXTURE_COORD_4.</code>
Returns:	None.
See Also:	<code>gluBeginSurface</code> , <code>gluBeginTrim</code>

gluPartialDisk

Purpose: Draws a partial quadric disk.

Include File: `<glu.h>`

Syntax:

```
void gluPartialDisk(GLUquadricObj *obj, GLdouble
  ↪ innerRadius,
               GLdouble outerRadius,
  ↪ GLint slices, GLint loops,
               GLdouble startAngle,
  ↪ GLdouble sweepAngle);
```

Description: This function draws a partial disk perpendicular to the z-axis. If *innerRadius* is 0, a solid (filled) circle is drawn instead of a washer. The *slices* argument controls the number of sides on the disk. The *loops* argument controls the number of rings generated out from the z-axis. The *startAngle* argument specifies the starting angle of the disk with 0° at the top of the disk and 90° at the right of the disk. The *sweepAngle* argument specifies the portion of the disk in degrees.

Parameters:

<i>obj</i>	<code>GLUquadricObj *</code> : The quadric state information to use for rendering.
<i>innerRadius</i>	<code>GLdouble</code> : The inside radius of the disk.
<i>outerRadius</i>	<code>GLdouble</code> : The outside radius of the disk.

slices **GLint**: The number of sides on the cylinder.

loops **GLint**: The number of rings out from the z-axis.

startAngle **GLdouble**: The start angle of the partial disk.

sweepAngle **GLdouble**: The angular size of the partial disk.

Returns: None.

See Also: **gluDeleteQuadric**, **gluNewQuadric**, **gluQuadricCallback**,
gluQuadricDrawStyle, **gluQuadricNormals**, **gluQuadricOrientation**,
gluQuadricTexture

gluPwlCurve

Purpose: Specifies a piecewise NURBS trimming curve.

Include File: `<glu.h>`

Syntax:

```
void gluPwlCurve(GLUnurbsObj *nObj, GLint count,
  ↪ GLfloat *array, GLint stride, GLenum type);
```

Description: This function defines a piecewise linear trimming curve for a NURBS surface. The array of points is in terms of the parametric u and v coordinate space. This space is a unit square exactly 1 unit in length along both axes. Clockwise-wound trimming curves eliminate the enclosed area; counterclockwise trimming curves discard the exterior area. Typically, a trimming region is first established around the entire surface area that trims away all points not on the surface. Then smaller trimming areas wound clockwise are placed within it to cut away sections of the curve. Trimming curves can be piecewise. This means one or more calls to **gluPwlCurve** or **gluNurbsCurve** can be called to define a trimming region as long as they share endpoints and define a close region in u/v space.

Parameters:

nObj **GLUnurbsObj** *: Specifies the NURBS object being trimmed.

count **GLint**: Specifies the number of points on the curve listed in **array*.

array **GLfloat** *: Specifies the array of boundary points for this curve.

stride **GLint**: Specifies the offset between points on the curve.

type **GLenum**: Specifies the type of curve. It can be **GLU_MAP1_TRIM_2**, used when the trimming curve is specified in terms of u and v coordinates, or **GLU_MAP1_TRIM_3**, used when a w (scaling) coordinate is also specified.

Returns: None.

gluQuadricCallback

Purpose: Defines a quadric callback function.

Include File: `<glu.h>`**Syntax:**

```
void gluQuadricCallback(GLUquadricObj *obj, GLenum
➥ which, void (*fn)());
```

Description: This function defines callback functions to be used when drawing quadric shapes. At present, the only defined callback function is `GLU_ERROR`, which is called whenever an OpenGL or GLU error occurs.

Parameters:

`obj` `GLUquadricObj *:` The quadric state information to use for rendering.
`which` `GLenum:` The callback function to define. It must be `GLU_ERROR`.
`fn` `void (*)():` The callback function (receives one `GLenum` containing the error).

Returns: None**See Also:** `gluDeleteQuadric`, `gluNewQuadric`, `gluQuadricDrawStyle`,
`gluQuadricNormals`, `gluQuadricOrientation`, `gluQuadricTexture`

gluQuadricDrawStyle

Purpose: Sets the drawing style of a quadric state object.**Include File:** `<glu.h>`**Syntax:**

```
void gluQuadricDrawStyle(GLUquadricObj *obj,
➥ GLenum drawStyle);
```

Description: This function selects a drawing style for all quadric shapes.**Parameters:**`obj` `GLUquadricObj *:` The quadric state information to use for rendering.`drawStyle` `GLenum:` The drawing style. Valid styles are as follows:

`GLU_FILL`: Quadrics are drawn filled, using polygon and strip primitives.

`GLU_LINE`: Quadrics are drawn "wireframe," using line primitives.

`GLU_SILHOUETTE`: Quadrics are drawn using line primitives; only the outside edges are drawn.

`GLU_POINT`: Quadrics are drawn using point primitives.

Returns: None.**See Also:** `gluDeleteQuadric`, `gluNewQuadric`, `gluQuadricCallback`, `gluQuadricNormals`,
`gluQuadricOrientation`, `gluQuadricTexture`

gluQuadricNormals

Purpose: Sets the type of lighting normals used for quadric objects.

Include File: `<glu.h>`

Syntax:

```
void gluQuadricNormals(GLUquadricObj *obj, GLenum  
→   normals);
```

Description: This function sets the type of lighting normals that are generated when drawing shapes using the specified quadric state object.

Parameters:

obj `GLUquadricObj *`: The quadric state information to use for rendering.

normals `GLenum`: The type of normal to generate. Valid types are as follows:

`GLU_NONE`: No lighting normals are generated.

`GLU_FLAT`: Lighting normals are generated for each polygon to generate a faceted appearance.

`GLU_SMOOTH`: Lighting normals are generated for each vertex to generate a smooth appearance.

Returns: None.

See Also: `gluDeleteQuadric`, `gluNewQuadric`, `gluQuadricCallback`, `gluQuadricDrawStyle`, `gluQuadricOrientation`, `gluQuadricTexture`

gluQuadricOrientation

Purpose: Sets the orientation of lighting normals for quadric objects.

Include File: `<glu.h>`

Syntax:

```
void gluQuadricOrientation(GLUquadricObj *obj,  
→   GLenum orientation);
```

Description: This function sets the direction of lighting normals for hollow objects. The orientation parameter can be `GLU_OUTSIDE` to point lighting normals outward or `GLU_INSIDE` to point them inward.

Parameters:

obj `GLUquadricObj *`: The quadric state information to use for rendering.

orientation GLenum: The orientation of lighting normals, `GLU_OUTSIDE` or `GLU_INSIDE`. The default is `GLU_OUTSIDE`.

Returns: None.

See Also: `gluDeleteQuadric`, `gluNewQuadric`, `gluQuadricCallback`,
`gluQuadricDrawStyle`, `gluQuadricNormals`, `gluQuadricTexture`

gluQuadricTexture

Purpose: Enables or disables texture coordinate generation for texture-mapping images onto quadrics.

Include File: `<glu.h>`

Syntax:

```
void gluQuadricTexture(GLUquadricObj *obj,
➥ GLboolean textureCoords);
```

Description: This function controls whether texture coordinates are generated for quadric shapes.

Parameters:

obj GLUquadricObj *: The quadric state information to use for rendering.

textureCoords GLboolean: `GL_TRUE` if texture coordinates should be generated; `GL_FALSE` otherwise.

Returns: None.

See Also: `gluDeleteQuadric`, `gluNewQuadric`, `gluQuadricCallback`,
`gluQuadricDrawStyle`, `gluQuadricNormals`, `gluQuadricOrientation`

gluSphere

Purpose: Draws a quadric sphere.

Include File: `<glu.h>`

Syntax:

```
void gluSphere(GLUquadricObj *obj, GLdouble radius
➥ , GLint slices, GLint stacks);
```

Description: This function draws a hollow sphere centered at the origin. The *slices* argument controls the number of lines of longitude on the sphere. The *stacks* argument controls the number of lines of latitude on the sphere.

Parameters:

obj GLUquadricObj *: The quadric state information to use for rendering.

radius GLdouble: The radius of the sphere.

slices *GLint*: The number of lines of longitude on the sphere.

stacks *GLint*: The number of lines of latitude on the sphere.

Returns: None.

See Also: *gluDeleteQuadric*, *gluNewQuadric*, *gluQuadricCallback*, *gluQuadricDrawStyle*, *gluQuadricNormals*, *gluQuadricOrientation*, *gluQuadricTexture*

gluTessBeginContour

Purpose: Specifies a new contour or hole in a complex polygon.

Include File: `<glu.h>`

Syntax:

```
void gluTessBeginContour(GLUtesselator *tobj);
```

Description: This function specifies a new contour or hole in a complex polygon.

Parameters:

tobj *GLUtesselator* *: The tessellator object to use for the polygon.

Returns: None.

See Also: *gluTessBeginPolygon*, *gluTessEndPolygon*, *gluTessEndContour*, *gluTessVertex*

gluTessBeginPolygon

Purpose: Starts tessellation of a complex polygon.

Include File: `<glu.h>`

Syntax:

```
void gluTessBeginPolygon(GLUtesselator *tobj,
    GLvoid *data);
```

Description: This function starts tessellation of a complex polygon.

Parameters:

tobj *GLUtesselator* *: The tessellator object to use for the polygon.

data *GLvoid* *: The data that is passed to *GLU_TESS_*_DATA* callbacks.

Returns: None.

See Also: *gluTessEndPolygon*, *gluTessBeginContour*, *gluTessEndContour*, *gluTessVertex*

gluTessCallback**Purpose:** Specifies a callback function for tessellation.**Include File:** `<glu.h>`**Syntax:**

```
void gluTessCallback(GLUTesselator *tobj, GLenum
➥ which, void (*fn)());
```

Description: This function specifies a callback function for various tessellation functions. Callback functions do not replace or change the tessellator performance. Rather, they provide the means to add information to the tessellated output (such as color or texture coordinates).

Parameters:

tobj `GLUTesselator *:` The tessellator object to use for the polygon.

which `GLenum:` The callback function to define. Valid functions appear in [Table 10.3](#)

fn `void (*)():` The function to call.

Returns: None.

Table 10.3. Tessellator Callback Identifiers

Constant	Description
<code>GLU_TESS_BEGIN</code>	Specifies a function that is called to begin a <code>GL_TRIANGLES</code> , <code>GL_TRIANGLE_STRIP</code> , or <code>GL_TRIANGLE_FAN</code> primitive. The function must accept a single <code>GLenum</code> parameter that specifies the primitive to be rendered and is usually set to <code>glBegin</code> .
<code>GLU_TESS_BEGIN_DATA</code>	Like <code>GLU_TESS_BEGIN</code> , specifies a function [GLU 1.2] that is called to begin a <code>GL_TRIANGLES</code> , <code>GL_TRIANGLE_STRIP</code> , or <code>GL_TRIANGLE_FAN</code> primitive. The function must accept a <code>GLenum</code> parameter that specifies the primitive to be rendered and a <code>GLvoid</code> pointer from the call to <code>gluTessBeginPolygon</code> .
<code>GLU_TESS_COMBINE</code>	Specifies a function that is called when [GLU 1.2] vertices in the polygon are coincident; that is, they are equal.
<code>GLU_TESS_COMBINE_DATA</code>	Like <code>GLU_TESS_COMBINE</code> , specifies a function [GLU 1.2] that is called when vertices in the polygon are coincident. The function also receives a pointer to the user data from <code>gluTessBeginPolygon</code> .
<code>GLU_TESS_EDGE_FLAG</code>	Specifies a function that marks whether succeeding <code>GLU_TESS_VERTEX</code> callbacks refer to original or generated vertices. The function must accept a single <code>GLboolean</code> argument that is <code>GL_TRUE</code> for original and <code>GL_FALSE</code> for generated vertices.

<code>GLU_TESS_EDGE_FLAG_DATA</code>	Specifies a function similar to the <code>GLU_TESS_EDGE_FLAG</code> , with the exception that a void pointer to user data is also accepted.
<code>GLU_TESS_END</code>	Specifies a function that marks the end of a drawing primitive, usually <code>glEnd</code> . It takes no arguments.
<code>GLU_TESS_END_DATA</code>	Specifies a function similar to <code>GLU_TESS_END</code> , with the addition of a void pointer to user data.
<code>GLU_TESS_ERROR</code>	Specifies a function that is called when an error occurs. It must take a single argument of type <code>GLenum</code> .
<code>GLU_TESS_VERTEX</code>	Specifies a function that is called before every vertex is sent, usually with <code>glVertex3dv</code> . The function receives a copy of the third argument to <code>gluTessVertex</code> .
<code>GLU_TESS_VERTEX_DATA</code>	Like <code>GLU_TESS_VERTEX</code> , specifies a function [GLU 1.2] that is called before every vertex is sent. The function also receives a copy of the second argument to <code>gluTessBeginPolygon</code> .

gluTessEndContour

Purpose: Ends a contour in a complex polygon.

Include File: `<glu.h>`

Syntax:

```
void gluTessEndContour(GLUtesselator *tobj);
```

Description: This function ends the current polygon contour.

Parameters:

`tobj` `GLUtesselator *:` The tessellator object to use for the polygon.

Returns: None.

See Also: `gluTessBeginPolygon`, `gluTessBeginContour`, `gluTessEndPolygon`, `gluTessVertex`

gluTessEndPolygon

Purpose: Ends tessellation of a complex polygon and renders it.

Include File: `<glu.h>`

Syntax:

```
void gluTessEndPolygon(GLUtesselator *tobj);
```

Description: This function ends tessellation of a complex polygon and renders the final result.

Parameters:

tobj `GLUtesselator *`: The tessellator object to use for the polygon.

Returns: None.

See Also: `gluTessBeginPolygon`, `gluTessBeginContour`, `gluTessEndContour`, `gluTessVertex`

gluTessProperty

Purpose: Sets a tessellator property value.

Include File: `<glu.h>`

Syntax:

```
void gluTessProperty(GLUtesselator *tobj, GLenum
    ↪ which, GLdouble value);
```

Description: This function sets a tessellator property value.

Parameters:

tobj `GLUtesselator *`: The tessellator object to change.

which `GLenum`: The property to change: `GLU_TESS_BOUNDARY_ONLY`, `GLU_TESS_TOLERANCE`, or `GLU_TESS_WINDING_RULE`.

value `GLdouble`: The value for the property.

For `GLU_TESS_BOUNDARY_ONLY`, the value can be `GL_TRUE` or `GL_FALSE`. If `GL_TRUE`, only the boundary of the polygon is displayed (no holes).

For `GLU_TESS_TOLERANCE`, the value is the coordinate tolerance for vertices in the polygon.

For `GLU_TESS_WINDING_RULE`, the value is one of `GLU_TESS_WINDING_NONZERO`, `GLU_TESS_WINDING_POSITIVE`, `GLU_TESS_WINDING_NEGATIVE`, `GLU_TESS_WINDING_ABS_GEQ_TWO`, or `GLU_TESS_WINDING_ODD`.

Returns: None.

See Also: `gluTessBeginPolygon`, `gluTessEndPolygon`, `gluTessBeginContour`, `gluTessEndContour`, `gluTessCallback`, `gluTessVertex`, `gluNewTess`, `gluDeleteTess`

gluTessVertex

Purpose: Adds a vertex to the current polygon path.

Include File: `<glu.h>`

Syntax:

```
void gluTessVertex(GLUtesselator *tobj, GLdouble
→ v[3], void *data);
```

Description: This function adds a vertex to the current tessellator path. The data argument is passed through to the `GL_VERTEX` callback function.

Parameters:

`tobj` `GLUtesselator *:` The tessellator object to use for the polygon.
`v` `GLdouble[3]:` The 3D vertex.
`data` `void *:` A data pointer to be passed to the `GL_VERTEX` callback function.

Returns: None.

See Also: `gluTessBeginPolygon`, `gluTessEndPolygon`, `gluTessBeginContour`, `gluTessEndContour`

Chapter 11. It's All About the Pipeline: Faster Geometry Throughput

by Richard S. Wright, Jr.

WHAT YOU'LL LEARN IN THIS CHAPTER:

How To	Functions You'll Use
Assemble polygons to create 3D objects	<code>glBegin/glEnd/glVertex</code>
Optimize object display with display lists	<code>glNewList/glEndList/glCallList</code>
Store and transfer geometry more efficiently	<code>glEnableClientState/</code> <code>glDisableClientState/</code> <code>glVertexPointer/glNormalPointer/</code> <code>glTexCoordPointer/glColorPointer/</code> <code>glEdgeFlagPointer/</code> <code>glFogCoordPointer/</code> <code>glSecondaryColorPointer/</code> <code>glArrayElement/glDrawArrays/</code> <code>glInterleavedArrays</code>
Reduce geometric bandwidth	<code>glDrawElements/</code> <code>glDrawRangeElements/</code> <code>glMultiDrawElements</code>

In the preceding chapters, we covered the basic OpenGL rendering techniques and technologies. Using this knowledge, there are few 3D scenes you can envision that cannot be realized using only the first half of this book. Now, however, we turn our attention to the techniques and practice of

rendering complex geometric models using your newfound knowledge of OpenGL rendering capabilities.

We begin with a basic overview of the many complex models assembled from simpler pieces. We then progress to some new OpenGL functionality that helps you more quickly move geometry and other OpenGL commands to the hardware renderer (graphics card). Finally, we introduce you to some higher level ideas to eliminate costly drawing commands and geometry that is outside your field of view.

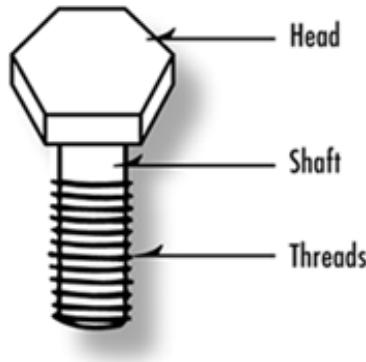
Model Assembly 101

Generally speaking, all 3D objects rendered with OpenGL are composed of some number of the 10 basic OpenGL rendering primitives: `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_QUADS`, `GL_QUAD_STRIP`, or `GL_POLYGON`. Managing all the individual vertices and primitive groups can become quite complex when the geometry grows in size or complexity. The usual way of dealing with highly complex objects is simply "divide and conquer"—break it down into many smaller, simpler pieces.

For example, the SNOWMAN sample in [Chapter 10](#), "Curves and Surfaces," was simply made out of spheres, cylinders, cones, and disks cleverly arranged with a few geometric transformations. We begin this chapter with another simple model that is composed of more than one individual part: a model of a metallic bolt (such as those holding your disk drive together). Although this particular bolt might not exist in any hardware store, it does have the essential features of our end goal.

The bolt will have a six-sided head and a threaded shaft, as do many typical steel bolts. Because this is a learning exercise, we simplify the threads by making them raised on the surface of the bolt shaft rather than carved out of the shaft. [Figure 11.1](#) provides a rough sketch of what we're aiming for. We build the three major components of this bolt—the head, shaft, and threads—individually and then put them together to form the final object.

Figure 11.1. The hex bolt to be modeled in this chapter.



Pieces and Parts

Any given programming task can be separated into smaller, more manageable tasks. Breaking down the tasks makes the smaller pieces easier to handle and code, and introduces some reusability into the code base as well. Three-dimensional modeling is no exception; you will typically create large, complex systems out of many smaller and more manageable pieces.

As previously mentioned, we have decided to break down the bolt into three pieces: head, shaft, and thread. Certainly, breaking down the tasks makes it much simpler for us to consider each section graphically, but it also gives us three objects that we can reuse. In more complex modeling applications, this reusability is of crucial importance. In a CAD-type application, for example, you would probably have many different bolts to model with various lengths, thicknesses, and thread densities. Instead of, say, a `RenderHead` function that draws the head of

the bolt, you might want to write a function that takes parameters specifying the number of sides, thickness, and diameter of the bolt head.

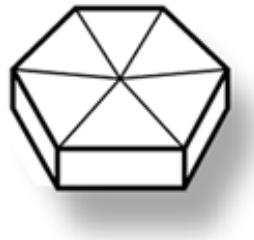
Another thing we do is model each piece of our bolt in coordinates that are most convenient for describing the object. Most often, individual objects or meshes are modeled around the origin and then translated and rotated into place. Later, when composing the final object, we can translate the components, rotate them, and even scale them if necessary to assemble our composite object. We do this for two very good reasons. First, this approach enables rotations to take place around the object's geometric center instead of some arbitrary point off to one side (depending on the whim of the modeler). The second reason will become more obvious later in the chapter. For now, suffice it to say that it is very useful to know the distance from any given point to the center of an object and to be able to more easily calculate an object's 3D extents (how big it is).

The Head

The head of our bolt has six smooth sides and is smooth on top and bottom. We can construct this solid object with two hexagons that represent the top and bottom of the head and a series of quadrilaterals around the edges to represent the sides. We could use `GL_QUAD_STRIP` to draw the head with a minimum number of vertices; however, as we discussed previously in [Chapter 6, "More on Colors and Materials,"](#) this approach would require that each edge share a surface normal. By using individual quads (`GL_QUADS`), we can at least cut down on sending one additional vertex to OpenGL per side (as opposed to sending down two triangles). For a small model such as this, the difference is negligible. For larger models, this step could mean a significant savings.

[Figure 11.2](#) illustrates how the bolt head is constructed with the triangle fan and quads. We use a triangle fan with six triangles for the top and bottom sections of the head. Then we compose each face of the side of the bolt with a single quad.

Figure 11.2. Primitive outline of bolt head.



We used a total of 18 primitives to draw the bolt head: 6 triangles (or one fan) each on the top and bottom and 6 quads to compose the sides of the bolt head. [Listing 11.1](#) contains the function that renders the bolt head. [Figure 11.3](#) shows what the bolt head looks like when rendered by itself (the completed program is the BOLT sample on the CD). This code contains only functions we've already covered, but it's more substantial than any of the simpler chapter examples. Also, note that the origin of the coordinate system is in the exact center of the bolt head.

Listing 11.1. Code to Render the Bolt Head

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Creates the head of the bolt
void RenderHead(void)
{
    float x,y,angle;                                // Calculated positions
    float height = 25.0f;                            // Thickness of the head
    float diameter = 30.0f;                           // Diameter of the head
    GLTVector3 vNormal,vCorners[4];                 // Storage of vertices and normals
    float step = (3.1415f/3.0f);                    // step = 1/6th of a circle = hexagon
    // Set material color for head of bolt
    glColor3f(0.0f, 0.0f, 0.7f);
```

```

// Begin a new triangle fan to cover the top
glFrontFace(GL_CCW);
glBegin(GL_TRIANGLE_FAN);
    // All the normals for the top of the bolt point straight up
    // the z axis.
    glNormal3f(0.0f, 0.0f, 1.0f);
    // Center of fan is at the origin
    glVertex3f(0.0f, 0.0f, height/2.0f);
    // Divide the circle up into 6 sections and start dropping
    // points to specify the fan. We appear to be winding this
    // fan backwards. This has the effect of reversing the winding
    // of what would have been a CW wound primitive. Avoiding a state
    // change with glFrontFace().
    // First and Last vertex closes the fan
    glVertex3f(0.0f, diameter, height/2.0f);
    for(angle = (2.0f*3.1415f)-step; angle >= 0; angle -= step)
    {
        // Calculate x and y position of the next vertex
        x = diameter*(float)sin(angle);
        y = diameter*(float)cos(angle);
        // Specify the next vertex for the triangle fan
        glVertex3f(x, y, height/2.0f);
    }

    // Last vertex closes the fan
    glVertex3f(0.0f, diameter, height/2.0f);
// Done drawing the fan that covers the bottom
glEnd();
// Begin a new triangle fan to cover the bottom
glBegin(GL_TRIANGLE_FAN);
    // Normal for bottom points straight down the negative z axis
    glNormal3f(0.0f, 0.0f, -1.0f);
    // Center of fan is at the origin
    glVertex3f(0.0f, 0.0f, -height/2.0f);
    // Divide the circle up into 6 sections and start dropping
    // points to specify the fan
    for(angle = 0.0f; angle < (2.0f*3.1415f); angle += step)
    {
        // Calculate x and y position of the next vertex
        x = diameter*(float)sin(angle);
        y = diameter*(float)cos(angle);
        // Specify the next vertex for the triangle fan
        glVertex3f(x, y, -height/2.0f);
    }

    // Last vertex, used to close the fan
    glVertex3f(0.0f, diameter, -height/2.0f);
// Done drawing the fan that covers the bottom
glEnd();
// Build the sides out of triangles (two each). Each face
// will consist of two triangles arranged to form a
// quadrilateral
glBegin(GL_QUADS);

    // Go around and draw the sides
    for(angle = 0.0f; angle < (2.0f*3.1415f); angle += step)
    {
        // Calculate x and y position of the next hex point
        x = diameter*(float)sin(angle);
        y = diameter*(float)cos(angle);
        // start at bottom of head
        vCorners[0][0] = x;
        vCorners[0][1] = y;

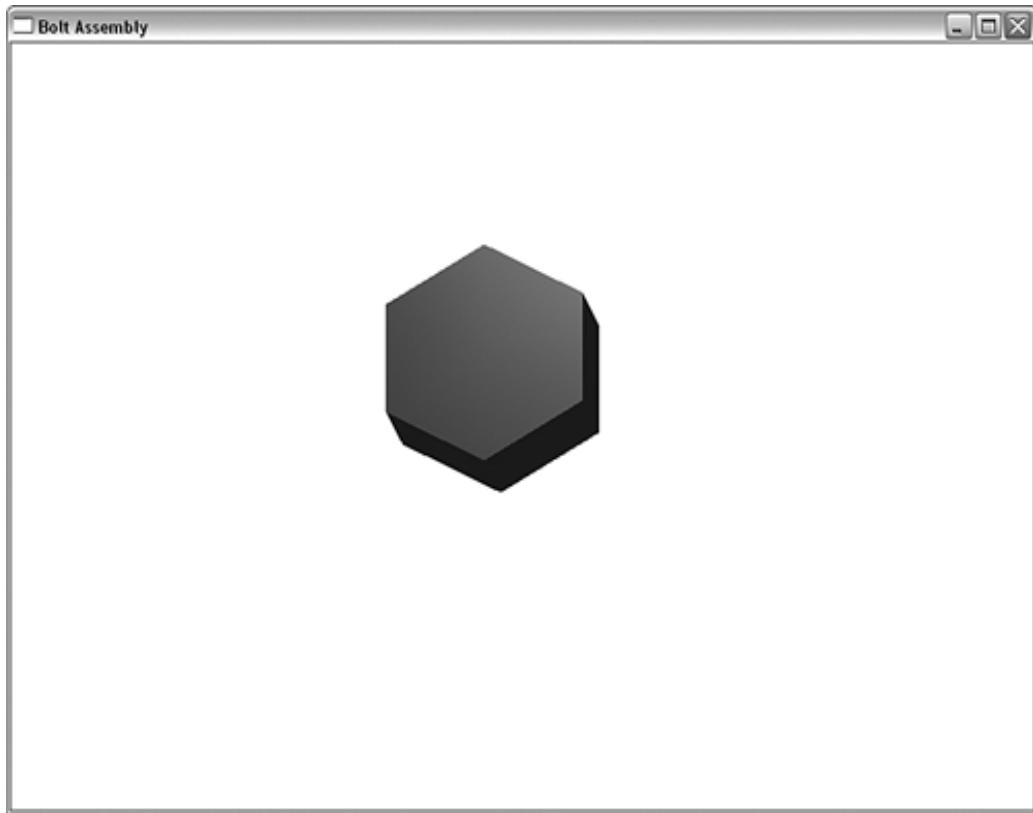
```

```

vCorners[0][2] = -height/2.0f;
// extrude to top of head
vCorners[1][0] = x;
vCorners[1][1] = y;
vCorners[1][2] = height/2.0f;
// Calculate the next hex point
x = diameter*(float)sin(angle+step);
y = diameter*(float)cos(angle+step);
// Make sure we aren't done before proceeding
if(angle+step < 3.1415*2.0)
{
    // If we are done, just close the fan at a
    // known coordinate.
    vCorners[2][0] = x;
    vCorners[2][1] = y;
    vCorners[2][2] = height/2.0f;
    vCorners[3][0] = x;
    vCorners[3][1] = y;
    vCorners[3][2] = -height/2.0f;
}
else
{
    // We aren't done, the points at the top and bottom
    // of the head.
    vCorners[2][0] = 0.0f;
    vCorners[2][1] = diameter;
    vCorners[2][2] = height/2.0f;
    vCorners[3][0] = 0.0f;
    vCorners[3][1] = diameter;
    vCorners[3][2] = -height/2.0f;
}
// The normal vectors for the entire face will
// all point the same direction
gltGetNormalVector(vCorners[0], vCorners[1], vCorners[2], vNormal);
glNormal3fv(vNormal);
// Specify each quad separately to lie next
// to each other.
glVertex3fv(vCorners[0]);
glVertex3fv(vCorners[1]);
glVertex3fv(vCorners[2]);
glVertex3fv(vCorners[3]);
}
glEnd();
}

```

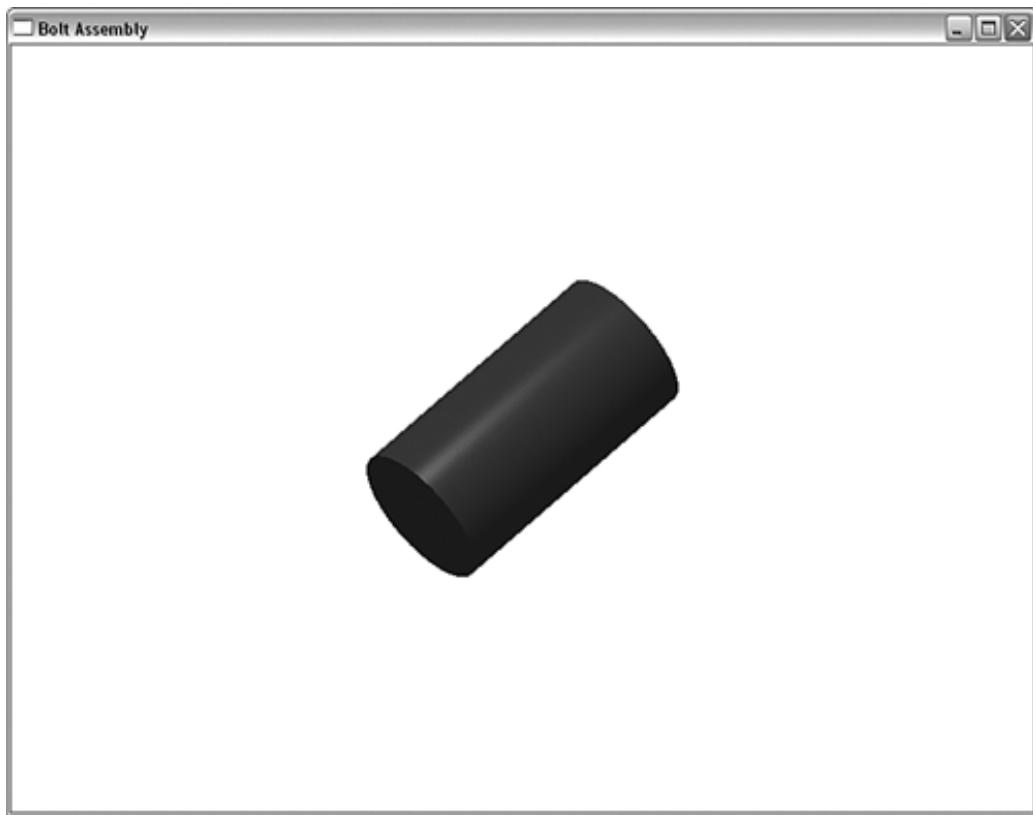
Figure 11.3. Output from the head program.



The Shaft

The shaft of the bolt is nothing more than a cylinder with a bottom on it. We compose a cylinder by plotting x,z values around in a circle and then take two y values at these points and get polygons that approximate the wall of a cylinder. This time, however, we compose this wall entirely out of a quad strip because each adjacent quad can share a normal for smooth shading (see [Chapter 5](#), "Color, Materials, and Lighting: The Basics"). [Figure 11.4](#) shows the rendered cylinder.

Figure 11.4. The shaft of the bolt, rendered as a quad strip around the shaft body.



We also create the bottom of the shaft with a triangle fan, as we did for the bottom of the bolt head previously. Notice now, however, that the smaller step size around the circle yields smaller flat facets, which make the cylinder wall more closely approximate a smooth curve. The step size also matches that used for the shaft wall so that they match evenly.

[Listing 11.2](#) provides the code to produce this cylinder. Notice that the normals are not calculated for the quads using the vertices of the quads. We usually set the normal to be the same for all vertices, but here, we break with this tradition to specify a new normal for each vertex. Because we are simulating a curved surface, the normal specified for each vertex is normal to the actual curve. (If this description seems confusing, review [Chapter 5](#) on normals and lighting effects.)

Listing 11.2. Code to Render the Shaft of the Bolt

```
////////////////////////////////////////////////////////////////
// Creates the shaft of the bolt as a cylinder with one end
// closed.
void RenderShaft(void)
{
    float x,z,angle;                                // Used to calculate cylinder wall
    float height = 75.0f;                            // Height of the cylinder
    float diameter = 20.0f;                           // Diameter of the cylinder
    GLTVector3 vNormal,vCorners[2];                 // Storage for vertex calculations
    float step = (3.1415f/50.0f);                   // Approximate the cylinder wall with
                                                // 100 flat segments.

    // Set material color for head of screw
    glColor3f(0.0f, 0.0f, 0.7f);
    // First assemble the wall as 100 quadrilaterals formed by
    // placing adjoining Quads together
    glFrontFace(GL_CCW);
    glBegin(GL_QUAD_STRIP);
        // Go around and draw the sides
        for(angle = (2.0f*3.1415f); angle > 0.0f; angle -= step)
    {
        // Calculate x and y position of the first vertex
        // ... (code continues for vertex calculations and rendering)
    }
}
```

```

x = diameter*(float)sin(angle);
z = diameter*(float)cos(angle);
// Get the coordinate for this point and extrude the
// length of the cylinder.
vCorners[0][0] = x;
vCorners[0][1] = -height/2.0f;
vCorners[0][2] = z;
vCorners[1][0] = x;
vCorners[1][1] = height/2.0f;
vCorners[1][2] = z;
// Instead of using real normal to actual flat section
// Use what the normal would be if the surface was really
// curved. Since the cylinder goes up the Y axis, the normal
// points from the Y axis out directly through each vertex.
// Therefore we can use the vertex as the normal, as long as
// we reduce it to unit length first and assume the y component
// to be zero
vNormal[0] = vCorners[1][0];
vNormal[1] = 0.0f;
vNormal[2] = vCorners[1][2];
// Reduce to length of one and specify for this point
glNormalNormalizeVector(vNormal);
glNormal3fv(vNormal);
glVertex3fv(vCorners[0]);
glVertex3fv(vCorners[1]);
}
// Make sure there are no gaps by extending last quad to
// the original location
glVertex3f(diameter*(float)sin(2.0f*3.1415f), -height/2.0f,
           diameter*(float)cos(2.0f*3.1415f));
glVertex3f(diameter*(float)sin(2.0f*3.1415f), height/2.0f,
           diameter*(float)cos(2.0f*3.1415f));
glEnd(); // Done with cylinder sides
// Begin a new triangle fan to cover the bottom
glBegin(GL_TRIANGLE_FAN);
// Normal points down the Y axis
glNormal3f(0.0f, -1.0f, 0.0f);
// Center of fan is at the origin
glVertex3f(0.0f, -height/2.0f, 0.0f);
// Spin around matching step size of cylinder wall
for(angle = (2.0f*3.1415f); angle > 0.0f; angle -= step)
{
    // Calculate x and y position of the next vertex
    x = diameter*(float)sin(angle);
    z = diameter*(float)cos(angle);
    // Specify the next vertex for the triangle fan
    glVertex3f(x, -height/2.0f, z);
}
// Be sure loop is closed by specifying initial vertex
// on arc as the last too
glVertex3f(diameter*(float)sin(2.0f*3.1415f), -height/2.0f,
           diameter*(float)cos(2.0f*3.1415f));
glEnd();
}

```

The Thread

The thread is the most complex part of the bolt. It's composed of two planes arranged in a V shape that follows a corkscrew pattern up the length of the shaft. [Figure 11.5](#) shows the rendered thread, and [Listing 11.3](#) provides the OpenGL code used to produce this shape.

Listing 11.3. Code to Render the Threads

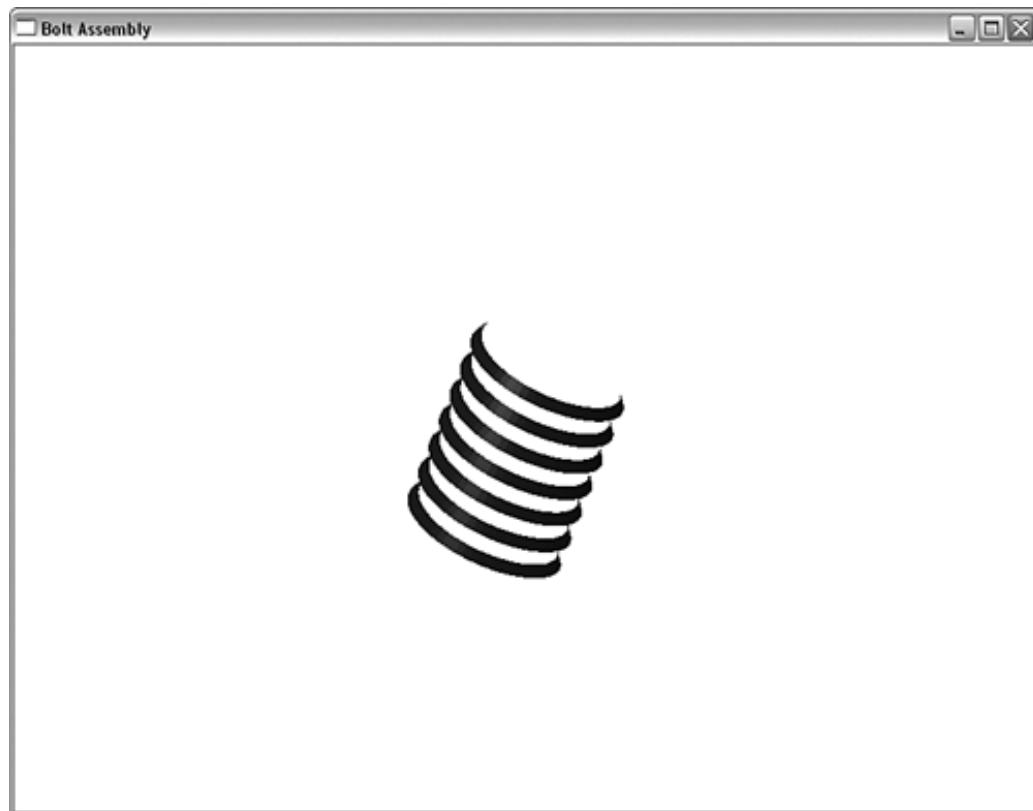
```
///////////////////////////////
// Spiraling thread
void RenderThread(void)
{
    float x,y,z,angle;           // Calculate coordinates and step angle
    float height = 75.0f;        // Height of the threading
    float diameter = 20.0f;      // Diameter of the threading
    GLTVector3 vNormal, vCorners[4]; // Storage for normal and corners
    float step = (3.1415f/32.0f); // one revolution
    float revolutions = 7.0f;     // How many times around the shaft
    float threadWidth = 2.0f;    // How wide is the thread
    float threadThick = 3.0f;    // How thick is the thread
    float zstep = .125f;         // How much does the thread move up
                                 // the Z axis each time a new segment
                                 // is drawn.

    // Set material color for head of screw
    glColor3f(0.0f, 0.0f, 0.4f);
    z = -height/2.0f+2.0f;      // Starting spot almost to the end

    // Go around and draw the sides until finished spinning up
    for(angle = 0.0f; angle < GLT_PI * 2.0f * revolutions; angle += step)
    {
        // Calculate x and y position of the next vertex
        x = diameter*(float)sin(angle);
        y = diameter*(float)cos(angle);
        // Store the next vertex next to the shaft
        vCorners[0][0] = x;
        vCorners[0][1] = y;
        vCorners[0][2] = z;
        // Calculate the position away from the shaft
        x = (diameter+threadWidth)*(float)sin(angle);
        y = (diameter+threadWidth)*(float)cos(angle);
        vCorners[1][0] = x;
        vCorners[1][1] = y;
        vCorners[1][2] = z;
        // Calculate the next position away from the shaft
        x = (diameter+threadWidth)*(float)sin(angle+step);
        y = (diameter+threadWidth)*(float)cos(angle+step);
        vCorners[2][0] = x;
        vCorners[2][1] = y;
        vCorners[2][2] = z + zstep;
        // Calculate the next position along the shaft
        x = (diameter)*(float)sin(angle+step);
        y = (diameter)*(float)cos(angle+step);
        vCorners[3][0] = x;
        vCorners[3][1] = y;
        vCorners[3][2] = z+ zstep;
        // We'll be using triangles, so make
        // counterclockwise polygons face out
        glFrontFace(GL_CCW);
        glBegin(GL_TRIANGLES); // Start the top section of thread
        // Calculate the normal for this segment
        gltGetNormalVector(vCorners[0], vCorners[1], vCorners[2], vNormal);
        glNormal3fv(vNormal);
        // Draw two triangles to cover area
        glVertex3fv(vCorners[0]);
        glVertex3fv(vCorners[1]);
        glVertex3fv(vCorners[2]);
        glVertex3fv(vCorners[2]);
        glVertex3fv(vCorners[3]);
    }
}
```

```
    glVertex3fv(vCorners[0]);
glEnd();
// Move the edge along the shaft slightly up the z axis
// to represent the bottom of the thread
vCorners[0][2] += threadThick;
vCorners[3][2] += threadThick;
// Recalculate the normal since points have changed, this
// time it points in the opposite direction, so reverse it
gltGetNormalVector(vCorners[0], vCorners[1], vCorners[2], vNormal);
vNormal[0] = -vNormal[0];
vNormal[1] = -vNormal[1];
vNormal[2] = -vNormal[2];
// Switch to clockwise facing out for underside of the
// thread.
glFrontFace(GL_CW);
// Draw the two triangles
glBegin(GL_TRIANGLES);
    glNormal3fv(vNormal);
    glVertex3fv(vCorners[0]);
    glVertex3fv(vCorners[1]);
    glVertex3fv(vCorners[2]);
    glVertex3fv(vCorners[2]);
    glVertex3fv(vCorners[3]);
    glVertex3fv(vCorners[0]);
glEnd();
// Creep up the Z axis
z += zstep;
}
}
```

Figure 11.5. The bolt threads winding up the shaft (shown without shaft).

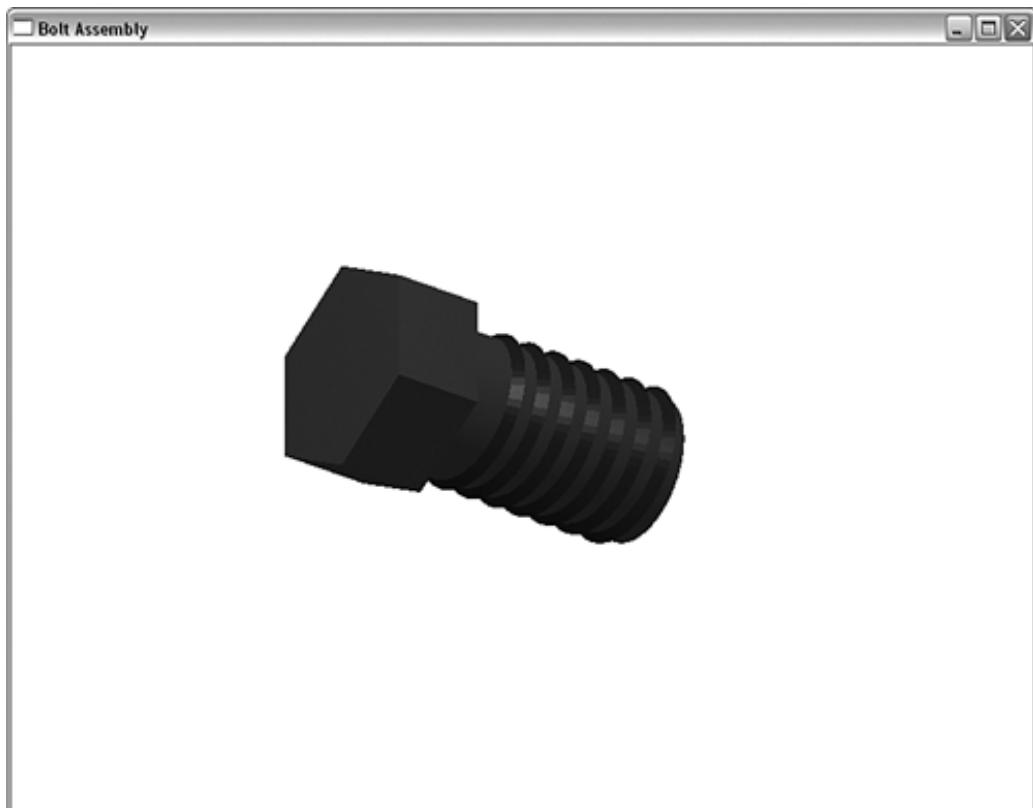


We assemble the bolt by drawing all three sections in their appropriate location. All sections are translated and rotated appropriately into place. The shaft is not modified at all, and the threads must be rotated to match the shaft. Finally, the head of the bolt must be rotated and translated to put it in its proper place. [Listing 11.4](#) provides the rendering code that manipulates and renders the three bolt components. [Figure 11.6](#) shows the final output of the bolt program.

Listing 11.4. Code to Render All the Pieces in Place

```
// Called to draw scene
void RenderScene(void)
{
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // Save the matrix state
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    // Rotate about x and y axes
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot, 0.0f, 0.0f, 1.0f);
    // Render just the Thread of the nut
    RenderShaft();
    glPushMatrix();
    glRotatef(-90.0f, 1.0f, 0.0f, 0.0f);
    RenderThread();
    glTranslatef(0.0f,0.0f,45.0f);
    RenderHead();
    glPopMatrix();
    glPopMatrix();
    // Swap buffers
    glutSwapBuffers();
}
```

Figure 11.6. Output from the bolt program.



So why all the gymnastics to place all these pieces together? We easily could have adjusted our geometry so that all the pieces would be drawn in their correct location. The point shown here is that many pieces modeled about their own local origins can be arranged together in a scene quite easily to create a more sophisticated model or environment. This basic principle and technique form the foundation of creating your own scene graph (see [Chapter 1](#), "Introduction to 3D Graphics and OpenGL") or virtual environment. We have been putting this principle into practice all along with the SPHEREWORLD samples in each chapter. In a complex and persistent (saved to disk) 3D environment, each piece's position and orientation could be stored individually using a `GLTFrame` structure from the `glTools` library.

Display Lists

The BOLT program produces a reasonable representation of the metal bolt we set out to model. Consisting of more than 1,700 triangles, this bolt is the most complex manually generated example in this book so far in terms of geometry. Comparatively speaking, however, this number of triangles isn't anywhere close to the largest number of polygons you'll encounter when composing larger scenes and more complex objects. In fact, the latest 3D accelerated graphics cards are rated at millions of triangles per second, and that's for the cheap ones! One of the goals of this chapter is to introduce you to some more efficient ways to store and render your geometry. One of the simplest and most effective ways to do this is to use OpenGL display lists.

OPTIMIZING MODEL RENDERING

In our BOLT example, we constructed the model by applying mathematics to represent the curves and surfaces as equations. Using points along these equations, we constructed a series of triangles to create the representative shape. Although the bolt head and shaft were simple, the code to create the thread geometry can be a bit more intimidating. In general, creating models or geometry can be quite time consuming due to many factors. Perhaps you have to compute all the vertices and normals such as we have here. Perhaps you need to read these values from disk or retrieve them over a network. With these considerations in mind, rendering performance could be hampered more by other tasks related to creating the geometry than by the actual rendering itself. OpenGL provides two solutions to overcoming these issues, each of which has its own merits. These solutions, display lists and vertex arrays, are the main focus of the remainder of this chapter.

Batch Processing

OpenGL has been described as a software interface to graphics hardware. As such, you might imagine that OpenGL commands are somehow converted into some specific hardware commands or operators by the driver and then sent on to the graphics card for immediate execution. If so, you would be *mostly* correct. Most OpenGL rendering commands are, in fact, converted into some hardware-specific commands, but these commands are not dispatched immediately to the hardware. Instead, they are accumulated in a local buffer until some threshold is reached, at which point they are *flushed* to the hardware.

The primary reason for this type of arrangement is that trips to the graphics hardware take a long time, at least in terms of computer time. To a human being, this process might take place very quickly, but to a CPU running at many billions of cycles per second, this is like waiting for a cruise ship to sail from North America to Europe and back. You certainly would not put a single person on a ship and wait for the ship to return before loading up the next person. If you have many people to send to Europe, you are going to cram as many people on the ship as you can! This analogy is very accurate: It is faster to send a large amount of data (within some limits) over the system bus to hardware all at once than to break it down into many bursts of smaller packages.

Keeping to the analogy, you also do not have to wait for the first cruise ship to return before you can begin filling the next ship with passengers. Sending the buffer to the graphics hardware (a

process called *flushing*) is an *asynchronous* operation. This means that the CPU can move on to other tasks and does not have to wait for the batch of rendering commands just sent to be completed. You can literally have the hardware rendering a given set of commands while the CPU is busy calling a new set of commands for the next graphics image (typically called a *frame* when you're creating an animation). This type of *parallelization* between the graphics hardware and the host CPU is highly efficient and often sought after by performance-conscious programmers.

Three events trigger a flush of the current batch of rendering commands. The first occurs when the driver's command buffer is full. You do not have access to this buffer, nor do you have any control over the size of the buffer. The hardware vendors work hard to tune the size and other characteristics of this buffer to work well with their devices. A flush also occurs when you execute a buffer swap. The buffer swap cannot occur until all pending commands have been executed (you want to see what you have drawn!), so the flush is initiated, followed by the command to perform the buffer swap. A buffer swap is an obvious indicator to the driver that you are done with a given scene and that all commands should be rendered. However, if you are doing single-buffered rendering, OpenGL has no real way of knowing when you're done sending commands and thus when to send the batch of commands to the hardware for execution. To facilitate this process, you can call the following function to manually trigger a flush:

```
void glFlush(void);
```

Some OpenGL commands, however, are not buffered for later execution—for example, `glReadPixels` and `glDrawPixels`. These functions directly access the framebuffer and read or write data directly. Therefore, it is useful to be able not only to flush the buffer, but also to wait for all the commands to be executed before calling one of these functions. For this reason, you also do not put these commands in a display list. For example, if you render an image that you want to read back with `glReadPixels`, you could read the framebuffer before the command batch has even been flushed. To both force a flush and wait for the all previous rendering commands to finish, call the following function:

```
void glFinish(void);
```

Preprocessed Batches

The work done every time you call an OpenGL command is not inconsequential. Commands are *compiled*, or converted, from OpenGL's high-level command language into low-level hardware commands understood by the hardware. For complex geometry, or just large amounts of vertex data, this process is performed many thousands of times, just to draw a single image onscreen. Often, the geometry or other OpenGL data remains the same from frame to frame. A solution to this needlessly repeated overhead is to save a chunk of data from the command buffer that performs some repetitive rendering task. This chunk of data can later be copied back to the command buffer all at once, saving the many function calls and compilation work done to create the data.

OpenGL provides a facility to create a preprocessed set of OpenGL commands (the chunk of data) that can then be quickly copied to the command buffer for more rapid execution. This precompiled list of commands is called a display list, and creating one or more of them is an easy and straightforward process. Just as you delimit an OpenGL primitive with `glBegin/glEnd`, you delimit a display list with `glNewList/glEndList`. A display list, however, is named with an integer value that you supply. The following code fragment represents a typical example of display list creation:

```
glNewList(<unsigned integer name>, GL_COMPILE);
...
...
// Some OpenGL Code
...
...
glEndList();
```

The named display list now contains all OpenGL rendering commands that occur between the `glNewList` and `glEndList` function calls. The `GL_COMPILE` parameter tells OpenGL to compile the list but not to execute it yet. You can also specify `GL_COMPILE_AND_EXECUTE` to simultaneously build the display list and execute the rendering instructions. Typically, however, display lists are built (`GL_COMPILE` only) during program initialization and then executed later during rendering.

The display list name can be any unsigned integer. However, if you use the same value twice, the second display list overwrites the previous one. For this reason, it is convenient to have some sort of mechanism to keep you from reusing the same display list more than once. This is especially helpful when you are incorporating libraries of code written by someone else who may have incorporated display lists and may have chosen the same display list names.

OpenGL provides built-in support for allocating unique display list names. The following function returns the first of a series of display list integers that are unique:

```
GLuint glGenLists(GLsizei range);
```

The display list names are reserved sequentially, with the first name being returned by the function. You can call this function as often as you want and for as many display list names at a time as you may need. A corresponding function frees display list names and releases any memory allocated for those display lists:

```
void glDeleteLists(GLuint list, GLsizei range);
```

A display list, containing any number of precompiled OpenGL commands, is then executed with a single command:

```
void glCallList(GLuint list);
```

You can also execute a whole array of display lists with this command:

```
void glCallLists(GLsizei n, GLenum type, const GLvoid *lists);
```

The first parameter specifies the number of display lists contained by the array `lists`. The second parameter contains the data type of the array; typically, it is `GL_UNSIGNED_BYTE`.

Display List Caveats

A few important points about display lists are worth mentioning here. Although on most implementations, a display list should improve performance, your mileage may vary depending on the amount of effort the vendor puts into optimizing display list creation and execution. It is rare, however, for display lists not to offer a noticeable boost in performance, and they are widely relied on in applications that use OpenGL.

Display lists are typically good at creating precompiled lists of OpenGL commands, especially if the list contains state changes (turning lighting on and off, for example). If you do not create a display list name with `glGenLists` first, you might get a working display list on some implementations, but not on others. Some commands simply do not make sense in a display list. For example, reading the framebuffer into a pointer with `glReadPixels` makes no sense in a display list.

Likewise, calls to `glTexImage2D` would store the original image data in the display list, followed by the command to load the image data as a texture. Basically, your textures stored this way would take up twice as much memory! Display lists excel, however, at precompiled lists of geometry, with texture objects bound either inside or outside the display lists. Finally, display lists cannot contain calls that create display lists. You can have one display list call another, but you cannot put calls to `glNewList`/`glEndList` inside a display list.

Converting to Display Lists

Converting the BOLT sample to use display lists requires only a few additional lines of code. First, we add three variables that contain the display list identifiers for the three pieces of the bolt:

```
// Display list identifiers
GLuint headList, shaftList, threadList;
```

Then, in the `SetupRC` function, we request three display list names and assign them to our display list variables:

```
// Get Display list names
headList = glGenLists(3);
shaftList = headList + 1;
threadList = headList + 2;
```

Next, we add the code to generate the three display lists. Each display list simply calls the function that draws that piece of geometry:

```
// Prebuild the display lists
glNewList(headList, GL_COMPILE);
    RenderHead();
glEndList();
glNewList(shaftList, GL_COMPILE);
    RenderShaft();
glEndList();
glNewList(threadList, GL_COMPILE);
    RenderThread();
glEndList();
```

Finally, in the `Render` function, we simply replace each function call for the bolt pieces with the appropriate display list call:

```
// Render just the Thread of the nut
//RenderShaft();
glCallList(shaftList);
glPushMatrix();
glRotatef(-90.0f, 1.0f, 0.0f, 0.0f);
//RenderThread();
glCallList(threadList);
glTranslatef(0.0f, 0.0f, 45.0f);
//RenderHead();
glCallList(headList);
```

In this example, we have created three display lists, one for each component of the bolt. We also could have placed the entire bolt in a single display list or even created a fourth display list that contained calls to the other three. You can find the complete code for the display list version of the bolt in the BOLTDL sample program on the CD.

Measuring Performance

It is difficult to demonstrate the performance enhancements made by using display lists with something as simple as the BOLT example. To demonstrate the advantages of using display lists (or vertex arrays, for that matter), we need two things. First, we need a sample program with a more serious amount of geometry. Second, we need a way to measure performance besides some subjective measure of how fast an animation appears to be running.

Most chapters have included a SPHEREWORLD sample program that demonstrates an immersive 3D environment, enhanced using techniques presented in that particular chapter. By this point in the book, the SphereWorld contains a lot of geometry. A highly tessellated ground and torus and a number of high-resolution spheres inhabit the plane. In addition, the planar shadow algorithm we used requires that nearly all the geometry be processed twice (once for the object, once for the shadow). Certainly, this sample program should see some visible benefit from a retrofit using display lists.

A simple and meaningful measure of rendering performance is the measure of how many frames (individual images) can be rendered per second. In fact, many games and graphical benchmarking programs have options to display the frame rate prominently, as a frames per second (fps) indicator. For this chapter's installment of SPHEREWORLD, we will add both a frame rate indication and the option to use or not use display lists. The difference measured in fps should give us a reasonable indication of the type of performance improvement that display lists can frequently contribute to our applications.

The frame rate is simply the number of frames rendered over some period of time, divided by the amount of time elapsed. Counting buffer swaps is relatively simple...counting seconds, as it turns out, is not as easy as it sounds. High-resolution time keeping is unfortunately an operating system and hardware platform feature that is not very portable. Time-keeping functions are also documented poorly and can imply misleading performance characteristics. For example, most standard C runtime functions that can return time to the nearest millisecond often have a resolution of many, many milliseconds. Subtracting two times should give you the difference between events, but sometimes the minimum amount of time you can actually measure is as much as 1/20th of a second. With timer resolution this poor, you can sometimes render several frames and buffer swaps without being able to see any difference in time pass at all!

The `glTools` library contains a time data structure and two time functions that isolate operating system dependencies and give fairly good timing resolution. On the PC, it is often on the order of millionths of a second, and on the Mac, you will get at least 10ms resolution. These functions work something like a stopwatch. In fact, the data structure that contains the last sampled time is defined as such:

```
GLTStopwatch frameTimer;
```

These next two functions reset the stopwatch and read the number of elapsed seconds (as a floating-point value) since the last time the stopwatch was reset:

```
void gltStopwatchReset(GLTStopwatch *pTimer);
float gltStopwatchRead(GLTStopwatch *pTimer);
```

You must reset the stopwatch at least one time before any read time values will have any meaning.

A Better Example

Because SPHEREWORLD is a fairly long program and has already been introduced in earlier chapters, [Listing 11.5](#) shows only the new `RenderScene` function, which contains some noteworthy changes.

Listing 11.5. Main Rendering Function for SPHEREWORLD

```

void RenderScene(void)
{
    static int iFrames = 0;           // Frame count
    static GLStopwatch frameTimer;   // Render time
    // Reset the stopwatch on first time
    if(iFrames == 0)
    {
        gltStopwatchReset(&frameTimer);
        iFrames++;
    }
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
    glPushMatrix();
    gltApplyCameraTransform(&frameCamera);
    // Position light before any other transformations
    glLightfv(GL_LIGHT0, GL_POSITION, fLightPos);
    // Draw the ground
    glColor3f(1.0f, 1.0f, 1.0f);
    if(iMethod == 0)
        DrawGround();
    else
        glCallList(groundList);
    // Draw shadows first
    glDisable(GL_DEPTH_TEST);
    glDisable(GL_LIGHTING);
    glDisable(GL_TEXTURE_2D);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glEnable(GL_STENCIL_TEST);
    glPushMatrix();
    glMultMatrixf(mShadowMatrix);
    DrawInhabitants(1);
    glPopMatrix();
    glDisable(GL_STENCIL_TEST);
    glDisable(GL_BLEND);
    glEnable(GL_LIGHTING);
    glEnable(GL_TEXTURE_2D);
    glEnable(GL_DEPTH_TEST);
    // Draw inhabitants normally
    DrawInhabitants(0);
    glPopMatrix();
    // Do the buffer Swap
    glutSwapBuffers();
    // Increment the frame count
    iFrames++;
    // Do periodic frame rate calculation
    if(iFrames == 101)
    {
        float fps;
        char cBuffer[64];
        fps = 100.0f / gltStopwatchRead(&frameTimer);
        if(iMethod == 0)
            sprintf(cBuffer,
                    "OpenGL SphereWorld without Display Lists %.1f fps", fps);
        else
            sprintf(cBuffer,
                    "OpenGL SphereWorld with Display Lists %.1f fps", fps);
        glutSetWindowTitle(cBuffer);
        gltStopwatchReset(&frameTimer);
        iFrames = 1;
    }
    // Do it again
}

```

```
glutPostRedisplay();
}
```

First, we have two static variables that will retain their values from one function call to the next. They are the frame counter and frame timer:

```
static int iFrames = 0;           // Frame count
static GLTStopwatch frameTimer; // Render time
```

Because the timer must be initialized before use, we use the frame counter as a sentinel. The `iFrames` variable contains only the value 0 the first time the scene is rendered, so here we perform the initial stopwatch reset:

```
// Reset the stopwatch on first time
if(iFrames == 0)
{
    gltStopwatchReset(&frameTimer);
    iFrames++;
}
```

Next, we render the scene *almost* as usual. Note this change to the place where the ground is drawn:

```
if(iMethod == 0)
    DrawGround();
else
    glCallList(groundList);
```

The `iMethod` variable is set to 0 when the pop-up menu selection is Without Display Lists and 1 when With Display Lists is selected. A display list is generated in the `SetupRC` function for the ground, the torus, and a sphere. The `DrawInhabitants` function, likewise, switches between the display list and base function call as indicated by the `iMethod` variable. After the buffer swap, we simply increment the frame counter:

```
// Do the buffer Swap
glutSwapBuffers();
// Increment the frame count
iFrames++;
```

The frame rate is not calculated every single frame. Instead, we count some number of frames and then divide the total time by the number of frames. This approach serves two purposes. First, if the timer resolution is not terribly great, spreading out the time helps mitigate this problem by reducing the percentage of the total time represented by the error. Second, the process of calculating and displaying the frame rate takes time and will slow down the rendering, as well as adversely affect the accuracy of the measurement.

When the frame counter reaches 101, we have rendered 100 frames (recall, we start at 1, not 0). So, we create a string buffer, fill it with our frame rate calculation, and simply pop it into the window title bar. Finally, we must reset the timer and our counter to 1:

```
// Do periodic frame rate calculation
if(iFrames == 101)
{
    float fps;
    char cBuffer[64];
    fps = 100.0f / gltStopwatchRead(&frameTimer);
    if(iMethod == 0)
        sprintf(cBuffer,
```

```
    "OpenGL SphereWorld without Display Lists %.1f fps", fps);
else
    sprintf(cBuffer,
        "OpenGL SphereWorld with Display Lists %.1f fps", fps);
glutSetWindowTitle(cBuffer);
glStopwatchReset(&frameTimer);
iFrames = 1;
}
```

Switching to display lists can have an amazing impact on performance. Some OpenGL implementations even try to store display lists in memory on the graphics card, if possible, further reducing the work required to get the data to the graphics processor. [Figure 11.7](#) shows the SPHEREWORLD sample running without using display lists. The frame rate is fairly high already on a modern consumer graphics card. However, in [Figure 11.8](#), you can see the frame rate is far and away higher.

Figure 11.7. SPHEREWORLD without display lists.

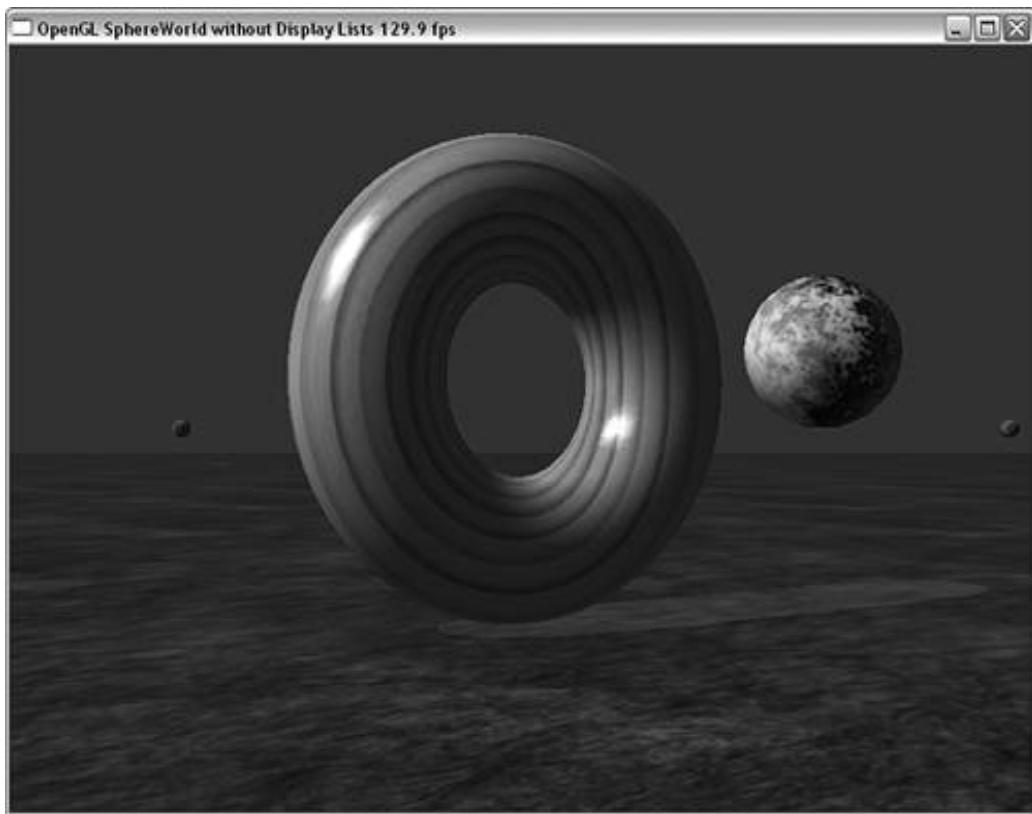
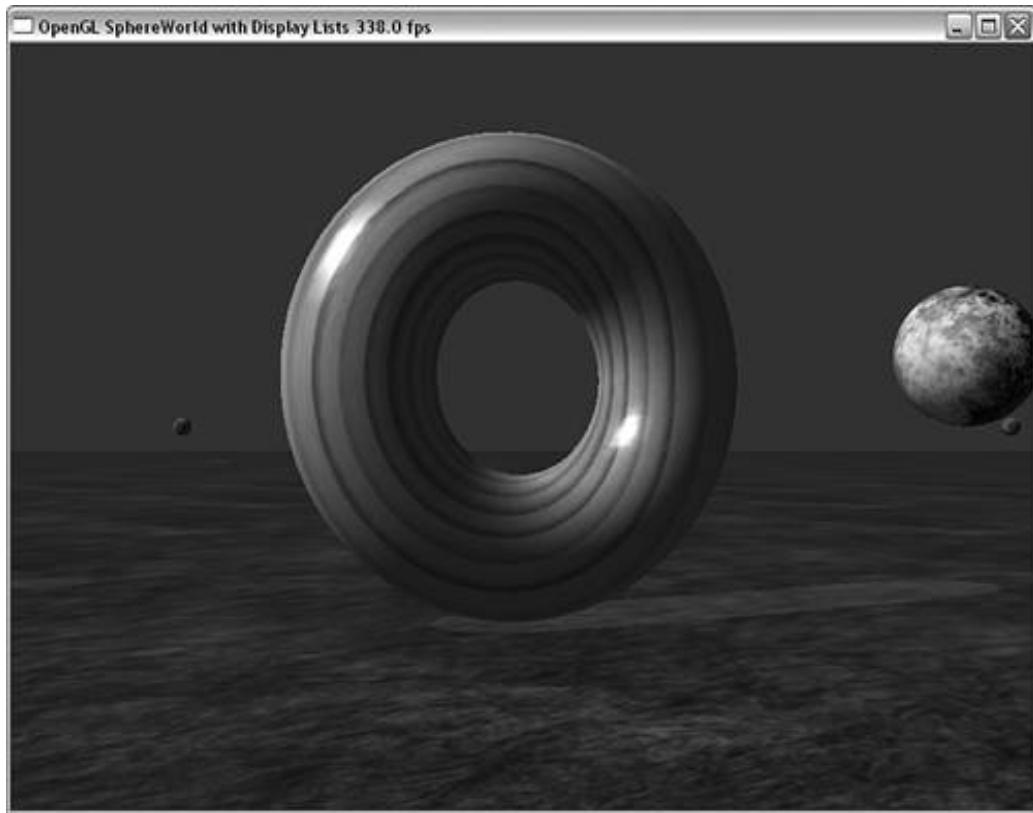


Figure 11.8. SPHEREWORLD with display lists.



Why should you care about rendering performance? The faster and more efficient your rendering code, the more visual complexity you can add to your scene without dragging down the frame rate too much. Higher frame rates yield smoother and better-looking animations. You can also use the extra CPU time to perform other tasks such as physics calculations or lengthy I/O operations on a separate thread.

Vertex Arrays

Display lists are frequently used for precompiling sets of OpenGL commands. In our BOLT example, display lists were perhaps a bit underused because all we really encapsulated was the creation of the geometry. In the same vein, SphereWorld's many spheres required a great deal of trigonometric calculations saved by placing the geometry in display lists. You might consider that we could just as easily have created some arrays to store the vertex data for the models and thus saved all the computation time just as easily as with the display lists.

You might be right about this way of thinking—to a point. Some implementations store display lists more efficiently than others, and if all you're really compiling is the vertex data, you can simply place the model's data in one or more arrays and render from the array of precalculated geometry. The only drawback to this approach is that you must still loop through the entire array moving data to OpenGL one vertex at a time. Depending on the amount of geometry involved, taking this approach could be a substantial performance penalty. The advantage, however, is that, unlike with display lists, the geometry does not have to be static. Each time you prepare to render the geometry, some function could be applied to all the geometry data and perhaps displace or modify it in some way. For example, say a mesh used to render the surface of an ocean could have rippling waves moving across the surface. A swimming whale or jellyfish could also be cleverly modeled with deformable meshes in this way.

With OpenGL, you can, in fact, have the best of both scenarios by using vertex arrays. With vertex arrays, you can precalculate or modify your geometry on the fly, but do a bulk transfer of all the geometry data at one time. Basic vertex arrays can be almost as fast as display lists, but without the requirement that the geometry be static. It might also simply be more convenient to store your data in arrays for other reasons and thus also render directly from the same arrays (this approach could also potentially be more memory efficient).

Using vertex arrays in OpenGL involves four basic steps. First, you must assemble your geometry data in one or more arrays. You can do this algorithmically or perhaps by loading the data from a disk file. Second, you must tell OpenGL where the data is. When rendering is performed, OpenGL pulls the vertex data directly from the arrays you have specified. Third, you must explicitly tell OpenGL which arrays you are using. You can have separate arrays for vertices, normals, colors, and so on, and you must let OpenGL know which of these data sets you want to use. Finally, you execute the OpenGL commands to actually perform the rendering using your vertex data.

To demonstrate these four steps, we revisit an old sample from another chapter. We've rewritten the SMOOTHER sample from [Chapter 6](#) for the STARFIELD sample in this chapter. The STARFIELD sample creates three arrays that contain randomly initialized positions for stars in a starry sky. We then use vertex arrays to render directly from these arrays, bypassing the `glBegin/glEnd` mechanism entirely. [Figure 11.9](#) shows the output of the STARFIELD sample program, and [Listing 11.6](#) shows the important portions of the source code.

Listing 11.6. Setup and Rendering Code for the STARFIELD Sample

```

// Array of small stars
#define SMALL_STARS 150
GLTVector2 vSmallStars[SMALL_STARS];
#define MEDIUM_STARS 40
GLTVector2 vMediumStars[MEDIUM_STARS];
#define LARGE_STARS 15
GLTVector2 vLargeStars[LARGE_STARS];
#define SCREEN_X 800
#define SCREEN_Y 600
// This function does any needed initialization on the rendering
// context.
void SetupRC()
{
    int i;
    // Populate star list
    for(i = 0; i < SMALL_STARS; i++)
    {
        vSmallStars[i][0] = (GLfloat)(rand() % SCREEN_X);
        vSmallStars[i][1] = (GLfloat)(rand() % (SCREEN_Y - 100))+100.0f;
    }
    // Populate star list
    for(i = 0; i < MEDIUM_STARS; i++)
    {
        vMediumStars[i][0] = (GLfloat)(rand() % SCREEN_X * 10)/10.0f;
        vMediumStars[i][1] = (GLfloat)(rand() % (SCREEN_Y - 100))+100.0f;
    }
    // Populate star list
    for(i = 0; i < LARGE_STARS; i++)
    {
        vLargeStars[i][0] = (GLfloat)(rand() % SCREEN_X*10)/10.0f;
        vLargeStars[i][1] =
            (GLfloat)(rand() % (SCREEN_Y - 100)*10.0f)/ 10.0f +100.0f;
    }
    // Black background
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f );
    // Set drawing color to white
    glColor3f(0.0f, 0.0f, 0.0f);
}
///////////////////////////////
// Called to draw scene
void RenderScene(void)
{
    GLfloat x = 700.0f;      // Location and radius of moon
    GLfloat y = 500.0f;

```

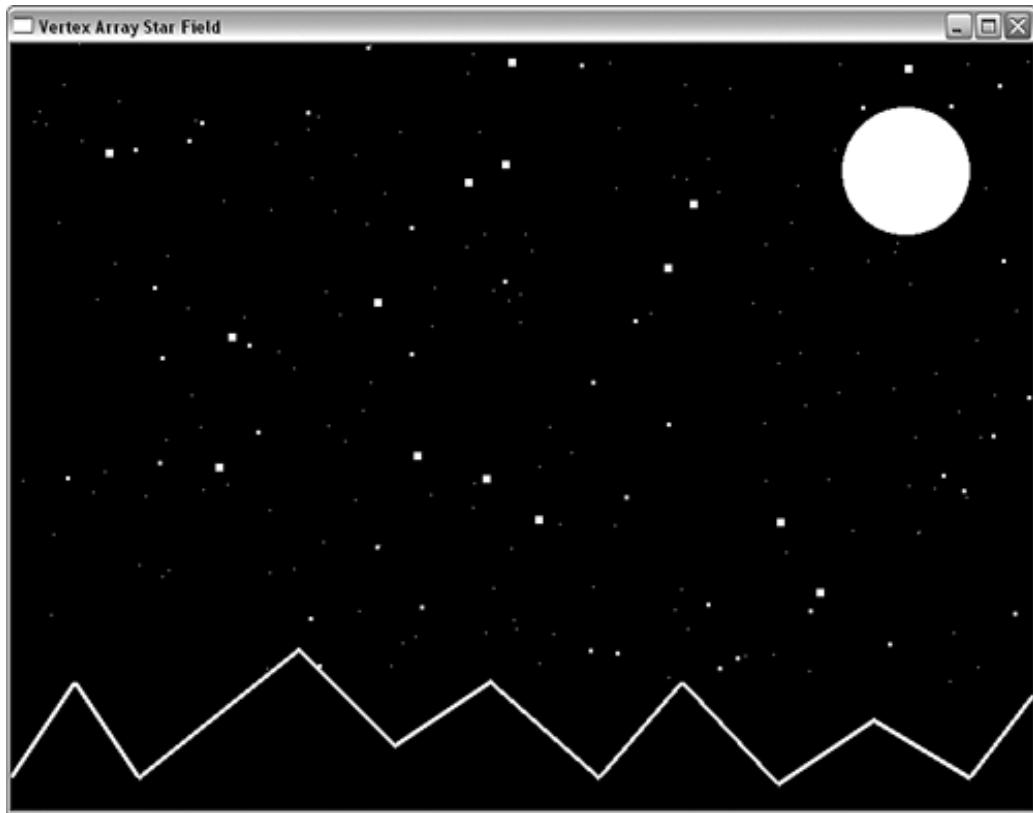
```

GLfloat r = 50.0f;
GLfloat angle = 0.0f; // Another looping variable
// Clear the window
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// Everything is white
glColor3f(1.0f, 1.0f, 1.0f);
// Using vertex arrays
glEnableClientState(GL_VERTEX_ARRAY);
// Draw small stars
glPointSize(1.0f);
// This code is no longer needed
// glBegin(GL_POINTS);
//     for(i = 0; i < SMALL_STARS; i++)
//         glVertex2fv(vSmallStars[i]);
// glEnd();
// Newer vertex array functionality
glVertexPointer(2, GL_FLOAT, 0, vSmallStars);
glDrawArrays(GL_POINTS, 0, SMALL_STARS);
// Draw medium sized stars
glPointSize(3.05f);
glVertexPointer(2, GL_FLOAT, 0, vMediumStars);
glDrawArrays(GL_POINTS, 0, MEDIUM_STARS);
// Draw largest stars
glPointSize(5.5f);
glVertexPointer(2, GL_FLOAT, 0, vLargeStars);
glDrawArrays(GL_POINTS, 0, LARGE_STARS);
// Draw the "moon"
glBegin(GL_TRIANGLE_FAN);
    glVertex2f(x, y);
    for(angle = 0; angle < 2.0f * 3.141592f; angle += 0.1f)
        glVertex2f(x + (float)cos(angle) * r, y + (float)sin(angle) * r);
    glVertex2f(x + r, y);
glEnd();

// Draw distant horizon
glLineWidth(3.5);
glBegin(GL_LINE_STRIP);
    glVertex2f(0.0f, 25.0f);
    glVertex2f(50.0f, 100.0f);
    glVertex2f(100.0f, 25.0f);
    glVertex2f(225.0f, 125.0f);
    glVertex2f(300.0f, 50.0f);
    glVertex2f(375.0f, 100.0f);
    glVertex2f(460.0f, 25.0f);
    glVertex2f(525.0f, 100.0f);
    glVertex2f(600.0f, 20.0f);
    glVertex2f(675.0f, 70.0f);
    glVertex2f(750.0f, 25.0f);
    glVertex2f(800.0f, 90.0f);
glEnd();
// Swap buffers
glutSwapBuffers();
}

```

Figure 11.9. Output from the STARFIELD program.



Loading the Geometry

The first prerequisite to using vertex arrays is that your geometry must be stored in arrays. In [Listing 11.6](#), you see three globally accessible arrays of two-dimensional vectors. They contain x and y coordinate locations for the three groups of stars:

```
// Array of small stars
#define SMALL_STARS 150
GLTVector2 vSmallStars[SMALL_STARS];
#define MEDIUM_STARS 40
GLTVector2 vMediumStars[MEDIUM_STARS];
#define LARGE_STARS 15
GLTVector2 vLargeStars[LARGE_STARS];
```

Recall that this sample program uses an orthographic projection and draws the stars as points at random screen locations. Each array is populated in the `SetupRC` function with a simple loop that picks random x and y values that fall within the portion of the window we want the stars to occupy. The following few lines from the listing show how just the small star list is populated:

```
// Populate star list
for(i = 0; i < SMALL_STARS; i++)
{
    vSmallStars[i][0] = (GLfloat)(rand() % SCREEN_X);
    vSmallStars[i][1] = (GLfloat)(rand() % (SCREEN_Y - 100))+100.0f;
}
```

Enabling Arrays

In the `RenderScene` function, we enable the use of an array of vertices with the following code:

```
// Using vertex arrays
glEnableClientState(GL_VERTEX_ARRAY);
```

This is the first new function for using vertex arrays, and it has a corresponding disabling function:

```
void glEnableClientState(GLenum array);
void glDisableClientState(GLenum array);
```

These functions accept the following constants, turning on and off the corresponding array usage: `GL_VERTEX_ARRAY`, `GL_COLOR_ARRAY`, `GL_SECONDARY_COLOR_ARRAY`, `GL_NORMAL_ARRAY`, `GL_FOG_COORDINATE_ARRAY`, `GL_TEXTURE_COORD_ARRAY`, and `GL_EDGE_FLAG_ARRAY`. For our `STARFIELD` example, we are sending down only a list of vertices. As you can see, you can also send down a corresponding array of normals, texture coordinates, colors, and so on.

Here's one question that commonly arises with the introduction of this function: Why did the OpenGL designers add a new `glEnableClientState` function instead of just sticking with `glEnable`. A good question. The reason has to do with how OpenGL is designed to operate. OpenGL was designed using a client/server model. The server is the graphics hardware, and the client is the host CPU and memory. On the PC, for example, the server would be the graphics card, and the client would be the PC's CPU and main memory. Because this state of enabled/disabled capability specifically applies to the client side of the picture, a new set of functions was derived.

Where's the Data?

Before we can actually use the vertex data, we must still tell OpenGL where to fetch the data. The following single line in the `STARFIELD` example does this:

```
glVertexPointer(2, GL_FLOAT, 0, vSmallStars);
```

Here, we find our next new function. The `glVertexPointer` function tells OpenGL where it can fetch the vertex data. There are also corresponding functions for the other types of vertex array data:

```
void glVertexPointer(GLint size, GLenum type, GLsizei stride,
                     const void *pointer);
void glColorPointer(GLint size, GLenum type, GLsizei stride,
                     const void *pointer);
void glTexCoordPointer(GLint size, GLenum type, GLsizei stride,
                      const void *pointer);
void glSecondaryColorPointer(GLint size, GLenum type, GLsizei stride,
                            const void *pointer);
void glNormalPointer(GLenum type, GLsizei stride, const void *pData);
void glFogCoordPointer(GLenum type, GLsizei stride, const void *pointer);
void glEdgeFlagPointer(GLenum type, GLsizei stride, const void *pointer);
```

These functions are all closely related and take nearly identical arguments. All but the normal, fog coordinate, and edge flag functions take a `size` argument first. This argument tells OpenGL the number of elements that make up the coordinate type. For example, vertices can consist of 2 (x, y), 3 (x,y,z), or 4 (x,y,z,w) components. Normals, however, are always three components, and fog coordinates and edge flags are always one component; thus, it would be redundant to specify the argument for these arrays.

The `type` parameter specifies the OpenGL data type for the array. Not all data types are valid for all vertex array specifications. [Table 11.1](#) lists the seven vertex array functions (index pointers are used for color index mode and are thus excluded here) and the valid data types that can be specified for the data elements.

Table 11.1. Valid Vertex Array Sizes and Data Types

Command	Elements	Valid Data Types
glColorPointer	3, 4	GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_DOUBLE
glEdgeFlagPointer	1	None specified (always GLboolean)
glFogCoordPointer	1	GL_FLOAT, GL_DOUBLE
glNormalPointer	3	GL_BYTE, GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
glSecondaryColorPointer	3	GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_DOUBLE
glTexCoordPointer	1, 2, 3, 4	GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
glVertexPointer	2, 3, 4	GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE

The *stride* parameter specifies the space in bytes between each array element. Typically, this value is just 0, and array elements have no data gaps between values. Finally, the *pointer* parameter is a pointer to the array of data. For arrays, this is simply the name of the array.

Draw!

Finally, we're ready to render using our vertex arrays. We can actually use the vertex arrays in two different ways. For illustration, first look at the nonvertex array method that simply loops through the array and passes a pointer to each array element to `glVertex`:

```
glBegin(GL_POINTS);
    for(i = 0; i < SMALL_STARS; i++)
        glVertex2fv(vSmallStars[i]);
glEnd();
```

Because OpenGL now knows about our vertex data, we can have OpenGL look up the vertex values for us with the following code:

```
glBegin(GL_POINTS);
    for(i = 0; i < SMALL_STARS; i++)
        glArrayElement(i);
glEnd();
```

The `glArrayElement` function looks up the corresponding array data from any arrays that have been enabled with `glEnableClientState`. If an array has been enabled, and a corresponding array has not been specified (`glVertexPointer`, `glColorPointer`, and so on), an illegal memory access will likely cause the program to crash. The advantage to using `glArrayElement` is that a single function call can now replace several function calls (`glNormal`, `glColor`, `glVertex`, and so forth) needed to specify all the data for a specific vertex. Sometimes you might want to jump around in the array in nonsequential order as well.

Most of the time, however, you will find that you are simply transferring a block of vertex data that needs to be traversed from beginning to end. In these cases (as is the case with the STARFIELD sample), OpenGL can transfer a single block of any enabled arrays with a single function call:

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

In this function, *mode* specifies the primitive to be rendered (one primitive batch per function call). The *first* parameter specifies where in the enabled arrays to begin retrieving data, and the *count* parameter tells how many array elements to retrieve. In the case of the STARFIELD example, we rendered the array of small stars as follows:

```
glDrawArrays(GL_POINTS, 0, SMALL_STARS);
```

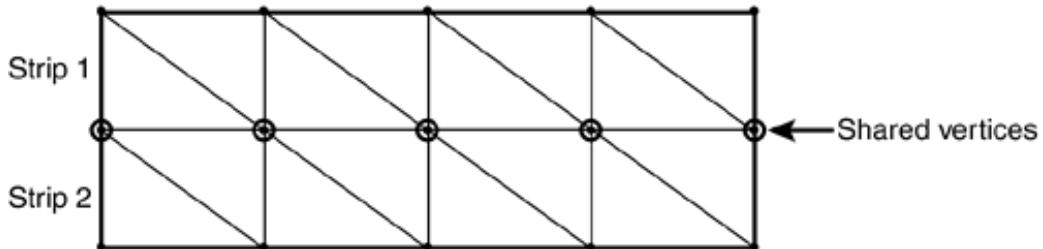
OpenGL implementations can optimize these block transfers, resulting in significant performance gains over multiple calls to the individual vertex functions such as `glVertex`, `glNormal`, and so forth.

Indexed Vertex Arrays

Indexed vertex arrays are vertex arrays that are not traversed in order from beginning to end, but are traversed in an order that is specified by a separate array of index values. This may seem a bit convoluted, but actually indexed vertex arrays can save memory and reduce transformation overhead. Under ideal conditions, they can actually be faster than display lists!

The reason for this extra efficiency is the array of vertices can be smaller than the array of indices. Adjoining primitives such as triangles can share vertices in ways not possible by just using triangle strips or fans. For example, using ordinary rendering methods or vertex arrays, there is no other mechanism to share a set of vertices between two adjacent triangle strips. [Figure 11.10](#) shows two triangle strips that share one edge. Although triangle strips make good use of shared vertices between triangles in the strip, there is no way to avoid the overhead of transforming the vertices shared between the two strips because each strip must be specified individually.

Figure 11.10. Two triangle strips in which the vertices share an edge.



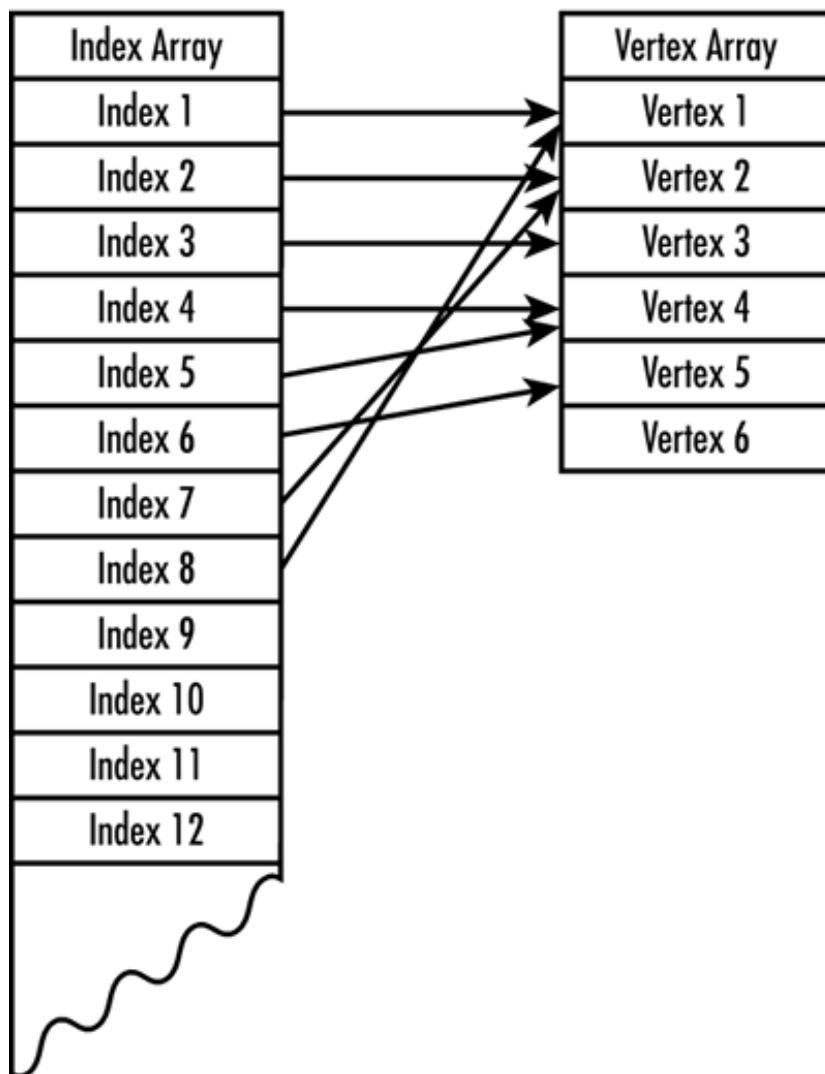
Now let's look at a simple example; then we'll look at a more complex model and examine the potential savings of using indexed arrays.

A Simple Cube

In the thread example, we repeated many normals and vertices. We can save a considerable amount of memory if we can reuse a normal or vertex in a vertex array without having to store it more than once. Not only is memory saved, but also a good OpenGL implementation is optimized to transform these vertices only once, saving valuable transformation time.

Instead of creating a vertex array containing all the vertices for a given geometric object, you can create an array containing only the unique vertices for the object. Then you can use another array of index values to specify the geometry. These indices reference the vertex values in the first array. [Figure 11.11](#) shows this relationship.

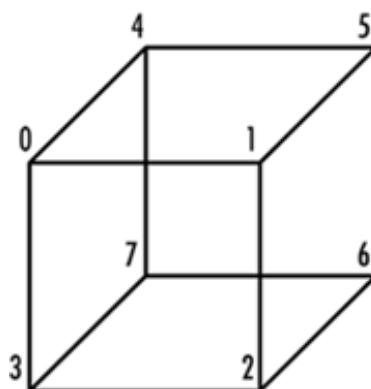
Figure 11.11. An index array referencing an array of unique vertices.



Each vertex consists of three floating-point values, but each index is only an integer value. A float and an integer are 4 bytes on most machines, which means you save 8 bytes for each reused vertex for the cost of 4 extra bytes for every vertex. For a small number of vertices, the savings might not be great; in fact, you might even use more memory using an indexed array than you would have by just repeating vertex information. For larger models, however, the savings can be substantial.

Figure 11.12 shows a cube with each vertex numbered. For our next sample program, CUBEDX, we create a cube using indexed vertex arrays.

Figure 11.12. A cube containing six unique numbered vertices.



[Listing 11.7](#) shows the code from the CUBEDX program to render the cube using indexed vertex arrays. The six unique vertices are in the `corners` array, and the indices are in the `indexes` array. In `RenderScene`, we set the polygon mode to `GL_LINE` so that the cube is wireframed.

Listing 11.7. Code from the CUBEDX Program to Use Indexed Vertex Arrays

```
// Array containing the six vertices of the cube
static GLfloat corners[] = { -25.0f, 25.0f, 25.0f, // 0 // Front of cube
                             25.0f, 25.0f, 25.0f, // 1
                             25.0f, -25.0f, 25.0f, // 2
                             -25.0f, -25.0f, 25.0f, // 3
                             -25.0f, 25.0f, -25.0f, // 4 // Back of cube
                             25.0f, 25.0f, -25.0f, // 5
                             25.0f, -25.0f, -25.0f, // 6
                             -25.0f, -25.0f, -25.0f }; // 7

// Array of indexes to create the cube
static GLubyte indexes[] = { 0, 1, 2, 3,           // Front Face
                            4, 5, 1, 0,           // Top Face
                            3, 2, 6, 7,           // Bottom Face
                            5, 4, 7, 6,           // Back Face
                            1, 5, 6, 2,           // Right Face
                            4, 0, 3, 7 };         // Left Face

// Rotation amounts
static GLfloat xRot = 0.0f;
static GLfloat yRot = 0.0f;

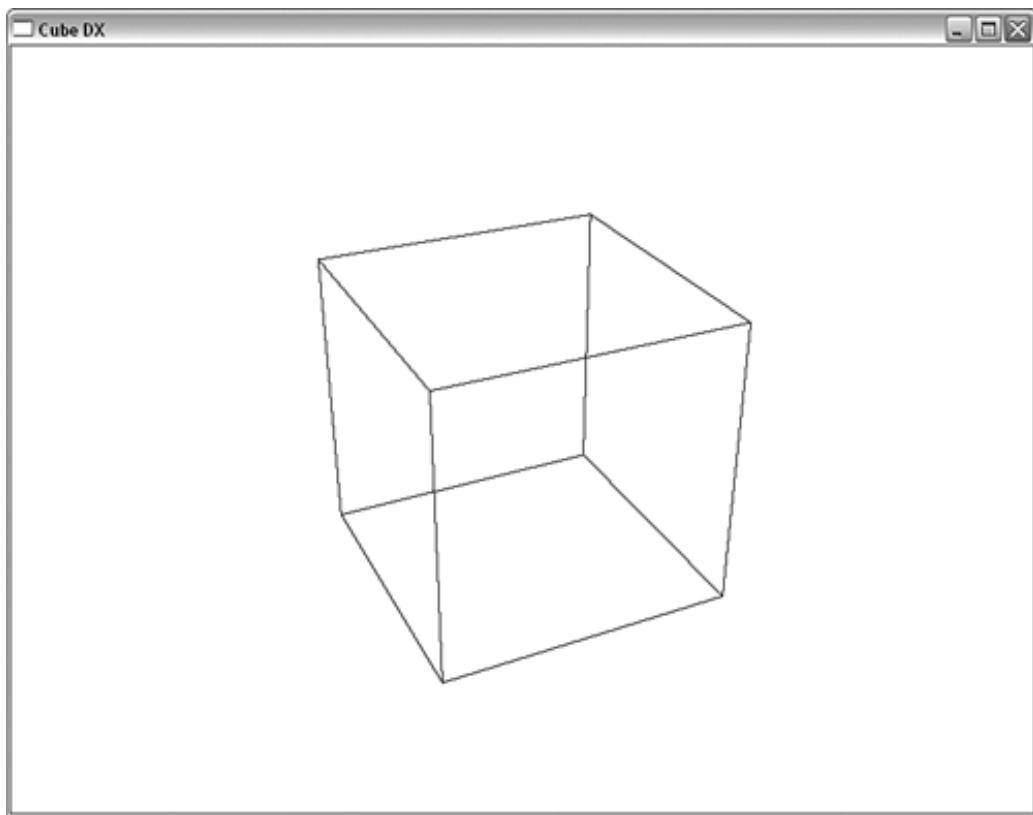
// Called to draw scene
void RenderScene(void)
{
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // Make the cube a wire frame
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    // Save the matrix state
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glTranslatef(0.0f, 0.0f, -200.0f);
    // Rotate about x and y axes
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot, 0.0f, 0.0f, 1.0f);
    // Enable and specify the vertex array
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(3, GL_FLOAT, 0, corners);
    // Using Drawarrays
    glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, indexes);
    glPopMatrix();
    // Swap buffers
    glutSwapBuffers();
}
```

OpenGL has native support for indexed vertex arrays, as shown in the `glDrawElements` function. The key line in [Listing 11.7](#) is

```
glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, indexes);
```

This line is much like the `glDrawArrays` function mentioned earlier, but now we are specifying an index array that determines the order in which the enabled vertex arrays are traversed. [Figure 11.13](#) shows the output from the program CUBEDX.

Figure 11.13. A wireframe cube drawn with an indexed vertex array.



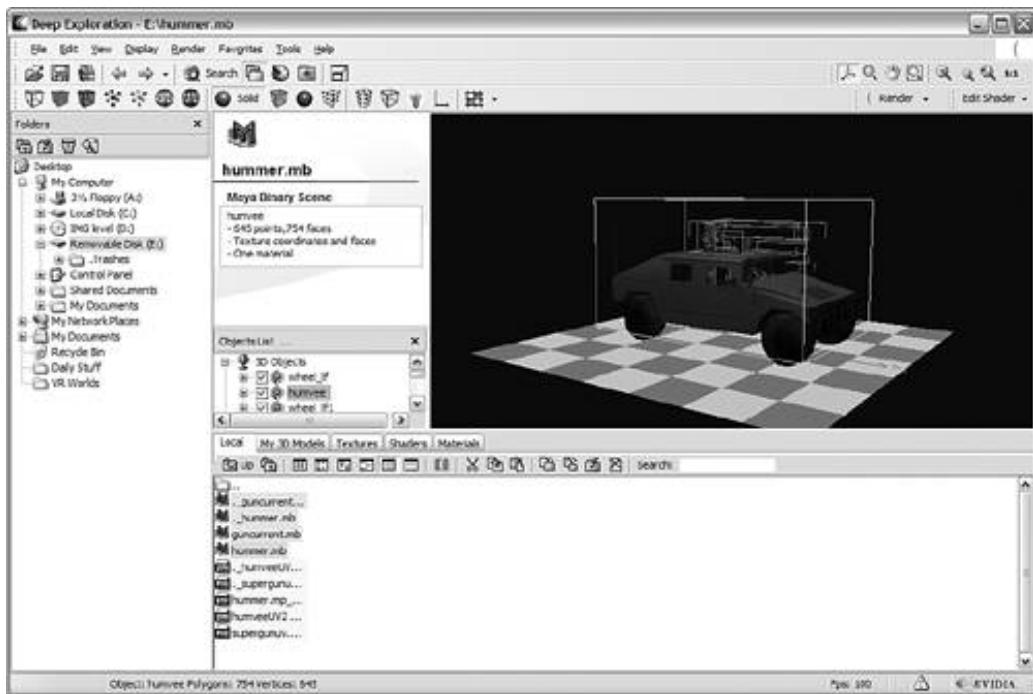
A variation on `glDrawElement` is the `glDrawRangeElements` function. This function is documented in the reference section and simply adds two parameters to specify the range of indices that will be valid. This hint can enable some OpenGL implementations to prefetch the vertex data, a potentially worthwhile performance optimization. A further enhancement is `glMultiDrawArrays`, which allows you to send multiple arrays of indices with a single function call.

One last vertex array function you'll find in the reference section is `glInterleavedArrays`. It allows you to combine several arrays into one aggregate array. There is no change to your access or traversal of the arrays, but the organization in memory can possibly enhance performance on some hardware implementations.

Getting Serious

With a few simple examples behind us, it's time to tackle a more sophisticated model with more vertex data. For this example, we use a model created by Full Sail student Stephen Carter, generously provided by the school's gaming department. We also use a product called Deep Exploration from Right Hemisphere that has a handy feature of exporting models as OpenGL code! A demo version of this product is available on the CD with this book. [Figure 11.14](#) shows Deep Exploration running and displaying the model that we will be working with.

Figure 11.14. Sample model to be rendered with OpenGL.



We had to modify the code output by Deep Exploration so that it would work with our GLUT framework and run on both the Macintosh and PC platforms. You can find the code that renders the model in the MODELTEST sample program. We do not include the entire program listing here because it is quite lengthy and mostly meaningless to human beings. It consists of a number of arrays representing 2,248 individual triangles (that's a lot of numbers to stare at!).

The approach taken with this tool is to produce the smallest possible amount of code to represent the given model. Deep Exploration has done an excellent job of compacting the data. There are 2,248 individual triangles, but using a clever indexing scheme, Deep Exploration has encoded this as only 1,254 individual vertices, 1,227 normals, and 2,141 texture coordinates. The following code shows the `DrawModel` function, which loops through the index set and sends OpenGL the texture, normal, and vertex coordinates for each individual triangle:

```
void DrawModel(void)
{
    int iFace, iPoint;
    glBegin(GL_TRIANGLES);
    for(iFace = 0; iFace < 2248; iFace++) // Each new triangle starts here
        for(iPoint = 0; iPoint < 3; iPoint++) // Each vertex specified here
    {
        // Lookup the texture value
        glTexCoord2fv(textures[face_indices[iFace][iPoint+6]]);
        // Lookup the normal value
        glNormal3fv(normals[face_indices[iFace][iPoint+3]]);
        // Lookup the vertex value
        glVertex3fv(vertices[face_indices[iFace][iPoint]]);
    }
    glEnd();
}
```

This approach is ideal when you must optimize the storage size of the model data—for example, to save memory in an embedded application, reduce storage space, or reduce bandwidth if the model must be transmitted over a network. However, for real-time applications where performance considerations can sometimes outweigh memory constraints, this code would perform quite poorly because once again you are back to square one, sending vertex data to OpenGL one vertex at a time.

The simplest and perhaps most obvious approach to speeding up this code is simply to place the

`DrawModel` function in a display list. Indeed, this is the approach we used in the MODELTEST program that renders this model. Let's look at the cost of this approach and compare it to rendering the same model with indexed vertex arrays.

Measuring the Cost

First, we calculate the amount of memory required to store the original compacted vertex data. We can do this simply by looking at the declarations of the data arrays and knowing how large the base data type is:

```
static short face_indices[2248][9] = { ...  
static GLfloat vertices [1254][3] = { ...  
static GLfloat normals [1227][3] = { ...  
static GLfloat textures [2141][2] = { ...
```

The memory for `face_indices` would be `sizeof(short)` x 2,248 x 9, which works out to 40,464 bytes. Similarly, we calculate the size of vertices, normals, and textures as 15,048, 14,724, and 17,128 bytes, respectively. This gives us a total memory footprint of 87,364 bytes or about 85KB.

But wait! When we draw the model into the display list, we copy all this data again into the display list, except that now we decompress our packed data so that many vertices are duplicated for adjacent triangles. We, in essence, undo all the work to optimize the storage of the geometry to draw it. We can't calculate exactly how much space the display list takes, but we can get a good estimate by calculating just the size of the geometry. There are 2,248 triangles. Each triangle has three vertices, each of which has a floating-point vertex (three floats), normal (three floats), and texture coordinate (two floats). Assuming four bytes for a float (`sizeof(float)`), we calculate this as follows:

$$2,248 \text{ (triangle)} \times 3 \text{ (vertices)} = 6,744 \text{ vertices.}$$

Each vertex has three components (x, y, z):

$$6,744 \times 3 = 20,232 \text{ floating-point values for geometry.}$$

Each vertex has a normal, meaning three more components:

$$6,744 \times 3 = 20,232 \text{ floating-point values for normals.}$$

Each vertex has a texture, meaning two more components:

$$6,744 \times 2 = 13,488 \text{ floating-point values for texture coordinates.}$$

This gives a total of 53,952 floats, at 4 bytes each = 215,808 bytes.

Total memory for the display list data and the original data is 311,736 bytes, just a tad more than 300KB. But don't forget the transformation cost—6,744 (2,248 x 3) vertices must be transformed by the OpenGL geometry pipeline. That's a lot of matrix multiplies!

Creating a Suitable Indexed Array

Just because the data in the MODELTEST sample is stored in arrays does not mean the data is ready to be used as any kind of OpenGL vertex array. In OpenGL, the vertex array, normal array, texture array, and any other arrays that you want to use must all be the same size. The reason is that all the array elements across arrays must be shared. For ordinary vertex arrays, as you march through the set of arrays, array element 0 from the vertex array must go with array element 0 from the normal array, and so on. For indexed arrays, we have the same limitation. Each index must address all the enabled arrays at the same corresponding array element.

For the sample program MODELIVA, we wrote a function that goes through the existing vertex array and reindexes the triangles so that all three arrays are the same size and all array elements correspond exactly one to another. The pertinent code is given in [Listing 11.8](#).

Listing 11.8. Code to Create a New Indexed Vertex Array

```
///////////
// These are hard coded for this particular example
GLushort uiIndexes[2248*3]; // Maximum number of indexes
GLfloat vVerts[2248*3][3]; // (Worst case scenario)
GLfloat vText[2248*3][2];
GLfloat vNorms[2248*3][3];
int iLastIndex = 0; // Number of indexes actually used
///////////
// Compare two floating point values and return true if they are
// close enough together to be considered the same.
inline bool IsSame(float x, float y, float epsilon)
{
    if(fabs(x-y) < epsilon)
        return true;
    return false;
}
///////////
// Goes through the arrays and looks for duplicate vertices
// that can be shared. This expands the original array somewhat
// and returns the number of true unique vertices that now
// populate the vVerts array.
int IndexTriangles(void)
{
    int iFace, iPoint, iMatch;
    float e = 0.000001; // How small a difference to equate
    // LOOP THROUGH all the faces
    int iIndexCount = 0;
    for(iFace = 0; iFace < 2248; iFace++)
    {
        for(iPoint = 0; iPoint < 3; iPoint++)
        {
            // Search for match
            for(iMatch = 0; iMatch < iLastIndex; iMatch++)
            {
                // If Vertex is the same...
                if(IsSame(vertices[face_indices[iFace][iPoint]][0],
                          vVerts[iMatch][0], e) &&
                   IsSame(vertices[face_indices[iFace][iPoint]][1],
                          vVerts[iMatch][1], e) &&
                   IsSame(vertices[face_indices[iFace][iPoint]][2],
                          vVerts[iMatch][2], e) &&
                   // AND the Normal is the same...
                   IsSame(normals[face_indices[iFace][iPoint+3]][0],
                          vNorms[iMatch][0], e) &&
                   IsSame(normals[face_indices[iFace][iPoint+3]][1],
                          vNorms[iMatch][1], e) &&
                   IsSame(normals[face_indices[iFace][iPoint+3]][2],
                          vNorms[iMatch][2], e) &&
                   // And Texture is the same...
                   IsSame(textures[face_indices[iFace][iPoint+6]][0],
                          vText[iMatch][0], e) &&
                   IsSame(textures[face_indices[iFace][iPoint+6]][1],
                          vText[iMatch][1], e))
                {
                    // Then add the index only
                    iIndexCount++;
                }
            }
        }
    }
    return iIndexCount;
}
```

```

        uiIndexes[iIndexCount] = iMatch;
        iIndexCount++;
        break;
    }
}
// No match found, add this vertex to the end of our list,
// and update the index array
if(iMatch == iLastIndex)
{
    // Add data and new index
    memcpy(vVerts[iMatch], vertices[face_indices[iFace][iPoint]],
           sizeof(float) * 3);
    memcpy(vNorms[iMatch], normals[face_indices[iFace][iPoint+3]],
           sizeof(float) * 3);
    memcpy(vText[iMatch], textures[face_indices[iFace][iPoint+6]],
           sizeof(float) * 2);
    uiIndexes[iIndexCount] = iLastIndex;
    iIndexCount++;
    iLastIndex++;
}
}
}
return iIndexCount;
}
///////////////////////////////
// Function to stitch the triangles together
// and draw the vehicle
void DrawModel(void)
{
    static int iIndexes = 0;
    char cBuffer[32];
    // The first time this is called, reindex the triangles. Report the results
    // in the window title
    if(iIndexes == 0)
    {
        iIndexes = IndexTriangles();
        sprintf(cBuffer, "Verts = %d Indexes = %d", iLastIndex, iIndexes);
        glutSetWindowTitle(cBuffer);
    }
    // Use vertices, normals, and texture coordinates
    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_NORMAL_ARRAY);
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);
    // Here's where the data is now
    glVertexPointer(3, GL_FLOAT, 0, vVerts);
    glNormalPointer(GL_FLOAT, 0, vNorms);
    glTexCoordPointer(2, GL_FLOAT, 0, vText);
    // Draw them
    glDrawElements(GL_TRIANGLES, iIndexes, GL_UNSIGNED_SHORT, uiIndexes);
}
}

```

First, we need to declare storage for our new indexed vertex array. Because we don't know ahead of time what our savings will be, or indeed whether there will be any savings, we allocate a block of arrays assuming the worst case scenario. If each vertex is unique, three floats for each vertex (which is 2,248 faces x 3 vertices each):

```

GLushort uiIndexes[2248*3];    // Maximum number of indexes
GLfloat vVerts[2248*3][3];    // (Worst case scenario)
GLfloat vText[2248*3][2];
GLfloat vNorms[2248*3][3];
int iLastIndex = 0;           // Number of indexes actually used

```

Looking for duplicates requires us to test many floating-point values for equality. This is usually a no-no because floats are notoriously noisy; their values can float around and vary slightly (forgive the pun!). You frequently can solve this problem by writing a special function that simply subtracts two floats and seeing whether the difference is small enough to call it even:

```
inline bool IsSame(float x, float y, float epsilon)
{
    if(fabs(x-y) < epsilon)
        return true;
    return false;
}
```

The `IndexTriangles` function is called only once; it goes through the existing array looking for duplicate vertices. For a vertex to be shared, all the vertex, normal, and texture coordinates must be exactly the same. If a match is found, that vertex is simply referenced in the new index array. If not, it is added to the end of the array of unique vertices and then referenced in the index array.

In the `DrawModel` function, the `IndexTriangles` function is called (only once), and the window caption reports how many unique vertices were identified and how many indices are needed to traverse the list of triangles:

```
// The first time this is called, reindex the triangles. Report the results
// in the window title
if(iIndexes == 0)
{
    iIndexes = IndexTriangles();
    sprintf(cBuffer, "Verts = %d Indexes = %d", iLastIndex, iIndexes);
    glutSetWindowTitle(cBuffer);
}
```

From here, rendering is straightforward. You enable the three sets of arrays:

```
// Use vertices, normals, and texture coordinates
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_NORMAL_ARRAY);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
```

Next, you tell OpenGL where the data is:

```
// Here's where the data is now
glVertexPointer(3, GL_FLOAT, 0, vVerts);
glNormalPointer(GL_FLOAT, 0, vNorms);
glTexCoordPointer(2, GL_FLOAT, 0, vText);
```

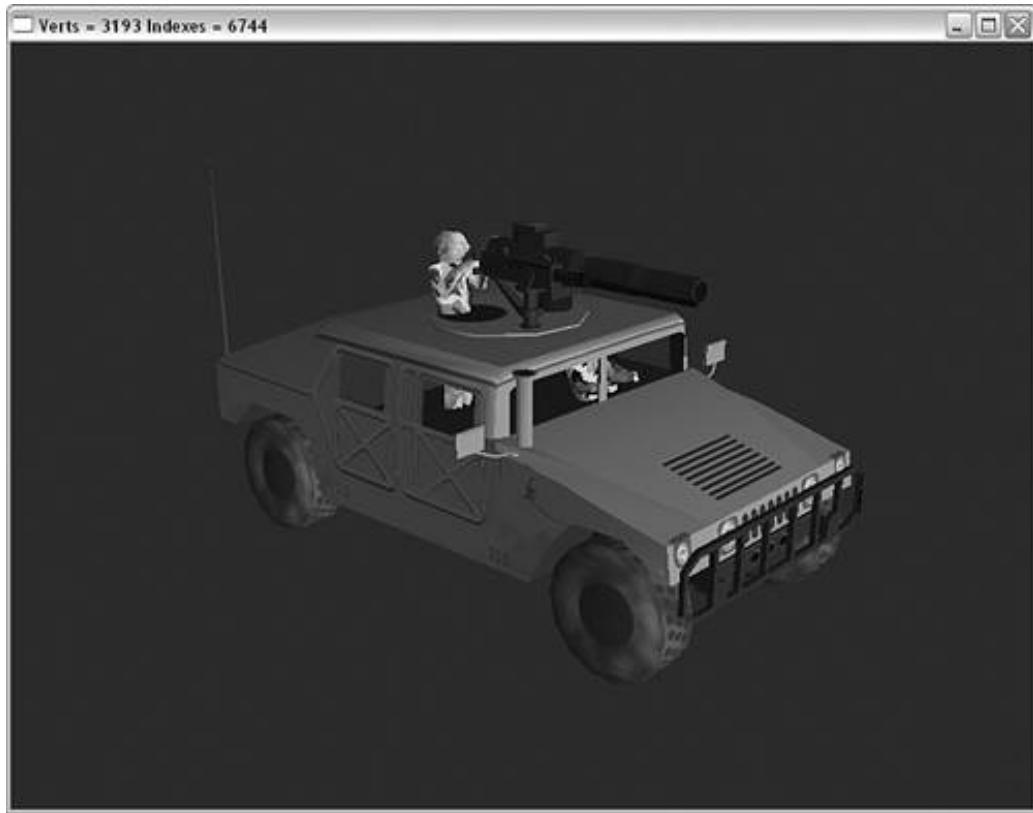
Then you fire off all the triangles:

```
// Draw them
glDrawElements(GL_TRIANGLES, iIndexes, GL_UNSIGNED_SHORT, uiIndexes);
```

Avoiding triangles for rendering might appear strange because they don't have the shared vertex advantage of strips and fans. However, with indexed vertex arrays, we can go back to large batches of triangles and still have the advantage of multiple shared vertex data—perhaps even beating the efficiency offered by strips and fans.

The final output of MODELIVA is shown in [Figure 11.15](#).

Figure 11.15. A model rendered with indexed vertex arrays.



Comparing the Cost

Now let's compare the cost of our two methods of rendering this model. From the output of the MODELIVA program, we see that 3,193 unique vertices were found; they can also share normals and texture coordinates. Rendering the entire model requires 6,744 indices still (this should come as no surprise!).

Each vertex has three components (x,y,z):

$$3,193 \text{ vertices} \times 3 = 9,579 \text{ floats}$$

Each normal also has three components:

$$3,193 \text{ normals} \times 3 = 9,579 \text{ floats}$$

Each texture coordinate has two components:

$$3,193 \text{ texture coordinates} \times 2 = 6,386 \text{ floats}$$

Multiplying each float by 4 bytes yields a memory overhead of 102,176 bytes. We still need to add in the index array of shorts. That's 6,744 elements times 2 bytes each = 13,488. This gives a grand total storage overhead of 115,664 bytes.

Table 11.2 shows these values side by side.

Table 11.2. Memory and Transformation Overhead for Three Rendering Methods

Rendering Mode	Memory	Vertices
Immediate Mode	95KB	6,744
Display List	300KB	6,744
Indexed Vertex Array	112KB	3,193

You can see that the immediate mode rendering used by the code output by Deep Exploration certainly has the smallest memory footprint. However, this transfers the geometry to OpenGL very slowly. If you put the immediate mode code into a display list, the geometry transfer takes place much faster, but the memory overhead for the model soars to three times that originally required. The indexed vertex array seems a good compromise at just more than twice the memory footprint, but less than half the transformation cost.

Of course, in this example, we actually allocated a much larger buffer to hold the maximum number of vertices that may have been required. In a production program, you might have tools that take this calculated indexed array and write it out to disk with a header that describes the required array dimensions. Reading this model back into the program then is a simple implementation of a basic model loader. The loaded model is then exactly in the format required by OpenGL.

Models with sharp edges and corners often have fewer vertices that are candidates for sharing. However, models with large smooth surface areas can stand to gain even more in terms of memory and transformation savings. With the added savings of less geometry to move through memory, and the corresponding savings in mathematical operations, indexed vertex arrays can sometimes dramatically outperform display lists, even for static geometry. For many real-time applications, indexed vertex arrays are often the method of choice for geometric rendering.

Summary

In this chapter, we slowed down the pace somewhat and just explained how to build a three-dimensional object, starting with using the OpenGL primitives to create simple 3D pieces and then assembling them into a larger and more complex object. Learning the API is the easy part, but your level of experience in assembling 3D objects and scenes will be what differentiates you from your peers. After you break down an object or scene into small and potentially reusable components, you can save building time by using display lists. You'll find many more functions for utilizing and managing display lists in the reference section.

The last half of the chapter was concerned not with how to organize your objects, but how to organize the geometry data used to construct these objects. By packing all the vertex data together in a single data structure (an array), you enable the OpenGL implementation to make potentially valuable performance optimizations. In addition, you can stream the data to disk and back, thus storing the geometry in a format that is ready for use in OpenGL. Although OpenGL does not have a "model format" as some higher level APIs do, the vertex array construct is certainly a good place to start if you want to build your own.

Generally, you can significantly speed up static geometry by using display lists, and you can use vertex arrays whenever you want dynamic geometry. Index vertex arrays, on the other hand, can potentially (but not always) give you the best of both worlds—flexible geometry data and highly efficient memory transfer and geometric processing. For many applications, vertex arrays are used almost exclusively. However, the old `glBegin/glEnd` construct still has many uses, besides allowing you to create display lists—any time the amount of geometry fluctuates dynamically from frame to frame, for example. There is little benefit to continually rebuilding a vertex array from scratch rather than letting the driver do the work with `glBegin/glEnd`.

Reference

glArrayElement

Purpose: Specifies an array element used to render a vertex.

Include File: `<gl.h>`

Syntax:

```
void glArrayElement(GLint index);
```

Description: You use this function with a `glBegin/glEnd` pair to specify vertex data. The indexed element from any enabled vertex arrays are passed to OpenGL as part of the primitive definition.

Parameters:

index `GLint`: The index of the array element to use.

Returns: None.

See Also: `glDrawArrays`, `glDrawElements`, `glDrawRangeElements`, `glInterleavedArrays`

glCallList

Purpose: Executes a display list.

Include File: `<gl.h>`

Syntax:

```
void glCallList(GLuint list);
```

Description: This function executes the display list identified by *list*. The OpenGL state machine is not restored after this function is called, so it is a good idea to call `glPushMatrix` beforehand and `glPopMatrix` afterward. Calls to `glCallList` can be nested. The `glGet` function with the `GL_MAX_LIST_NESTING` argument returns the maximum number of allowable nests. For Microsoft Windows, this value is 64.

Parameters:

list `GLuint`: Identifies the display list to be executed.

Returns: None.

See Also: `glCallLists`, `glDeleteLists`, `glGenLists`, `glNewList`

glCallLists

Purpose: Executes a list of display lists.

Include File: `<gl.h>`

Syntax:

```
void glCallLists(GLsizei n, GLenum type, const
→ GLvoid *lists);
```

Description: This function calls the display lists listed in the **lists* array sequentially. This array can be of nearly any data type. The result is converted or clamped to the nearest integer value to determine the actual index of the display list. Optionally, the list values can be offset by a value specified by the *glListBase* function.

Parameters:

n GLsizei: The number of elements in the array of display lists.

type GLenum: The data type of the array stored at **lists*. It can be any one of the following values: *GL_BYTE*, *GL_UNSIGNED_BYTE*, *GL_SHORT*, *GL_UNSIGNED_SHORT*, *GL_INT*, *GL_UNSIGNED_INT*, *GL_FLOAT*, *GL_2_BYTES*, *GL_3_BYTES*, and *GL_4_BYTES*.

**lists* GLvoid: An array of elements of the type specified in *type*. The data type is *void* to allow any of the preceding data types to be used.

Returns: None.

See Also: *glCallList*, *glDeleteLists*, *glGenLists*, *glListBase*, *glNewList*

glColorPointer

Purpose: Defines an array of color data for OpenGL vertex array functionality.

Include File: *<gl.h>*

Syntax:

```
void glColorPointer(GLint size, GLenum type,
→ GLsizei stride, const GLvoid *pointer);
```

Description: This function defines the location, organization, and type of data to be used for vertex color data when OpenGL is using the vertex array functions. The buffer pointed to by this function can contain dynamic data but must remain valid data. The data is read afresh from the vertex array buffer supplied here whenever OpenGL evaluates vertex arrays.

Parameters:

size GLint: The number of components per color. Valid values are 3 and 4.

type GLenum: The data type of the array. It can be any of the valid OpenGL data types for color component data: *GL_BYTE*, *GL_UNSIGNED_BYTE*, *GL_SHORT*, *GL_UNSIGNED_SHORT*, *GL_INT*, *GL_UNSIGNED_INT*, *GL_FLOAT*, and *GL_DOUBLE*.

stride GLsizei: The byte offset between colors in the array. A value of 0 indicates that the data is tightly packed.

pointer GLvoid*: A pointer that specifies the location of the beginning of the vertex array data.

Returns: None.

See Also: [glVertexPointer](#), [glNormalPointer](#), [glTexCoordPointer](#), [glEdgeFlagPointer](#), [glFogCoordPointer](#), [glInterleavedArrays](#)

glDeleteLists

Purpose: Deletes a continuous range of display lists.

Include File: `<gl.h>`

Syntax:

```
void glDeleteLists(GLuint list, GLsizei range);
```

Description: This function deletes a range of display lists. The range goes from an initial value and proceeds until the number of lists deleted as specified by `range` is completed. Deleting unused display lists can save considerable memory. Unused display lists in the range of those specified are ignored and do not cause an error.

Parameters:

`list` `GLuint`: The integer name of the first display list to delete.

`range` `GLsizei`: The number of display lists to be deleted following the initially specified list.

Returns: None.

See Also: [glCallList](#), [glCallLists](#), [glGenLists](#), [glIsList](#), [glNewList](#)

glDrawArrays

Purpose: Creates a sequence of primitives from any enabled vertex arrays.

Include File: `<gl.h>`

Syntax:

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

Description: This function enables you to render a series of primitives using the data in the currently enabled vertex arrays. The function takes the primitive type and processes all the vertices within the specified range.

Parameters:

`mode` `GLenum`: The kind of primitive to render. It can be any of the valid OpenGL primitive types: `GL_POINTS`, `GL_LINES`, `GL_LINE_LOOP`, `GL_LINE_STRIP`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_QUADS`, `GL_QUAD_STRIP`, and `GL_POLYGON`

`first` `GLint`: The first index of the enabled arrays to use.

`count` `GLsizei`: The number of indices to use.

Returns: None.**See Also:** [glDrawElements](#), [glDrawRangeElements](#), [glInterleavedArrays](#)

glDrawElements

Purpose: Renders primitives from array data, using an index into the array.**Include File:** `<gl.h>`**Syntax:**

```
void glDrawElements(GLenum mode, GLsizei count,
    ➔ GLenum type, GLvoid *pointer);
```

Description: Rather than traverse the array data sequentially, this function traverses an index array sequentially. This index array typically accesses the vertex data in a nonsequential and often repetitious way, allowing for shared vertex data.

Parameters:

mode `GLenum`: The primitive type to be rendered. It can be `GL_POINTS`, `GL_LINES`, `GL_LINE_LOOP`, `GL_LINE_STRIP`, `GL_TRIANGLES`, `GL_TRIANGLE_FAN`, `GL_TRIANGLE_STRIP`, `GL_QUAD`, `GL_QUAD_STRIP`, or `GL_POLYGON`.

count `GLsizei`: The byte offset between coordinates in the array. A value of 0 indicates that the data is tightly packed.

type `GLenum`: The type of data used in the index array. It can be any one of `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, or `GL_UNSIGNED_INT`.

pointer `GLvoid*`: A pointer that specifies the location of the index array.

Returns: None.**See Also:** [glArrayElement](#), [glDrawArrays](#), [glDrawRangeElements](#), [glDrawMultiRangeElements](#)

glDrawRangeElements

Purpose: Renders primitives from array data, using an index into the array and a specified range of valid index values.**Include File:** `<gl.h>`**Syntax:**

```
void glDrawRangeElements(GLenum mode, GLuint start
    ➔ , GLuint end,
                           GLsizei count,
    ➔ GLenum type, GLvoid *pointer);
```

Description: Rather than traverse the array data sequentially, this function traverses an index array sequentially. This index array typically accesses the vertex data in a nonsequential and often repetitious way, allowing for shared vertex data. In addition to this shared functionality with `glDrawElements`, this function takes a range of valid index values. Some OpenGL implementations can use this information to prefetch the vertex data for higher performance.

Parameters:

mode `GLenum`: The primitive type to be rendered. It can be `GL_POINTS`, `GL_LINES`, `GL_LINE_LOOP`, `GL_LINE_STRIP`, `GL_TRIANGLES`, `GL_TRIANGLE_FAN`, `GL_TRIANGLE_STRIP`, `GL_QUAD`, `GL_QUAD_STRIP`, or `GL_POLYGON`.

start `GLint`: The first index of the index range that will be used.

end `GLint`: The last index of the index range that will be used.

count `GLsizei`: The byte offset between coordinates in the array. A value of 0 indicates that the data is tightly packed.

type `GLenum`: The type of data used in the index array. It can be any one of `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, or `GL_UNSIGNED_INT`.

pointer `GLvoid*`: A pointer that specifies the location of the index array.

Returns: None.

See Also: `glArrayElement`, `glDrawArrays`, `glDrawElements`, `glDrawMultiRangeElements`

glEdgeFlagPointer

Purpose: Defines an array of edge flags for OpenGL vertex array functionality.

Include File: `<gl.h>`

Syntax:

```
void glEdgeFlagPointer(GLsizei stride, const
→ GLvoid *pointer);
```

Description: This function defines the location of data to be used for the edge flag array when OpenGL is using the vertex array functions. The buffer pointed to by this function can contain dynamic data but must remain valid data. The data is read afresh from the vertex array buffer supplied here whenever OpenGL evaluates vertex arrays. Note that there is no *type* argument as in the other vertex array pointer functions. The data type for edge flags must be `GLboolean`.

Parameters:

stride `GLsizei`: The byte offset between edge flags in the array. A value of 0 indicates that the data is tightly packed.

pointer `GLvoid*`: A pointer that specifies the location of the beginning of the vertex array data.

Returns: None.

See Also: `glColorPointer`, `glNormalPointer`, `glTexCoordPointer`, `glVertexPointer`, `glFogCoordPointer`, `glEdgeFlagPointer`, `glSecondaryColorPointer`

glEnableClientState/glDisableClientState**Purpose:** Specify the array type to enable or disable for use with OpenGL vertex arrays.**Include File:** `<gl.h>`**Syntax:**

```
void glEnableClientState(GLenum array);
void glDisableClientState(GLenum array);
```

Description: These functions tell OpenGL that you will or will not be specifying vertex arrays for geometry definitions. Each array type can be enabled or disabled individually. The use of vertex arrays does not preclude use of the normal `glVertex` family of functions. The specification of vertex arrays cannot be stored in a display list.**Parameters:**

array `GLenum`: The name of the array to enable or disable. Valid values are `GL_VERTEX_ARRAY`, `GL_COLOR_ARRAY`, `GL_SECONDARY_COLOR_ARRAY`, `GL_NORMAL_ARRAY`, `GL_FOG_COORDINATE_ARRAY`, `GL_TEXTURE_COORD_ARRAY`, and `GL_EDGE_FLAG_ARRAY`.

Returns: None.**See Also:** `glVertexPointer`, `glNormalPointer`, `glTexCoordPointer`, `glColorPointer`, `glEdgeFlagPointer`, `glSecondaryColorPointer`, `glFogCoordPointer`**glEndList****Purpose:** Delimits the end of a display list.**Include File:** `<gl.h>`**Syntax:**

```
void glEndList( void );
```

Description: Display lists are created by first calling `glNewList`. Thereafter, all OpenGL commands are compiled and placed in the display list. The `glEndList` function terminates the creation of this display list.**Returns:** None.**See Also:** `glCallList`, `glCallLists`, `glDeleteLists`, `glGenLists`, `glIsList`**glFogCoordPointer****Purpose:** Defines an array of fog coordinates for OpenGL vertex array functionality.**Include File:** `<gl.h>`

Syntax:

```
void glFogCoordPointer(GLenum type, GLsizei stride
→, const GLvoid *pointer);
```

Description: This function defines the location, organization, and type of data to be used for fog coordinates when OpenGL is using the vertex array functions. The buffer pointed to by this function can contain dynamic data but must remain valid data. The data is read afresh from the vertex array buffer supplied here whenever OpenGL evaluates vertex arrays.

Parameters:

type **GLenum**: The data type of the array. It can be any of the valid OpenGL data types for color component data: **GL_BYTE**, **GL_UNSIGNED_BYTE**, **GL_SHORT**, **GL_UNSIGNED_SHORT**, **GL_INT**, **GL_UNSIGNED_INT**, **GL_FLOAT**, and **GL_DOUBLE**.

stride **GLsizei**: The byte offset between colors in the array. A value of 0 indicates that the data is tightly packed.

pointer **GLvoid***: A pointer that specifies the location of the beginning of the vertex array data.

Returns: None.

See Also: [glColorPointer](#), [glSecondaryColorPointer](#), [glNormalPointer](#), [glTexCoordpointer](#), [glEdgeFlagPointer](#)

glGenLists

Purpose: Generates a continuous range of empty display lists.

Include File: `<gl.h>`

Syntax:

```
GLuint glGenLists(GLsizei range);
```

Description: This function creates a range of empty display lists. The number of lists generated depends on the value specified in *range*. The return value is then the first display list in this range of empty display lists. The purpose of this function is to reserve a range of display list values for future use.

Parameters:

range **GLsizei**: The number of empty display lists requested.

Returns: The first display list of the range requested. The display list values following the return value up to *range* – 1 are created empty.

See Also: [glCallList](#), [glCallLists](#), [glDeleteLists](#), [glNewList](#)

glInterleavedArrays

Purpose: Enables and disables multiple vertex arrays simultaneously and specifies an address that points to all the vertex data contained in one aggregate array.

Include File: `<gl.h>`

Syntax:

```
void glInterleavedArrays(GLenum format, GLsizei
→ stride, GLvoid *pointer);
```

Description: Similar to the `glXXXXPointer` functions, this function enables and disables several vertex arrays simultaneously. All the enabled arrays are interleaved together in one aggregate array. This functionality could be achieved by careful use of the `stride` parameter in the other vertex array functions, but this function saves several steps and can be optimized by the OpenGL implementation.

Parameters:

`format` `GLenum`: The packing format of the vertex data in the interleaved array. It can be any one of the values shown in [Table 11.3](#).

`stride` `GLsizei`: The byte offset between coordinates in the array. A value of 0 indicates that the data is tightly packed.

`pointer` `GLvoid*`: A pointer that specifies the location of the interleaved array.

Returns: None.

Table 11.3. Supported Interleaved Vertex Array Formats

Format	Details
<code>GL_V2F</code>	Two <code>GL_FLOAT</code> values for the vertex data.
<code>GL_V3F</code>	Three <code>GL_FLOAT</code> values for the vertex data.
<code>GL_C4UB_V2F</code>	Four <code>GL_UNSIGNED_BYTE</code> values for color data and two <code>GL_FLOAT</code> values for the vertex data.
<code>GL_C4UB_V3F</code>	Four <code>GL_UNSIGNED_BYTE</code> values for color data and three <code>GL_FLOAT</code> values for vertex data.
<code>GL_C3F_V3F</code>	Three <code>GL_FLOAT</code> values for color data and three <code>GL_FLOAT</code> values for vertex data.
<code>GL_N3F_V3F</code>	Three <code>GL_FLOAT</code> values for normal data and three <code>GL_FLOAT</code> values for vertex data.
<code>GL_C4F_N3F_V3F</code>	Four <code>GL_FLOAT</code> values for color data, three <code>GL_FLOAT</code> values for normal data, and three <code>GL_FLOAT</code> values for vertex data.
<code>GL_T2F_V3F</code>	Two <code>GL_FLOAT</code> values for texture coordinates, three <code>GL_FLOAT</code> values for vertex data.
<code>GL_T4F_V4F</code>	Four <code>GL_FLOAT</code> values for texture coordinates and four <code>GL_FLOAT</code> values for vertex data.

<code>GL_T2F_C4UB_V3F</code>	Two <code>GL_FLOAT</code> values for texture coordinates, four <code>GL_UNSIGNED_BYTE</code> values for color data, and three <code>GL_FLOAT</code> values for vertex data.
<code>GL_T2F_C3F_V3F</code>	Two <code>GL_FLOAT</code> values for texture data, three <code>GL_FLOAT</code> values for color data, and three <code>GL_FLOAT</code> values for vertex data.
<code>GL_T2F_N3F_V3F</code>	Two <code>GL_FLOAT</code> values for texture coordinates, three <code>GL_FLOAT</code> values for normals, and three <code>GL_FLOAT</code> values for vertex data.
<code>GL_T2F_C4F_N3F_V3F</code>	Two <code>GL_FLOAT</code> values for texture coordinates, four <code>GL_FLOAT</code> values for color data, three <code>GL_FLOAT</code> values for normals, and three <code>GL_FLOAT</code> values for vertex data.
<code>GL_T4F_C4F_N3F_V4F</code>	Four <code>GL_FLOAT</code> values for texture coordinates, four <code>GL_FLOAT</code> values for colors, three <code>GL_FLOAT</code> values for normals, and four <code>GL_FLOAT</code> for vertex data.

See Also: `glColorPointer`, `glEdgeFlagPointer`, `glSecondaryColorPointer`, `glFogCoordPointer`, `glNormalPointer`, `glTexCoordPointer`, `glVertexPointer`

glIsList

Purpose:

Tests for the existence of a display list.

Include File:

`<gl.h>`

Syntax:

```
GLboolean glIsList(GLuint list);
```

Description: This function enables you to find out whether a display list exists for a given identifier. You can use this function to test display list values before using them.

Parameters:

`list` `GLuint`: The value of a potential display list. This function tests this value to see whether a display list is defined for it.

Returns: `GL_TRUE` if the display list exists; otherwise, `GL_FALSE`.

See Also: `glCallList`, `glCallLists`, `glDeleteLists`, `glGenLists`, `glNewList`

glListBase

Purpose: Specifies an offset to be added to the list values specified in a call to `glCallLists`.

Include File: `<gl.h>`

Syntax:

```
void glListBase(GLuint base);
```

Description: The `glCallLists` function calls a series of display lists listed in an array. This function sets an offset value that can be added to each display list name for this function. By default, this value is 0. You can retrieve the current value by calling `glGet(GL_LIST_BASE)`.

Parameters:

base `GLuint`: Sets an integer offset value that will be added to display list names specified in calls to `glCallLists`. This value is 0 by default.

Returns: None.

See Also: `glCallLists`

glMultiDrawElements

Purpose: Renders primitives from multiple arrays of data, using an array of indices into the arrays.

Include File: `<gl.h>`

Syntax:

```
void glMultiDrawElements(GLenum mode, GLsizei
→ *count, GLenum type,
→ GLvoid
→ **indices, GLsizei primcount);
```

Description: This function has the effect of multiple calls to `glDrawElements`. For each set of primitives, an array is passed in the `count` parameter that specifies the number of array elements for each primitive batch. The `indices` array contains an array of arrays; each array is the corresponding element array for each primitive batch.

Parameters:

mode `GLenum`: The primitive type to be rendered. It can be `GL_POINTS`, `GL_LINES`, `GL_LINE_LOOP`, `GL_LINE_STRIP`, `GL_TRIANGLES`, `GL_TRIANGLE_FAN`, `GL_TRIANGLE_STRIP`, `GL_QUAD`, `GL_QUAD_STRIP`, or `GL_POLYGON`.

count `GLsizei*`: An array of the number of vertices contained in each array of elements.

type `GLenum`: The type of data used in the index array. It can be any one of `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, or `GL_UNSIGNED_INT`.

indices `GLvoid**`: An array of pointers to lists of array elements.

primcount `GLsizei`: The number of arrays of elements contained by the `count` and `indices` arrays.

Returns: None.

See Also: `glDrawElements`, `glDrawRangeElements`

glNewList**Purpose:** Begins the creation or replacement of a display list.**Include File:** `<gl.h>`**Syntax:**`void glNewList(GLuint list, GLenum mode);`

Description: A display list is a group of OpenGL commands that are stored for execution on command. You can use display lists to speed up drawings that are computationally intensive or that require data to be read from a disk. The `glNewList` function begins a display list with an identifier specified by the integer `list` parameter. The display list identifier is used by `glCallList` and `glCallLists` to refer to the display list. If it's not unique, a previous display list may be overwritten. You can use `glGenLists` to reserve a range of display list names and `glIsList` to test a display list identifier before using it. Display lists can be compiled only or compiled and executed. After `glNewList` is called, all OpenGL commands are stored in the display list in the order they were issued until `glEndList` is called. The following commands are executed when called and are never stored in the display list itself: `glIsList`, `glGenLists`, `glDeleteLists`, `glFeedbackBuffer`, `glSelectBuffer`, `glRenderMode`, `glReadPixels`, `glPixelStore`, `glFlush`, `glFinish`, `glIsEnabled`, and `glGet`.

Parameters:

`list` `GLuint`: The numerical name of the display list. If the display list already exists, it is replaced by the new display list.

`mode` `GLenum`: Display lists may be compiled and executed later or compiled and executed simultaneously. Specify `GL_COMPILE` to only compile the display list or `GL_COMPILE_AND_EXECUTE` to execute the display list as it is being compiled.

Returns: None.**See Also:** `glCallList`, `glCallLists`, `glDeleteLists`, `glGenLists`, `glIsList`**glNormalPointer****Purpose:** Defines an array of normals for OpenGL vertex array functionality.**Include File:** `<gl.h>`**Syntax:**

```
void glNormalPointer(GLenum type, GLsizei stride,
    const GLvoid *pointer);
```

Description: This function defines the location, organization, and type of data to be used for vertex normals when OpenGL is using the vertex array functions. The buffer pointed to by this function can contain dynamic data but must remain valid data. The data is read afresh from the vertex array buffer supplied here whenever OpenGL evaluates vertex arrays.

Parameters:

type **GLenum:** The data type of the array. It can be any of the valid OpenGL data types for vertex normals: **GL_BYTE**, **GL_SHORT**, **GL_INT**, **GL_FLOAT**, and **GL_DOUBLE**.

stride **GLsizei:** The byte offset between normals in the array. A value of 0 indicates that the data is tightly packed.

pointer **GLvoid*:** A pointer that specifies the location of the beginning of the vertex normal array data.

Returns: None.

See Also: [glColorPointer](#), [glVertexPointer](#), [glTexCoordPointer](#), [glEdgeFlagPointer](#), [glInterleavedArrays](#), [glSecondaryColorPointer](#)

glSecondaryColorPointer

Purpose: Defines an array of secondary color data for OpenGL vertex array functionality.

Include File: `<gl.h>`

Syntax:

```
void glSecondaryColorPointer(GLint size, GLenum
→ type, GLsizei stride,
                           const GLvoid
→ *pointer);
```

Description: This function defines the location, organization, and type of data to be used for vertex secondary color data when OpenGL is using the vertex array functions. The buffer pointed to by this function can contain dynamic data but must remain valid data. The data is read afresh from the vertex array buffer supplied here whenever OpenGL evaluates vertex arrays.

Parameters:

size **GLint:** The number of components per color. The only valid value is 3.

type **GLenum:** The data type of the array. It can be any of the valid OpenGL data types for color component data: **GL_BYTE**, **GL_UNSIGNED_BYTE**, **GL_SHORT**, **GL_UNSIGNED_SHORT**, **GL_INT**, **GL_UNSIGNED_INT**, **GL_FLOAT**, and **GL_DOUBLE**.

stride **GLsizei:** The byte offset between colors in the array. A value of 0 indicates that the data is tightly packed.

pointer **GLvoid*:** A pointer that specifies the location of the beginning of the vertex array data.

Returns: None.

See Also: [glColorPointer](#), [glFogCoordPointer](#), [glNormalPointer](#), [glTexCoordPointer](#), [glEdgeFlagPointer](#)

glTexCoordPointer**Purpose:** Defines an array of texture coordinates for OpenGL vertex array functionality.**Include File:** `<gl.h>`**Syntax:**

```
void glTexCoordPointer(GLint size, GLenum type,
→ GLsizei stride,
                    const GLvoid *pointer);
```

Description: This function defines the location, organization, and type of data to be used for texture coordinates when OpenGL is using the vertex array functions. The buffer pointed to by this function can contain dynamic data but must remain valid data. The data is read afresh from the vertex array buffer supplied here whenever OpenGL evaluates vertex arrays.**Parameters:**

size **GLint**: The number of coordinates per array element. Valid values are 1, 2, 3, and 4.

type **GLenum**: The data type of the array. It can be any of the valid OpenGL data types for texture coordinates: **GL_SHORT**, **GL_INT**, **GL_FLOAT**, and **GL_DOUBLE**.

stride **GLsizei**: The byte offset between coordinates in the array. A value of 0 indicates that the data is tightly packed.

pointer **GLvoid***: A pointer that specifies the location of the beginning of the vertex array data.

Returns: None.**See Also:** `glColorPointer`, `glNormalPointer`, `glSecondaryColorPointer`, `glVertexPointer`, `glEdgeFlagPointer`, `glInterleavedArrays`**glVertexPointer****Purpose:** Defines an array of vertex data for OpenGL vertex array functionality.**Include File:** `<gl.h>`**Syntax:**

```
void glVertexPointer(GLint size, GLenum type,
→ GLsizei stride,
                    const GLvoid *pointer);
```

Description: This function defines the location, organization, and type of data to be used for vertex data when OpenGL is using the vertex array functions. The buffer pointed to by this function can contain dynamic data but must remain valid data. The data is read afresh from the vertex array buffer supplied here whenever OpenGL evaluates vertex arrays.

Parameters:

<i>size</i>	<code>GLint</code> : The number of vertices per coordinate. Valid values are 2, 3, and 4.
<i>type</i>	<code>GLenum</code> : The data type of the array. It can be any of the valid OpenGL data types for vertex data: <code>GL_SHORT</code> , <code>GL_INT</code> , <code>GL_FLOAT</code> , and <code>GL_DOUBLE</code> .
<i>stride</i>	<code>GLsizei</code> : The byte offset between vertices in the array. A value of 0 indicates that the data is tightly packed.
<i>pointer</i>	<code>GLvoid*</code> : A pointer that specifies the location of the beginning of the vertex array data.
Returns:	None.
See Also:	<code>glColorPointer</code> , <code>glNormalPointer</code> , <code>glSecondaryColorPointer</code> , <code>glTexCoordPointer</code> , <code>glEdgeFlagPointer</code> , <code>glInterleavedArrays</code>

Chapter 12. Interactive Graphics

by Richard S. Wright, Jr.

WHAT YOU'LL LEARN IN THIS CHAPTER:

How To	Functions You'll Use
Assign OpenGL selection names to primitives or groups of primitives	<code>glInitNames</code> / <code>glPushName</code> / <code>glPopName</code>
Use selection to determine which objects are under the mouse	<code>glSelectBuffer</code> / <code>glRenderMode</code>
Use feedback to get information about where objects are drawn	<code>glFeedbackBuffer</code> / <code>gluPickMatrix</code>

Thus far, you have learned to create some sophisticated 3D graphics using OpenGL, and many applications do no more than generate these scenes. But many graphics applications (notably, games, CAD, 3D modeling, and so on) require more interaction with the scene itself. In addition to menus and dialog boxes, often you need to provide a way for the user to interact with a graphical scene. Typically, this interaction usually happens with a mouse.

Selection, a powerful feature of OpenGL, allows you to take a mouse click at some position over a window and determine which of your objects are beneath it. The act of selecting a specific object on the screen is called *picking*. With OpenGL's selection feature, you can specify a viewing volume and determine which objects fall within that viewing volume. A powerful utility function, `gluPickMatrix`, produces a matrix for you, based purely on screen coordinates and the pixel dimensions you specify; you use this matrix to create a smaller viewing volume placed beneath the mouse cursor. Then you use selection to test this viewing volume to see which objects are contained by it.

Feedback allows you to get information from OpenGL about how your vertices are transformed and illuminated when they are drawn to the frame buffer. You can use this information to transmit rendering results over a network, send them to a plotter, or add other graphics (say with GDI, for Windows programmers) to your OpenGL scene that appear to interact with the OpenGL objects. Feedback does not serve the same purpose as selection, but the mode of operation is similar and they can work productively together. You'll see this teamwork later in the `SELECT` sample program.

Selection

Selection is actually a rendering mode, but in selection mode, no pixels are actually copied to the frame buffer. Instead, primitives that are drawn within the viewing volume (and thus would normally appear in the frame buffer) produce hit records in a selection buffer. This buffer, unlike other OpenGL buffers, is just an array of integer values.

You must set up this selection buffer in advance and name your primitives or groups of primitives (your objects or models) so they can be identified in the selection buffer. You then parse the selection buffer to determine which objects intersected the viewing volume. One potential use for this capability is for visibility determination. Named objects that do not appear in the selection buffer fell outside the viewing volume and would not have been drawn in render mode. Although selection mode is fast enough for object picking, using it for general-purpose frustum-culling performs significantly slower than any of the techniques we discussed in [Chapter 11](#), "It's All About the Pipeline: Faster Geometry Throughput." Typically, you modify the viewing volume before entering selection mode and call your drawing code to determine which objects are in some restricted area of your scene. For picking, you specify a viewing volume that corresponds to the mouse pointer and then check which named objects are rendered beneath the mouse.

Naming Your Primitives

You can name every single primitive used to render your scene of objects, but doing so is rarely useful. More often, you name groups of primitives, thus creating names for the specific objects or pieces of objects in your scene. Object names, like display list names, are nothing more than unsigned integers.

The names list is maintained on the name stack. After you initialize the name stack, you can push names on the stack or simply replace the name currently on top of the stack. When a hit occurs during selection, all the names currently on the name stack are appended to the end of the selection buffer. Thus, a single hit can return more than one name if needed.

For our first example, we keep things simple. We create a simplified (and not-to-scale) model of the inner planets of the solar system. When the left mouse button is down, we display a message box describing which planet was clicked. [Listing 12.1](#) shows some of the rendering code for our sample program, PLANETS. We have created macro definitions for the sun, Mercury, Venus, Earth, and Mars.

Listing 12.1. Naming the Sun and Planets in the PLANETS Program

```
///////////
// Define object names
#define SUN      1
#define MERCURY  2
#define VENUS    3
#define EARTH    4
#define MARS    5
///////////
// Called to draw scene
void RenderScene(void)
{
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // Save the matrix state and do the rotations
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    // Translate the whole scene out and into view
    glTranslatef(0.0f, 0.0f, -300.0f);
    // Initialize the names stack
    glInitNames();
    glPushName(0);
```

```

// Name and draw the Sun
glColor3f(1.0f, 1.0f, 0.0f);
glLoadName(SUN);
DrawSphere(15.0f);
// Draw Mercury
glColor3f(0.5f, 0.0f, 0.0f);
glPushMatrix();
    glTranslatef(24.0f, 0.0f, 0.0f);
    glLoadName(MERCURY);
    DrawSphere(2.0f);
glPopMatrix();
// Draw Venus
    glColor3f(0.5f, 0.5f, 1.0f);
glPushMatrix();
    glTranslatef(60.0f, 0.0f, 0.0f);
    glLoadName(VENUS);
    DrawSphere(4.0f);
glPopMatrix();
// Draw the Earth
    glColor3f(0.0f, 0.0f, 1.0f);
glPushMatrix();
    glTranslatef(100.0f, 0.0f, 0.0f);
    glLoadName(EARTH);
    DrawSphere(8.0f);
glPopMatrix();
// Draw Mars
    glColor3f(1.0f, 0.0f, 0.0f);
glPushMatrix();
    glTranslatef(150.0f, 0.0f, 0.0f);
    glLoadName(MARS);
    DrawSphere(4.0f);
glPopMatrix();
// Restore the matrix state
glPopMatrix();    // Modelview matrix
glutSwapBuffers();
}

```

In PLANETS, the function initializes and clears the name stack, and `glPushName` pushes 0 on the stack to put at least one entry on the stack. For the sun and each planet, we call `glLoadName` to name the object or objects about to be drawn. This name, in the form of an unsigned integer, is not pushed on the name stack but rather replaces the current name on top of the stack. Later, we discuss keeping an actual stack of names. For now, we just replace the top name of the name stack each time we draw an object (the sun or a particular planet).

Working with Selection Mode

As mentioned previously, OpenGL can operate in three different rendering modes. The default mode is `GL_RENDER`, in which all the drawing actually occurs onscreen. To use selection, we must change the rendering mode to selection by calling the OpenGL function:

```
glRenderMode(GL_SELECTION);
```

When we actually want to draw again, we use the following call to place OpenGL back in rendering mode:

```
glRenderMode(GL_RENDER);
```

The third rendering mode is `GL_FEEDBACK`, discussed later in this chapter.

The naming code in [Listing 12.1](#) has no effect unless we first switch the rendering mode to selection mode. Most often, you use the same function to render the scene in both `GL_RENDER` mode and `GL_SELECTION` mode, as we have done here.

[Listing 12.2](#) provides the GLUT callback code triggered by the clicking of the left mouse button. This code checks for a left button click and then forwards the mouse coordinates to `ProcessSelection`, which processes the mouse click for this example.

Listing 12.2. Code That Responds to the Left Mouse Button Click

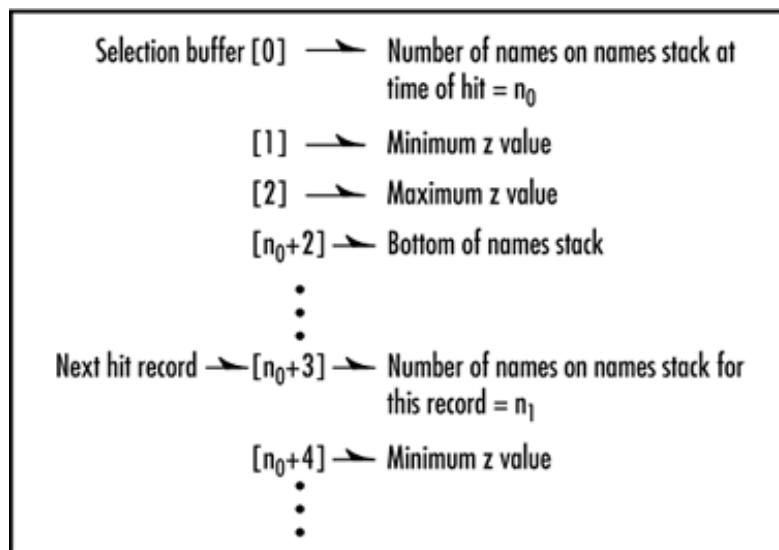
```
///////////////////////////////
// Process the mouse click
void MouseCallback(int button, int state, int x, int y)
{
    if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
        ProcessSelection(x, y);
}
```

The Selection Buffer

The selection buffer is filled with hit records during the rendering process. A hit record is generated whenever a primitive or collection of primitives is rendered that would have been contained in the viewing volume. Under normal conditions, this is simply anything that would have appeared onscreen.

The selection buffer is an array of unsigned integers, and each hit record occupies at least four elements of the array. The first array index contains the number of names that are on the name stack when the hit occurs. For the PLANETS example ([Listing 12.1](#)), this is always 1. The next two entries contain the minimum and maximum window z coordinates of all the vertices contained by the viewing volume since the last hit record. This value, which ranges from [0,1], is scaled to the maximum size of an unsigned integer for storage in the selection buffer. The fourth entry is the bottom of the name stack. If more than one name appears on the name stack (indicated by the first index element), they follow the fourth element. This pattern, illustrated in [Figure 12.1](#), is then repeated for all the hit records contained in the selection buffer. We explain why this pattern can be useful when we discuss picking.

Figure 12.1. Hit record for the selection buffer.



The format of the selection buffer gives you no way of knowing how many hit records you need to parse. The selection buffer is not actually filled until you switch the rendering mode back to `GL_RENDER`. When you do this with the `glRenderMode` function, the return value is the number of hit records copied.

[Listing 12.3](#) shows the processing function called when a mouse click occurs for the PLANETS sample program. It shows the selection buffer being allocated and specified with `glSelectBuffer`. This function takes two arguments: the length of the buffer and a pointer to the buffer itself.

Listing 12.3. Function to Process the Mouse Click

```
////////////////////////////////////////////////////////////////
// Process the selection, which is triggered by a right mouse
// click at (xPos, yPos)
#define BUFFER_LENGTH 64
void ProcessSelection(int xPos, int yPos)
{
    GLfloat fAspect;
    // Space for selection buffer
    static GLuint selectBuff[BUFFER_LENGTH];
    // Hit counter and viewport storage
    GLint hits, viewport[4];
    // Setup selection buffer
    glSelectBuffer(BUFFER_LENGTH, selectBuff);
    // Get the viewport
    glGetIntegerv(GL_VIEWPORT, viewport);
    // Switch to projection and save the matrix
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    // Change render mode
    glRenderMode(GL_SELECT);
    // Establish new clipping volume to be unit cube around
    // mouse cursor point (xPos, yPos) and extending two pixels
    // in the vertical and horizontal direction
    glLoadIdentity();
    gluPickMatrix(xPos, viewport[3] - yPos, 2, 2, viewport);
    // Apply perspective matrix
    fAspect = (float)viewport[2] / (float)viewport[3];
    gluPerspective(45.0f, fAspect, 1.0, 425.0);
    // Draw the scene
    RenderScene();
    // Collect the hits
    hits = glRenderMode(GL_RENDER);
    // If a single hit occurred, display the info.
    if(hits == 1)
        ProcessPlanet(selectBuff[3]);
    // Restore the projection matrix
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    // Go back to modelview for normal rendering
    glMatrixMode(GL_MODELVIEW);
}
```

Picking

Picking occurs when you use the mouse position to create and use a modified viewing volume during selection. When you create a smaller viewing volume positioned in your scene under the mouse position, only objects that would be drawn within that viewing volume generate hit records. By examining the selection buffer, you can then see which objects, if any, were clicked on by the

mouse.

The `gluPickMatrix` function is a handy utility that creates a matrix describing the new viewing volume:

```
void gluPickMatrix(GLdouble x, GLdouble y, GLdouble width,
                   GLdouble height, GLint viewport[4]);
```

The `x` and `y` parameters are the center of the desired viewing volume in OpenGL window coordinates. You can plug in the mouse position here, and the viewing volume will be centered directly underneath the mouse. The `width` and `height` parameters then specify the dimensions of the viewing volume in window pixels. For clicks near an object, use a large value; for clicks next to the object or directly on the object, use a smaller value. The `viewport` array contains the window coordinates of the currently defined viewport. You can easily obtain this information by calling

```
glGetIntegerv(GL_VIEWPORT, viewport);
```

Remember, as discussed in [Chapter 2](#), "Using OpenGL," that OpenGL window coordinates are the opposite of most systems' window coordinates with respect to the way pixels are counted vertically. Note in [Listing 12.3](#), we subtract the mouse y coordinate from the viewport's height. This yields the proper vertical window coordinate for OpenGL:

```
gluPickMatrix(xPos, viewport[3] - yPos, 2, 2, viewport);
```

To use `gluPickMatrix`, you should first save the current projection matrix state (thus saving the current viewing volume). Then call `glLoadIdentity` to create a unit-viewing volume. Calling `gluPickMatrix` then translates this viewing volume to the correct location. Finally, you must apply any further perspective projections you may have applied to your original scene; otherwise, you won't get a true mapping. Here's how it's done for the PLANETS example (from [Listing 12.3](#)):

```
// Switch to projection and save the matrix
glMatrixMode(GL_PROJECTION);
glPushMatrix();
// Change render mode
glRenderMode(GL_SELECT);
// Establish new clipping volume to be unit cube around
// mouse cursor point (xPos, yPos) and extending two pixels
// in the vertical and horizontal direction
glLoadIdentity();
gluPickMatrix(xPos, viewport[3] - yPos, 2, 2, viewport);
// Apply perspective matrix
fAspect = (float)viewport[2] / (float)viewport[3];
gluPerspective(45.0f, fAspect, 1.0, 425.0);
// Draw the scene
RenderScene();
// Collect the hits
hits = glRenderMode(GL_RENDER);
```

In this segment, the viewing volume is saved first. Then the selection mode is entered, the viewing volume is modified to include only the area beneath the mouse cursor, and the scene is redrawn by calling `RenderScene`. After the scene is rendered, we call `glRenderMode` again to place OpenGL back into normal rendering mode and get a count of generated hit records.

In the next segment, if a hit occurred (for this example, there is either one hit or none), we pass the entry in the selection buffer that contains the name of the object selected or our `ProcessPlanet` function. Finally, we restore the projection matrix (thus, the old viewing volume is restored) and switch the active matrix stack back to the modelview matrix, which is usually the default:

```

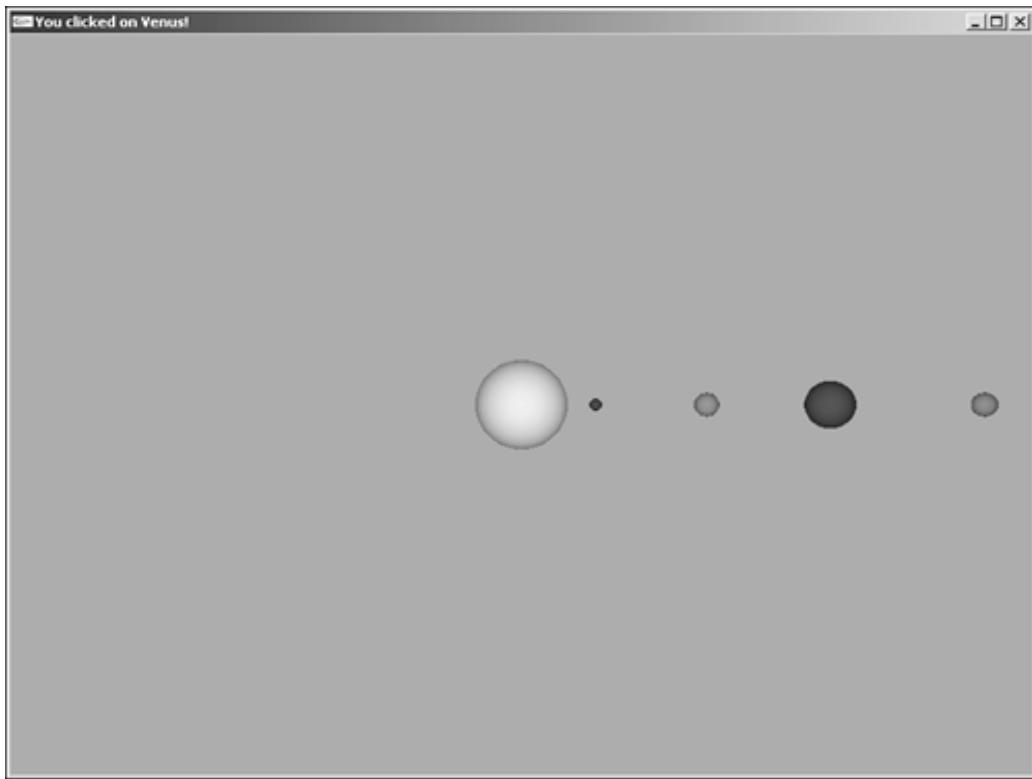
// If a single hit occurred, display the info.
if(hits == 1)
    ProcessPlanet(selectBuff[3]);
// Restore the projection matrix
glMatrixMode(GL_PROJECTION);
glPopMatrix();
// Go back to modelview for normal rendering
glMatrixMode(GL_MODELVIEW);

```

The `ProcessPlanet` function simply displays a message in the windows' caption telling which planet was clicked. This code is not shown because it is fairly trivial, consisting of no more than a switch statement and some `glutSetWindowTitle` function calls.

The output from PLANETS is shown in [Figure 12.2](#), where you can see the result of clicking the second planet from the sun.

Figure 12.2. Output from PLANETS after clicking a planet.



Although we don't go into any great detail here, it is worth discussing briefly the z values from the selection buffer. In the PLANETS example, each object or model was distinct and off alone in its own space. What if you apply this same method to several objects or models that perhaps overlap? You get multiple hit records! How do you know which one the user clicked? This situation can be tricky and requires some forethought. You can use the z values to determine which object was closest to the user in viewspace, which is the most likely object that was clicked. Still, for some shapes and geometry, if you aren't careful, it can be difficult to sort out precisely what the user intended to pick.

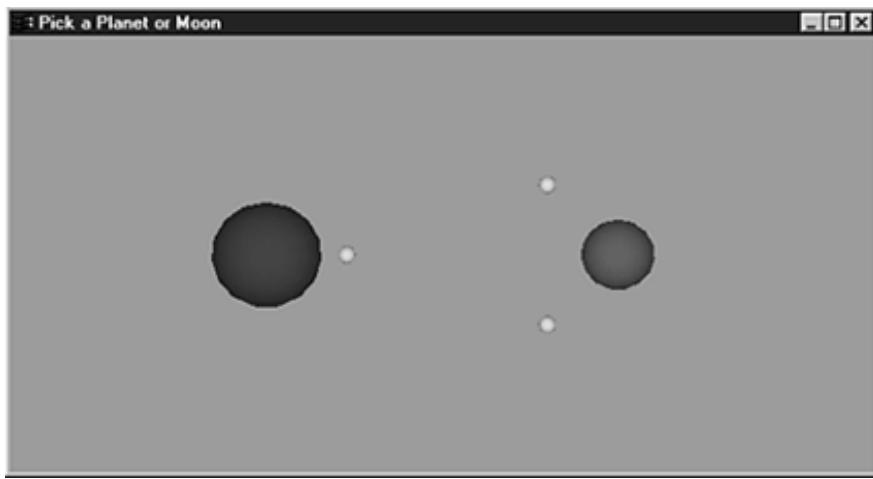
Hierarchical Picking

For the PLANETS example, we didn't push any names on the stack, but rather just replaced the existing one whenever a new object was to be rendered. This single name residing on the name stack was the only name returned in the selection buffer. We can also get multiple names when a selection hit occurs, by placing more than one name on the name stack. This capability is useful,

for instance, in drill-down situations when you need to know not only that a particular bolt was selected, but also that it belonged to a particular wheel, on a particular car, and so forth.

To demonstrate multiple names being returned on the name stack, we stick with the astronomy theme of our previous example. [Figure 12.3](#) shows two planets (okay, so use a little imagination)—a large blue planet with a single moon and a smaller red planet with two moons.

Figure 12.3. Two planets with their respective moons.



Rather than just identify the planet or moon that is clicked, we want to also identify the planet that is associated with the particular moon. The code in [Listing 12.4](#) shows our new rendering code for this scene. We push the names of the moons onto the name stack so that it contains the name of the planet as well as the name of the moon when selected.

Listing 12.4. Rendering Code for the MOONS Sample Program

```
///////////
// Define object names
#define EARTH    1
#define MARS    2
#define MOON1   3
#define MOON2   4
///////////
// Called to draw scene
void RenderScene(void)
{
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // Save the matrix state and do the rotations
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    // Translate the whole scene out and into view
    glTranslatef(0.0f, 0.0f, -300.0f);
    // Initialize the names stack
    glInitNames();
    glPushName(0);
    // Draw the Earth
    glPushMatrix();
    glColor3f(0.0f, 0.0f, 1.0f);
    glTranslatef(-100.0f, 0.0f, 0.0f);
    glLoadName(EARTH);
    DrawSphere(30.0f);
    // Draw the Moon
    glTranslatef(45.0f, 0.0f, 0.0f);
    glPushMatrix();
    glColor3f(1.0f, 0.0f, 0.0f);
    glTranslatef(45.0f, 0.0f, 0.0f);
    glLoadName(MARS);
    DrawSphere(15.0f);
    glPopMatrix();
    glPopName();
}
```

```

glColor3f(0.85f, 0.85f, 0.85f);
glPushName(MOON1);
DrawSphere(5.0f);
glPopName();
glPopMatrix();
// Draw Mars
glPushMatrix();
glColor3f(1.0f, 0.0f, 0.0f);
glTranslatef(100.0f, 0.0f, 0.0f);
glLoadName(MARS);
DrawSphere(20.0f);
// Draw Moon1
glTranslatef(-40.0f, 40.0f, 0.0f);
glColor3f(0.85f, 0.85f, 0.85f);
glPushName(MOON1);
DrawSphere(5.0f);
glPopName();
// Draw Moon2
glTranslatef(0.0f, -80.0f, 0.0f);
glPushName(MOON2);
DrawSphere(5.0f);
glPopName();
glPopMatrix();
// Restore the matrix state
glPopMatrix(); // Modelview matrix
glutSwapBuffers();
}

```

Now in our `ProcessSelection` function, we still call the `ProcessPlanet` function that we wrote, but this time, we pass the entire selection buffer:

```

// If a single hit occurred, display the info.
if(hits == 1)
    ProcessPlanet(selectBuff);

```

[Listing 12.5](#) shows the more substantial `ProcessPlanet` function for this example. In this instance, the bottom name on the name stack is always the name of the planet because it was pushed on first. If a moon is clicked, it is also on the name stack. This function displays the name of the planet selected, and if it was a moon, that information is also displayed. A sample output is shown in [Figure 12.4](#).

Listing 12.5. Code That Parses the Selection Buffer for the MOONS Sample Program

```

///////////////////////////////
// Parse the selection buffer to see which
// planet/moon was selected
void ProcessPlanet(GLuint *pSelectBuff)
{
    int id, count;
    char cMessage[64];
    strcpy(cMessage, "Error, no selection detected");
    // How many names on the name stack
    count = pSelectBuff[0];
    // Bottom of the name stack
    id = pSelectBuff[3];
    // Select on earth or mars, whichever was picked
    switch(id)
    {
        case EARTH:
            strcpy(cMessage, "You clicked Earth.");

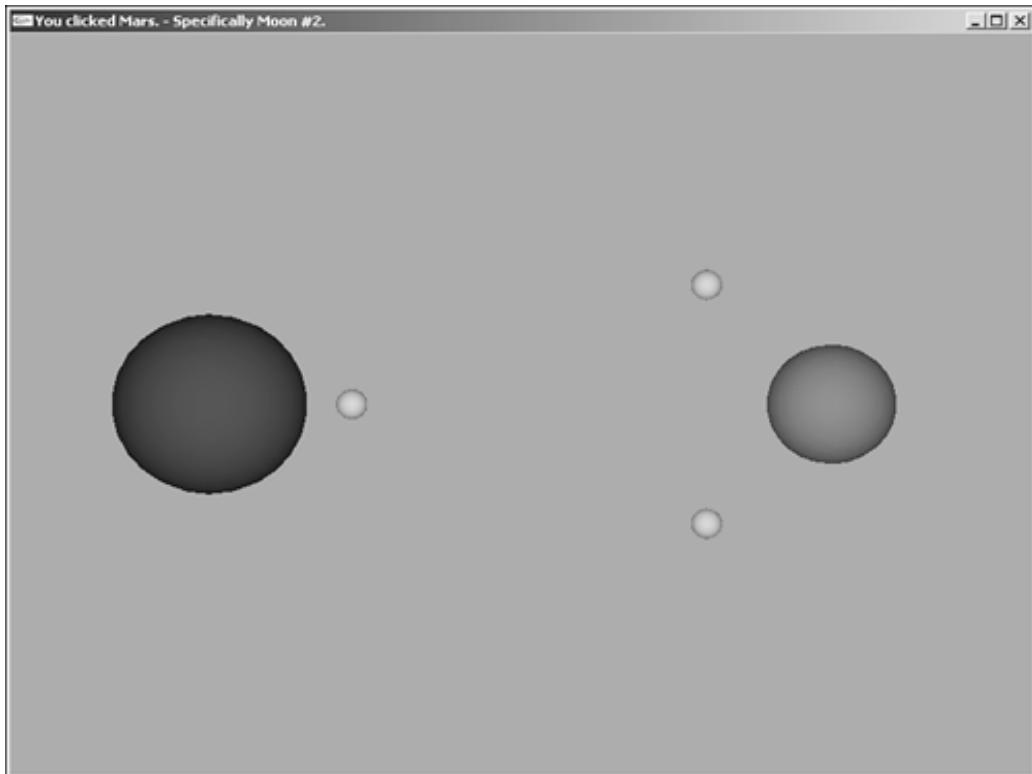
```

```

// If there is another name on the name stack,
// then it must be the moon that was selected
// This is what was actually clicked on
if(count == 2)
    strcat(cMessage, " - Specifically the moon.");
break;
case MARS:
strcpy(cMessage, "You clicked Mars.");
// We know the name stack is only two deep. The precise
// moon that was selected will be here.
if(count == 2)
{
    if(pSelectBuff[4] == MOON1)
        strcat(cMessage, " - Specifically Moon #1.");
    else
        strcat(cMessage, " - Specifically Moon #2.");
}
break;
}
// Display the message about planet and moon selection
glutSetWindowTitle(cMessage);
}

```

Figure 12.4. Sample output from the MOONS sample program.



Feedback

Feedback, like selection, is a rendering mode that does not produce output in the form of pixels on the screen. Instead, information is written to a feedback buffer indicating how the scene would have been rendered. This information includes transformed vertex data in window coordinates, color data resulting from lighting calculations, and texture data—essentially everything needed to rasterize the primitives.

You enter feedback mode the same way you enter selection mode, by calling `glRenderMode` with a

`GL_FEEDBACK` argument. You must reset the rendering mode to `GL_RENDER` to fill the feedback buffer and return to normal rendering mode.

The Feedback Buffer

The feedback buffer is an array of floating-point values specified with the `glFeedback` function:

```
void glFeedbackBuffer(GLsizei size, GLenum type, GLfloat *buffer);
```

This function takes the size of the feedback buffer, the type and amount of drawing information wanted, and a pointer to the buffer itself.

Valid values for `type` appear in [Table 12.1](#). The type of data specifies how much data is placed in the feedback buffer for each vertex. Color data is represented by a single value in color index mode or four values for RGBA color mode.

Table 12.1. Feedback Buffer Types

Color Data	Vertex Texture Data	Total Values	Type	Coordinates
<code>GL_2D</code>	x, y	N/A	N/A	2
<code>GL_3D</code>	x, y, z	N/A	N/A	3
<code>GL_3D_COLOR</code>	x, y, z	C	N/A	3 + C
<code>GL_3D_COLOR_TEXTURE</code>	x, y, z	C	4	7 + C
<code>GL_4D_COLOR_TEXTURE</code>	x, y, z, w	C	4	8 + C

Feedback Data

The feedback buffer contains a list of tokens followed by vertex data and possibly color and texture data. You can parse for these tokens (see [Table 12.2](#)) to determine the types of primitives that would have been rendered. One limitation of feedback occurs when using multiple texture units. In this case, only texture coordinates from the first texture unit are returned.

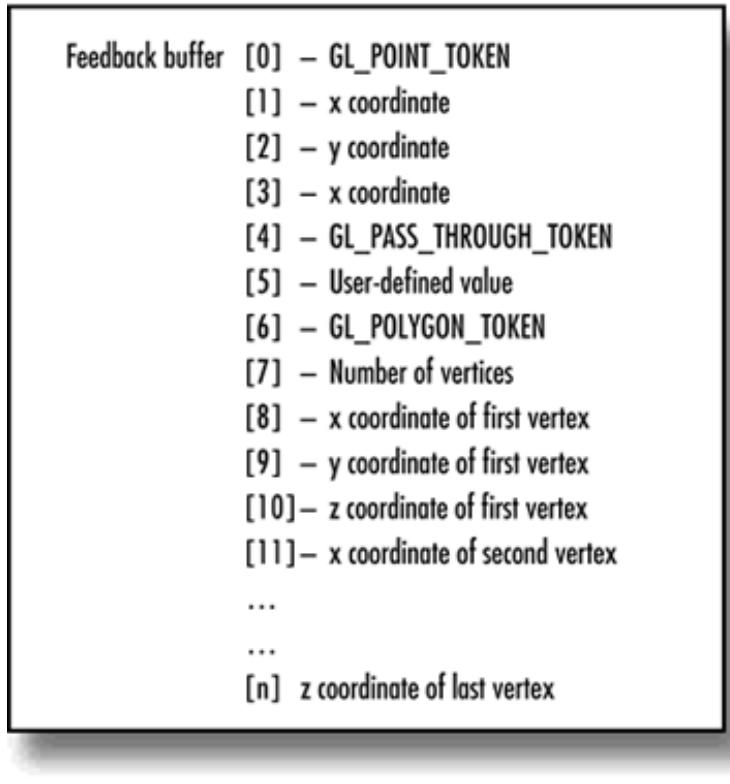
Table 12.2. Feedback Buffer Tokens

Token	Primitive
<code>GL_POINT_TOKEN</code>	Points
<code>GL_LINE_TOKEN</code>	Line
<code>GL_LINE_RESET_TOKEN</code>	Line segment when line stipple is reset
<code>GL_POLYGON_TOKEN</code>	Polygon
<code>GL_BITMAP_TOKEN</code>	Bitmap
<code>GL_DRAW_PIXEL_TOKEN</code>	Pixel rectangle drawn
<code>GL_COPY_PIXEL_TOKEN</code>	Pixel rectangle copied

GL_PASS_THROUGH_TOKEN User-defined marker

The point, bitmap, and pixel tokens are followed by data for a single vertex and possibly color and texture data. This depends on the data type from [Table 12.1](#) specified in the call to `glFeedbackBuffer`. The line tokens return two sets of vertex data, and the polygon token is immediately followed by the number of vertices that follow. The user-defined marker (**GL_PASS_THROUGH_TOKEN**) is followed by a single floating-point value that is user defined. [Figure 12.5](#) shows an example of a feedback buffer's memory layout if a **GL_3D** type were specified. Here, we see the data for a point, token, and polygon rendered in that order.

Figure 12.5. A sample memory layout for a feedback buffer.



Passthrough Markers

When your rendering code is executing, the feedback buffer is filled with tokens and vertex data as each primitive is specified. Just as you can in selection mode, you can flag certain primitives by naming them. In feedback mode, you can set markers between your primitives, as well. You do so by calling `glPassThrough`:

```
void glPassThrough(GLfloat token);
```

This function places a **GL_PASS_THROUGH_TOKEN** in the feedback buffer, followed by the value you specify when calling the function. This process is somewhat similar to naming primitives in selection mode. It's the only way of labeling objects in the feedback buffer.

A Feedback Example

An excellent use of feedback is to obtain window coordinate information regarding any objects that you render. You can then use this information to place controls or labels near the objects in the window or other windows around them.

To demonstrate feedback, we use selection to determine which of two objects on the screen has been clicked by the user. Then we enter feedback mode and render the scene again to obtain the vertex information in window coordinates. Using this data, we determine the minimum and maximum x and y values for the object and use those values to draw a focus rectangle around the object. The result is graphical selection and deselection of one or both objects.

Label the Objects for Feedback

Listing 12.6 shows the rendering code for our sample program, SELECT. Don't confuse this example with a demonstration of selection mode! Even though selection mode is employed in our example to select an object on the screen, we are demonstrating the process of getting enough information about that object—using feedback—to draw a rectangle around it using normal Windows coordinates GDI commands. Notice the use of `glPassThrough` to label the objects in the feedback buffer, right after the calls to `glLoadName` to label the objects in the selection buffer. Because these functions are ignored when the render mode is `GL_RENDER`, they have an effect only when rendering for selection or feedback.

Listing 12.6. Rendering Code for the SELECT Sample Program

```
///////////
// Object Names
#define TORUS    1
#define SPHERE   2
///////////
// Render the torus and sphere
void DrawObjects(void)
{
    // Save the matrix state and do the rotations
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    // Translate the whole scene out and into view
    glTranslatef(-0.75f, 0.0f, -2.5f);
    // Initialize the names stack
    glInitNames();
    glPushName(0);
    // Set material color, Yellow
    // torus
    glColor3f(1.0f, 1.0f, 0.0f);
    glLoadName(TORUS);
    glPassThrough((GLfloat)TORUS);
    DrawTorus(40, 20);

    // Draw Sphere
    glColor3f(0.5f, 0.0f, 0.0f);
    glTranslatef(1.5f, 0.0f, 0.0f);
    glLoadName(SPHERE);
    glPassThrough((GLfloat)SPHERE);
    DrawSphere(0.5f);
    // Restore the matrix state
    glPopMatrix();    // Modelview matrix
}
///////////
// Called to draw scene
void RenderScene(void)
```

```

{
// Clear the window with current clearing color
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// Draw the objects in the scene
DrawObjects();
// If something is selected, draw a bounding box around it
if(selectedObject != 0)
{
    int viewport[4];
    // Get the viewport
    glGetIntegerv(GL_VIEWPORT, viewport);
    // Remap the viewing volume to match window coordinates (approximately)
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    // Establish clipping volume (left, right, bottom, top, near, far)
    glOrtho(viewport[0], viewport[2], viewport[3], viewport[1], -1, 1);
    glMatrixMode(GL_MODELVIEW);

    glDisable(GL_LIGHTING);
    glColor3f(1.0f, 0.0f, 0.0f);
    glBegin(GL_LINE_LOOP);
        glVertex2i(boundingRect.left, boundingRect.top);
        glVertex2i(boundingRect.left, boundingRect.bottom);
        glVertex2i(boundingRect.right, boundingRect.bottom);
        glVertex2i(boundingRect.right, boundingRect.top);
    glEnd();
    glEnable(GL_LIGHTING);
}
glMatrixMode(GL_PROJECTION);
glPopMatrix();
glMatrixMode(GL_MODELVIEW);
glutSwapBuffers();
}

```

For this example, the rendering code is broken into two functions: `RenderScene` and `DrawObjects`. `RenderScene` is our normal top-level rendering function, but we have moved the actual drawing of the objects that we may select to outside this function. The `RenderScene` function draws the objects, but it also draws the bounding rectangle around an object if it is selected. `selectedObject` is a variable we will use in a moment to indicate which object is currently selected.

Step 1: Select the Object

[Figure 12.6](#) shows the output from this rendering code, displaying a torus and sphere. When the user clicks one of the objects, the function `ProcessSelection` is called (see [Listing 12.7](#)). This is similar to the selection code in the previous two examples (in [Listings 12.3](#) and [12.5](#)).

Listing 12.7. Selection Processing for the SELECT Sample Program

```

///////////////////////////////
// Process the selection, which is triggered by a right mouse
// click at (xPos, yPos).
#define BUFFER_LENGTH 64
void ProcessSelection(int xPos, int yPos)
{
    // Space for selection buffer
    static GLuint selectBuff[BUFFER_LENGTH];
    // Hit counter and viewport storage

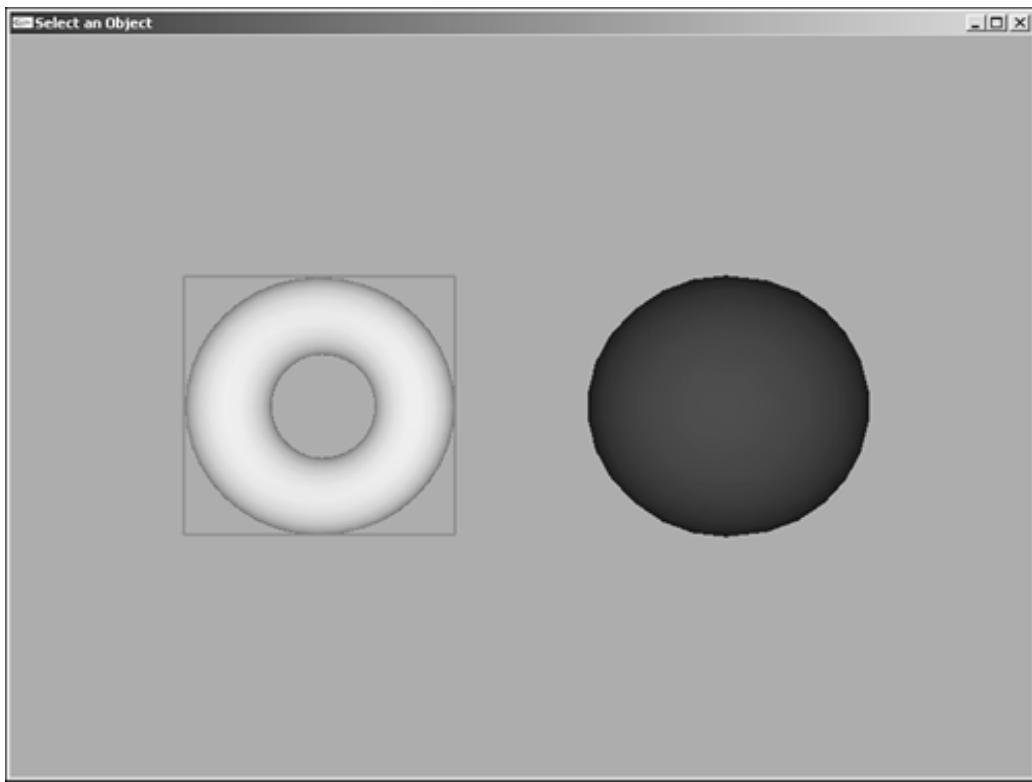
```

```

GLint hits, viewport[4];
// Setup selection buffer
glSelectBuffer(BUFFER_LENGTH, selectBuff);
// Get the viewport
glGetIntegerv(GL_VIEWPORT, viewport);
// Switch to projection and save the matrix
glMatrixMode(GL_PROJECTION);
glPushMatrix();
// Change render mode
glRenderMode(GL_SELECT);
// Establish new clipping volume to be unit cube around
// mouse cursor point (xPos, yPos) and extending two pixels
// in the vertical and horizontal direction
glLoadIdentity();
gluPickMatrix(xPos, viewport[3] - yPos, 2, 2, viewport);
// Apply perspective matrix
gluPerspective(60.0f, fAspect, 1.0, 425.0);
// Draw the scene
DrawObjects();
// Collect the hits
hits = glRenderMode(GL_RENDER);
// Restore the projection matrix
glMatrixMode(GL_PROJECTION);
glPopMatrix();
// Go back to modelview for normal rendering
glMatrixMode(GL_MODELVIEW);
// If a single hit occurred, display the info.
if(hits == 1)
{
    MakeSelection(selectBuff[3]);
    if(selectedObject == selectBuff[3])
        selectedObject = 0;
    else
        selectedObject = selectBuff[3];
}
glutPostRedisplay();
}

```

Figure 12.6. Output from the SELECT program after the sphere has been clicked.



Step 2: Get Feedback on the Object

Now that we have determined which object was clicked (we saved this in the `selectedObject` variable), we set up the feedback buffer and render again in feedback mode. [Listing 12.8](#) shows the code that sets up feedback mode for this example and calls `DrawObjects` to redraw just the torus and sphere scene. This time, however, the `glPassThrough` functions put markers for the objects in the feedback buffer.

Listing 12.8. Load and Parse the Feedback Buffer

```
//////////  
// Go into feedback mode and draw a rectangle around the object  
#define FEED_BUFF_SIZE 32768  
void MakeSelection(int nChoice)  
{  
    // Space for the feedback buffer  
    static GLfloat feedBackBuff[FEED_BUFF_SIZE];  
    // Storage for counters, etc.  
    int size,i,j,count;  
    // Initial minimum and maximum values  
    boundingRect.right = boundingRect.bottom = -999999.0f;  
    boundingRect.left = boundingRect.top = 999999.0f;  
    // Set the feedback buffer  
    glFeedbackBuffer(FEED_BUFF_SIZE,GL_2D, feedBackBuff);  
    // Enter feedback mode  
    glRenderMode(GL_FEEDBACK);  
    // Redraw the scene  
    DrawObjects();  
    // Leave feedback mode  
    size = glRenderMode(GL_RENDER);  
    // Parse the feedback buffer and get the  
    // min and max X and Y window coordinates  
    i = 0;  
    while(i < size)
```

```

    {
    // Search for appropriate token
    if(feedBackBuff[i] == GL_PASS_THROUGH_TOKEN)
        if(feedBackBuff[i+1] == (GLfloat)nChoice)
        {
        i+= 2;
        // Loop until next token is reached
        while(i < size && feedBackBuff[i] != GL_PASS_THROUGH_TOKEN)
            {
            // Just get the polygons
            if(feedBackBuff[i] == GL_POLYGON_TOKEN)
                {
                // Get all the values for this polygon
                count = (int)feedBackBuff[++i]; // How many vertices
                i++;
                for(j = 0; j < count; j++) // Loop for each vertex
                    {
                    // Min and Max X
                    if(feedBackBuff[i] > boundingRect.right)
                        boundingRect.right = feedBackBuff[i];
                    if(feedBackBuff[i] < boundingRect.left)
                        boundingRect.left = feedBackBuff[i];
                    i++;
                    // Min and Max Y
                    if(feedBackBuff[i] > boundingRect.bottom)
                        boundingRect.bottom = feedBackBuff[i];
                    if(feedBackBuff[i] < boundingRect.top)
                        boundingRect.top = feedBackBuff[i];
                    i++;
                    }
                }
            else
                i++; // Get next index and keep looking
        }
    break;
}
i++;
}
}

```

When the feedback buffer is filled, we search it for `GL_PASS_THROUGH_TOKEN`. When we find one, we get the next value and determine whether it is the one we are looking for. If so, the only task that remains is to loop through all the polygons for this object and get the minimum and maximum window x and y values. These values are stored in the `boundingRect` structure and then used by the `RenderScene` function to draw a focus rectangle around the selected object.

Summary

Selection and feedback are two powerful features of OpenGL that enable you to facilitate the user's active interaction with a scene. Selection and picking are used to identify an object or region of a scene in OpenGL coordinates rather than just window coordinates. Feedback returns valuable information about how an object or primitive is actually drawn in the window. You can use this information to implement features such as annotations or bounding boxes in your scene.

Reference

glFeedbackBuffer

Purpose: Sets the buffer to be used for feedback data.

Include File: `<gl.h>`

Syntax:

```
void glFeedbackBuffer(GLsizei size, GLenum type,
→ GLfloat *buffer);
```

Description: This function establishes the feedback buffer and type of vertex information desired. Feedback is a rendering mode; rather than render to the frame buffer, OpenGL sends vertex data to the buffer specified by **buffer*. These blocks of data can include x, y, z, and w coordinate positions (in window coordinates); color data for color index mode or RGBA color mode; and texture coordinates. The amount and type of information desired are specified by the *type* argument.

Parameters:

size `GLsizei`: The maximum number of entries allocated for **buffer*. If a block of data written to the feedback would overflow the amount of space allocated, only the part of the block that will fit in the buffer is written.

type `GLenum`: Specifies the kind of vertex data to be returned in the feedback buffer. Each vertex generates a block of data in the feedback buffer. For each of the following types, the block of data contains a primitive token identifier followed by the vertex data. The vertex data specifically includes the following:

`GL_2D`: x and y coordinate pairs.

`GL_3D`: x, y, and z coordinate triplets.

`GL_3D_COLOR`: x, y, z coordinates and color data (one value for color index, four for RGBA).

`GL_3D_COLOR_TEXTURE`: x, y, z coordinates; color data (one or four values); and four texture coordinates. If multiple texture units are employed, only coordinates from the first texture unit are returned.

`GL_4D_COLOR_TEXTURE`: x, y, z, and w coordinates; color data (one or four values); and four texture coordinates.

buffer `GLfloat*`: Buffer where feedback data will be stored.

Returns: None.

See Also: `glPassThrough`, `glRenderMode`, `glSelectBuffer`

glInitNames

Purpose: Initializes the name stack.

Include File: `<gl.h>`

Syntax:

```
void glInitNames(void );
```

Description: The name stack is used to allow drawing primitives or groups of primitives to be named with an unsigned integer when rendered in selection mode. Each time a primitive is named, its name is pushed on the name stack with `glPushName`, or the current name is replaced with `glLoadName`. This function sets the name stack to its initial condition with no names on the stack.

Returns: None.

See Also: `glInitNames`, `glPushName`, `glRenderMode`, `glSelectBuffer`

glLoadName

Purpose: Loads a name onto the name stack.

Include File: `<gl.h>`

Syntax:

```
void glLoadName(GLuint name);
```

Description: This function places the name specified in the top entry of the name stack. The name stack is used to name primitives or groups of primitives when rendered in selection mode. The current name on the name stack is actually replaced by the name specified with this function.

Parameters:

`name` `GLuint`: Specifies the name to be placed on the name stack. Selection names are unsigned integers.

Returns: None.

See Also: `glInitNames`, `glPushName`, `glRenderMode`, `glSelectBuffer`

glPassThrough

Purpose: Places a marker in the feedback buffer.

Include File: `<gl.h>`

Syntax:

```
void glPassThrough(GLfloat token);
```

Description: When OpenGL is placed in feedback mode, no pixels are drawn to the frame buffer. Instead, information about the drawing primitives is placed in a feedback buffer. This function allows you to place the token `GL_PASS_THROUGH_TOKEN` in the midst of the feedback buffer data, which is followed by the floating-point value specified by `token`. This function is called in your rendering code and has no effect unless in feedback mode.

Parameters:

`token` `GLfloat`: A value to be placed in the feedback buffer following the `GL_PASS_THROUGH_TOKEN`.

Returns: None.

See Also: `glFeedbackBuffer`, `glRenderMode`

glPopName

Purpose: Pops (removes) the top entry from the name stack.

Include File: `<gl.h>`

Syntax:

```
void glPopName(void);
```

Description: The name stack is used during selection to identify drawing commands. This function removes a name from the top of the name stack. The current depth of the name stack can be retrieved by calling `glGet` with `GL_NAME_STACK_DEPTH`. Popping off an empty name stack generates an OpenGL error (see `glGetError`).

Returns: None.

See Also: `glInitNames`, `glLoadName`, `glRenderMode`, `glSelectBuffer`, `glPushName`

glPushName

Purpose: Specifies a name that will be pushed on the name stack.

Include File: `<gl.h>`

Syntax:

```
void glPushName(GLuint name);
```

Description: The name stack is used during selection to identify drawing commands. This function pushes a name on the name stack to identify any subsequent drawing commands. The name stack's maximum depth can be retrieved by calling `glGet` with `GL_MAX_NAME_STACK_DEPTH` and the current depth by calling `glGet` with `GL_NAME_STACK_DEPTH`. The maximum depth of the name stack can vary with implementation, but all implementations must support at least 64 entries. Pushing past the end of the name stack generates an OpenGL error and will most likely be ignored by the implementation.

Parameters:

name `GLuint`: The name to be pushed onto the name stack.

Returns: None.

See Also: `glInitNames`, `glLoadName`, `glRenderMode`, `glSelectBuffer`, `glPopName`

glRenderMode

Purpose: Sets one of three rasterization modes.

Include File: `<gl.h>`

Syntax:

```
GLint glRenderMode(GLenum mode);
```

Description: OpenGL operates in three modes when you call your drawing functions:

`GL_RENDER`: Render mode (the default). Drawing functions result in pixels in the frame buffer.

`GL_SELECT`: Selection mode. No changes to the frame buffer are made. Rather, hit records written to the selection buffer record primitives that would have been drawn within the viewing volume. The selection buffer must be allocated and specified first with a call to `glSelectBuffer`.

`GL_FEEDBACK`: Feedback mode. No changes to the frame buffer are made. Instead, coordinates and attributes of vertices that would have been rendered in render mode are written to a feedback buffer. The feedback buffer must be allocated and specified first with a call to `glFeedbackBuffer`.

Parameters:

mode `GLenum`: Specifies the rasterization mode. May be any one of `GL_RENDER`, `GL_SELECT`, or `GL_FEEDBACK`. The default value is `GL_RENDER`.

Returns: The return value depends on the rasterization mode that was set the last time this function was called:

`GL_RENDER`: Zero.

`GL_SELECT`: The number of hit records written to the selection buffer.

`GL_FEEDBACK`: The number of values written to the feedback buffer. Note that this is not the same as the number of vertices written.

See Also: `glFeedbackBuffer`, `glInitNames`, `glLoadName`, `glPassThrough`, `glPushName`, `glSelectBuffer`

glSelectBuffer

Purpose: Sets the buffer to be used for selection values.**Include File:** `<gl.h>`**Syntax:**`void glSelectBuffer(GLsizei size, GLuint *buffer);`**Description:** When OpenGL is in selection mode (`GL_SELECT`), drawing commands do not produce pixels in the frame buffer. Instead, they produce hit records that are written to the selection buffer that is established by this function. Each hit record consists of the following data:

- The number of names on the name stack when the hit occurred.
- The minimum and maximum z values of all the vertices of the primitives that intersected the viewing volume. This value is scaled to range from 0.0 to 1.0.
- The contents of the name stack at the time of the hit, starting with the bottom element.

Parameters:***size*** `GLsizei`: The number of values that can be written into the buffer established by `*buffer`.***buffer*** `GLuint*`: A pointer to memory that will contain the selection hit records.**Returns:** None.**See Also:** `glFeedbackBuffer`, `glInitNames`, `glLoadName`, `glPushName`, `glRenderMode`

gluPickMatrix

Purpose: Defines a picking region that can be used to identify user selections.**Include File:** `<glu.h>`**Syntax:**`void gluPickMatrix(GLdouble x, GLdouble y,
→ GLdouble width, GLdouble height, GLint viewport[4]);`**Description:** This function creates a matrix that will define a smaller viewing volume based on screen coordinates for the purpose of selection. By using the mouse coordinates with this function in selection mode, you can determine which of your objects are under or near the mouse cursor. The matrix created is multiplied by the current projection matrix. Typically, you should call `glLoadIdentity` before calling this function and then multiply the perspective matrix that you used to create the viewing volume in the first place. If you are using `gluPickMatrix` to pick NURBS surfaces, you must turn off the NURBS property `GLU_AUTO_LOAD_MATRIX` before using this function.**Parameters:*****x, y*** `GLdouble`: The center of the picking region in window coordinates.***width, height*** `GLdouble`: The width and height of the desired picking region in window coordinates.

<code>viewport</code>	<code>GLint[4]</code> : The current viewport. You can get the current viewport by calling <code>glGetIntegerv</code> with <code>GL_VIEWPORT</code> .
Returns:	None.
See Also:	<code>glGet</code> , <code>glLoadIdentity</code> , <code>glMultMatrix</code> , <code>glRenderMode</code> , <code>gluPerspective</code>

Part II: OpenGL Everywhere

One of the most often touted features and advantages to using OpenGL for rendering is that it is widely available across operating systems and hardware platforms. Although the core OpenGL functionality remains consistent, you will notice a few quirks when moving from one platform to another. So far, we have been using GLUT as a means of achieving source portability for our sample programs. GLUT is fine for learning purposes, but professional software developers know that to build high-quality and feature-rich applications, it is often necessary to incorporate specific operating system features or support the GUI features that users expect of programs that run in their favorite environment.

The next three chapters concern themselves with using OpenGL on three different but popular platforms. Each operating system has its own advantages and quirks when it comes to using OpenGL. Each chapter begins by describing the common task of getting OpenGL up and running but then delves into some special features of OpenGL that are specific to the platform, as well as some notes on maximizing performance on that platform.

OpenGL. It's everywhere. Do the math.

Chapter 13. Wiggle: OpenGL on Windows

by Richard S. Wright, Jr.

WHAT YOU'LL LEARN IN THIS CHAPTER:

How To	Functions You'll Use
Request and select an OpenGL pixel format	<code>ChoosePixelFormat/DescribePixelFormat/SetPixelFormat</code>
Create and use OpenGL rendering contexts	<code>wglCreateContext/wglDeleteContext/wglGetCurrentContext</code>
Respond to window messages	<code>WM_PAINT/WM_CREATE/WM_DESTROY/WM_SIZE</code>
Use double buffering in Windows	<code>SwapBuffers</code>

OpenGL is purely a graphics API, with user interaction and the screen or window handled by the host environment. To facilitate this partnership, each environment usually has some extensions that "glue" OpenGL to its own window management and user interface functions. This glue is code that associates OpenGL drawing commands to a particular window. It is also necessary to provide functions for setting buffer modes, color depths, and other drawing characteristics.

For Microsoft Windows, this glue code is embodied in a set of new functions added to the Windows API. They are called the *wiggle functions* because they are prefixed with `wgl` rather than `gl`. These gluing functions are explained in this chapter, where we dispense with using the GLUT library for

our OpenGL framework and build full-fledged Windows applications that can take advantage of all the operating system's features. You will see what characteristics a Windows window must have to support OpenGL graphics. You will learn which messages a well-behaved OpenGL window should handle and how. The concepts of this chapter are introduced gradually, as we build a model OpenGL program that provides a framework for Windows-specific OpenGL support.

So far in this book, you've needed no prior knowledge of 3D graphics and only a rudimentary knowledge of C programming. For this chapter, however, we assume you have at least an entry-level knowledge of Windows programming. Otherwise, we would have wound up writing a book twice the size of this one, and we would have spent more time on the details of Windows programming and less on OpenGL programming. If you are new to Windows, or if you cut your teeth on one of the application frameworks and aren't all that familiar with Windows procedures, message routing, and so forth, you might want to check out some of the recommended reading in [Appendix A, "Further Reading,"](#) before going too much further in this chapter.

OpenGL Implementations on Windows

OpenGL became available for the Win32 platform with the release of Windows NT version 3.5. It was later released as an add-on for Windows 95 and then became a shipping part of the Windows 95 operating system (OSR2). OpenGL is now a native API on any Win32 platform (Windows 95/98/ME, Windows NT/2000/XP/2003), with its functions exported from `opengl32.dll`. You need to be aware of four flavors of OpenGL on Windows: Generic, ICD, MCD, and the Extended. Each has its pros and cons from both the user and developer point of view. You should at least have a high-level understanding of how these implementations work and what their drawbacks might be.

Generic OpenGL

A generic implementation of OpenGL is simply a software implementation that does not use specific 3D hardware. The Microsoft implementation bundled with Windows is a generic implementation. The Silicon Graphics Incorporated (SGI) OpenGL for Windows implementation (no longer widely available) optionally made use of MMX instructions, but it was not considered dedicated 3D hardware, so it was still considered a generic software implementation. Another implementation called *MESA* (www.mesa3d.org) is not strictly a "real" OpenGL implementation—it's a "work-a-like"—but for most purposes, you can consider it to be so. MESA can also be hooked to hardware, but this should be considered a special case of the mini-driver (discussed shortly).

Although the MESA implementation has kept up with OpenGL's advancing feature set over the years, the Microsoft generic implementation has not been updated since OpenGL version 1.1. Not to worry, we will soon show you how to get to all the OpenGL functionality your graphics card supports.

Installable Client Driver

The Installable Client Driver (ICD) was the original hardware driver interface provided for Windows NT. The ICD must implement the entire OpenGL pipeline using a combination of software and the specific hardware for which it was written. Creating an ICD from scratch is a considerable amount of work for a vendor to undertake.

The ICD drops in and works with Microsoft's OpenGL implementation. Applications linked to `opengl32.dll` are automatically dispatched to the ICD driver code for OpenGL calls. This mechanism is ideal because applications do not have to be recompiled to take advantage of OpenGL hardware should it become available. The ICD is actually a part of the display driver and does not affect the existing `openGL32.dll` system DLL. This driver model provides the vendor with the most opportunities to optimize its driver and hardware combination.

Mini-Client Driver

The Mini-Client Driver (MCD) was a compromise between a software and hardware

implementation. Most early PC 3D hardware provided hardware-accelerated rasterization only. (See "[The "Pipeline"](#)" section in [Chapter 2](#), "Using OpenGL.") The MCD driver model allowed applications to use Microsoft's generic implementation for features that were not available in hardware. For example, transform and lighting could come from Microsoft's OpenGL software, but the actual rendering of lit shaded triangles would be handled by the hardware.

The MCD driver implementation made it easy for hardware vendors to create OpenGL drivers for their hardware. Most of the work was done by Microsoft, and whatever features the vendors did not implement in hardware was handed back to the Microsoft generic implementation.

The MCD driver model showed great promise for bringing OpenGL to the PC mass market. Initially available for Windows NT, a software development kit (SDK) was provided to hardware vendors to create MCD drivers for Windows 95. After many hardware vendors had completed their MCD drivers, Microsoft decided not to license the code for public release. This gave their own proprietary 3D API a temporary advantage in the consumer marketplace.

The MCD driver model today is largely obsolete, but a few implementations are still in use. One reason for its demise is that the MCD driver model cannot support Intel's Accelerated Graphics Port (AGP) texturing efficiently. Another is that SGI began providing an optimized ICD driver kit to vendors that made writing ICDs almost as easy as writing MCDs. (This move was a response to Microsoft's temporary withdrawal of support for OpenGL on Windows 95.) This driver kit and model has now replaced the MCD model and SDK.

Mini-Driver

A mini-driver is not a real display driver. Instead, it is a drop-in replacement for `opengl32.dll` that makes calls to a hardware vendor's proprietary 3D hardware driver. Typically, these mini-drivers convert OpenGL calls to roughly equivalent calls in a vendor's proprietary 3D API. The first mini-driver was written by 3Dfx for its Voodoo graphics card. This DLL drop-in converted OpenGL calls into the Voodoo's native Glide (the 3Dfx 3D API) programming interface.

Id software first wrote a version of its popular game Quake that used OpenGL and ran with this 3Dfx mini-driver. For this reason, as mini-drivers were developed for other graphics cards, they were sometimes called "Quake drivers." Many of the higher-end OpenGL hardware vendors would sarcastically refer to the consumer boards as "Quake accelerators," not worthy of comparison to their hardware. Many other game hardware vendors hopped on the bandwagon and began supplying mini-drivers for their hardware, too. Although they popularized OpenGL for games, mini-drivers often had missing OpenGL functions or features. Any application that used OpenGL did not necessarily work with a mini-driver. Typically, these drivers provided only the barest functionality needed to run a popular game.

Fortunately, the widespread popularity of OpenGL has made the mini-driver obsolete on newer commodity PCs. You may still come across this beast, however, in older PCs or installations using older graphics hardware. One persisting incarnation of the mini-driver is the wrapper DLL that takes OpenGL function calls and converts them to Direct 3D functionality.

Extended OpenGL

If you are developing software for any version of Microsoft Windows, you are most likely making use of header files and an import library that works with Microsoft's `opengl32.dll`. This DLL is designed to provide a generic (software-rendered) fallback if 3D hardware is not installed, and as a dispatch mechanism that works with the official OpenGL driver model for hardware-based OpenGL implementations. Using this header and import library alone gives you access only to functions and capabilities present in OpenGL 1.1.

In the introduction, we outlined for you the features that have been added to OpenGL since 1.1 up until OpenGL 2.0 with a fully integrated shading language. Take note, however, that OpenGL 1.1 is still a very capable and full-featured graphics API and is suitable for a wide range of graphical applications, including games or business graphics. Even without the additional features of

OpenGL 1.2 and beyond, graphics hardware performance has increased exponentially, and most PC graphics cards have the entire OpenGL pipeline implemented in special-purpose hardware. OpenGL 1.1 can still produce screaming fast and highly complex 3D renderings!

Many applications still will require or at least be significantly enhanced by making use of the newer OpenGL innovations. To get to the newer OpenGL features (which are widely supported), you need to use the same OpenGL extension mechanism that you use to get to vendor-specific OpenGL enhancements. OpenGL extensions were introduced in [Chapter 2](#), and the specifics of using this extension mechanism on Windows are covered later in this chapter in the section "[OpenGL and WGL Extensions](#)."

This may sound like a bewildering environment in which to develop 3D graphics—especially if you plan to port your applications to say, the Macintosh platform, where OpenGL features are updated more consistently with each OS release. Some strategies, however, can make such development more manageable. First, you can call the following function so your application can tell at runtime which OpenGL version the hardware driver supports:

```
glGetString(GL_VERSION);
```

This way, you can gracefully decide whether the application is going to be able to run at all on the user's system. Because OpenGL and its extensions are dynamically loaded, there is no reason your programs should not at least start and present the user with a friendly and informative error or diagnostic message.

You also need to think carefully about what OpenGL features your application *must* have. Can the application be written to use only OpenGL 1.1 features? Will the application be usable at all if no hardware is present and the user must use the built-in software renderer? If the answer to either of these questions is yes, you should first write your application's rendering code using only the import library for OpenGL 1.1. This gives you the widest possible audience for your application.

When you have the basic rendering code in place, you can go back and consider performance optimizations or special visual effects available with newer OpenGL features that you want to make available in your program. By checking the OpenGL version early in your program, you can introduce different rendering paths or functions that will optionally perform better or provide additional visual effects to your rendering. For example, static texture maps could be replaced with fragment programs, or standard fog replaced with volumetric fog made possible through vertex programs. Using the latest and greatest features allows you to really show off your program, but if you rely on them exclusively, you may be severely limiting your audience...and sales.

Sometimes, however, your application really *must* have some newer OpenGL feature. For example, a medical visualization package may require that 3D texturing or the imaging subset be available. In these types of more specialized or vertical markets, your application will simply have to require some minimal OpenGL support to run. The OpenGL version required in these cases will be listed among any other minimum system requirements that you specify are needed for your software. Again, your application can check for these details at startup.

Basic Windows Rendering

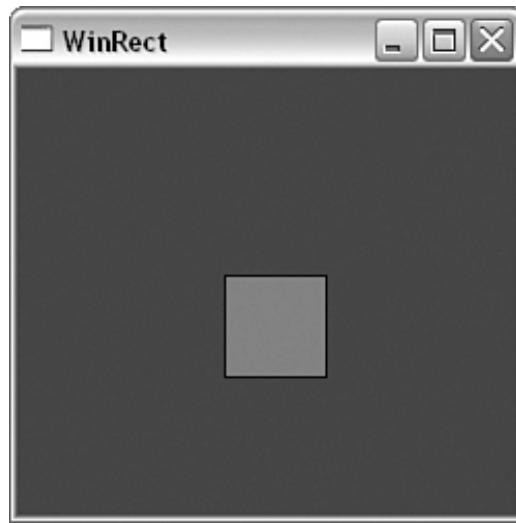
The GLUT library provided only one window, and OpenGL function calls always produced output in that window. (Where else would they go?) Your own real-world Windows applications, however, will often have more than one window. In fact, dialog boxes, controls, and even menus are actually windows at a fundamental level; having a useful program that contains only one window is nearly impossible. How does OpenGL know where to draw when you execute your rendering code? Before we answer this question, let's first review how we normally draw in a window without using OpenGL.

GDI Device Contexts

Normally, when you draw in a window without using OpenGL, you use the Windows Graphics Device Interface (GDI) functions. Each window has a device context that actually receives the graphics output, and each GDI function takes a device context as an argument to indicate which window you want the function to affect. You can have multiple device contexts, but only one for each window.

The sample program WINRECT on the companion CD draws an ordinary window with a blue background and a red square in the center. The output from this program, shown in [Figure 13.1](#), should look familiar to you. This is nearly the same image produced by the second OpenGL program in [Chapter 2](#), GLRECT. Unlike that earlier example, however, the WINRECT program does not use GLUT; we wrote it entirely with the Windows API. This code is relatively generic as far as Windows programming goes. A `WinMain` function gets things started and keeps the message pump going, and a `WndProc` function handles messages for the main window.

Figure 13.1. Output from WINRECT.



Your familiarity with Windows programming should extend to the details of creating and displaying a window, so we cover only the code from this example that is responsible for drawing the background and square, and won't list the entire program here.

First, you must create a blue and a red brush for filling and painting. The handles for these brushes are declared globally:

```
// Handles to GDI brushes we will use for drawing
HBRUSH hBlueBrush, hRedBrush;
```

Then you create the brushes in the `WinMain` function, using the `RGB` macro to create solid red and blue brushes:

```
// Create a blue and red brush for drawing and filling
// operations.          //     Red,  green,  blue
hBlueBrush = CreateSolidBrush(RGB(    0,      0,    255));
hRedBrush = CreateSolidBrush(RGB( 255,      0,      0));
```

When you specify the window style, you set the background to use the blue brush in the window class structure:

```
wc.hbrBackground = hBlueBrush; // Use blue brush for background
```

Window size and position (previously set with `glutPositionWindow` and `glutReshapeWindow`) are set when the window is created:

```

// Create the main application window
hWnd = CreateWindow(
    lpszAppName,
    lpszAppName,
    WS_OVERLAPPEDWINDOW,
    100, 100,           // Size and dimensions of window
    250, 250,
    NULL,
    NULL,
    hInstance,
    NULL);

```

Finally, the actual painting of the window interior is handled by the `WM_PAINT` message handler in the `WndProc` function:

```

case WM_PAINT:
{
    PAINTSTRUCT ps;
    HBRUSH hOldBrush;
    // Start painting
    BeginPaint(hWnd,&ps);
    // Select and use the red brush
    hOldBrush = SelectObject(ps.hdc,hRedBrush);
    // Draw a rectangle filled with the currently
    // selected brush
    Rectangle(ps.hdc,100,100,150,150);
    // Deselect the brush
    SelectObject(ps.hdc,hOldBrush);
    // End painting
    EndPaint(hWnd,&ps);
}
break;

```

The call to `BeginPaint` prepares the window for painting and sets the `hdc` member of the `PAINTSTRUCT` structure to the device context to be used for drawing in this window. This handle to the device context is used as the first parameter to all GDI functions, identifying which window they should operate on. This code then selects the red brush for painting operations and draws a filled rectangle at the coordinates (100,100,150,150). Then the brush is deselected, and `EndPaint` cleans up the painting operation for you.

Before you jump to the conclusion that OpenGL should work in a similar way, remember that the GDI is Windows specific. Other environments do not have device contexts, window handles, and the like. Although the ideas may be similar, they are certainly not called the same thing and might work and behave differently. OpenGL, on the other hand, was designed to be completely portable among environments and hardware platforms (and it didn't start on Windows anyway!). Adding a device context parameter to the OpenGL functions would render your OpenGL code useless in any environment other than Windows.

OpenGL does have a context identifier, however, and it is called the *rendering context*. The rendering context is similar in many respects to the GDI device context because it is the rendering context that remembers current colors, state settings, and so on, much like the device context holds onto the current brush or pen color.

Pixel Formats

The Windows concept of the device context is limited for 3D graphics because it was designed for 2D graphics applications. In Windows, you request a device context identifier for a given window. The nature of the device context depends on the nature of the device. If your desktop is set to 16-

bit color, the device context Windows gives you knows about and understands 16-bit color only. You cannot tell Windows, for example, that one window is to be a 16-bit color window and another is to be an 8-bit color window.

Although Windows lets you create a memory device context, you still have to give it an existing window device context to emulate. Even if you pass in `NULL` for the window parameter, Windows uses the device context of your desktop. You, the programmer, have no control over the intrinsic characteristics of a windows device context.

Any window or device that will be rendering 3D graphics has far more characteristics to it than simply color depth, especially if you are using a hardware rendering device (3D graphics card). Up until now, GLUT has taken care of these details for you. When you initialized GLUT, you told it what buffers you needed (double or single color buffer, depth buffer, stencil, and alpha).

Before OpenGL can render into a window, you must first configure that window according to your rendering needs. Do you want hardware or software rendering? Will the rendering be single or double buffered? Do you need a depth buffer? How about stencil, destination alpha, or an accumulation buffer? After you set these parameters for a window, you cannot change them later. To switch from a window with only a depth and color buffer to a window with only a stencil and color buffer, you have to destroy the first window and re-create a new window with the characteristics you need.

Describing a Pixel Format

The 3D characteristics of the window are set one time, usually just after window creation. The collective name for these settings is the pixel format. Windows provides a structure named `PIXELFORMATDESCRIPTOR` that describes the pixel format. This structure is defined as follows:

```
typedef struct tagPIXELFORMATDESCRIPTOR {
    WORD nSize;           // Size of this structure
    WORD nVersion;        // Version of structure (should be 1)
    DWORD dwFlags;         // Pixel buffer properties
    BYTE iPixelType;      // Type of pixel data (RGBA or Color Index)
    BYTE cColorBits;       // Number of color bit planes in color buffer
    BYTE cRedBits;         // How many bits for red
    BYTE cRedShift;        // Shift count for red bits
    BYTE cGreenBits;       // How many bits for green
    BYTE cGreenShift;      // Shift count for green bits
    BYTE cBlueBits;         // How many bits for blue
    BYTE cBlueShift;        // Shift count for blue
    BYTE cAlphaBits;        // How many bits for destination alpha
    BYTE cAlphaShift;       // Shift count for destination alpha
    BYTE cAccumBits;        // How many bits for accumulation buffer
    BYTE cAccumRedBits;     // How many red bits for accumulation buffer
    BYTE cAccumGreenBits;   // How many green bits for accumulation buffer
    BYTE cAccumBlueBits;    // How many blue bits for accumulation buffer
    BYTE cAccumAlphaBits;   // How many alpha bits for accumulation buffer
    BYTE cDepthBits;         // How many bits for depth buffer
    BYTE cStencilBits;      // How many bits for stencil buffer
    BYTE cAuxBuffers;        // How many auxiliary buffers
    BYTE iLayerType;         // Obsolete - ignored
    BYTE bReserved;          // Number of overlay and underlay planes
    DWORD dwLayerMask;       // Obsolete - ignored
    DWORD dwVisibleMask;     // Transparent color of underlay plane
    DWORD dwDamageMask;      // Obsolete - ignored
} PIXELFORMATDESCRIPTOR;
```

For a given OpenGL device (hardware or software), the values of these members are not arbitrary. Only a limited number of pixel formats is available for a given window. Pixel formats are said to be exported by the OpenGL driver or software renderer. Most of these structure members are self-explanatory, but a few require some additional explanation:

<i>nSize</i>	The size of the structure; set to <code>sizeof(PIXELFORMATDESCRIPTOR)</code> .
<i>nVersion</i>	Set to 1.
<i>dwFlags</i>	A set of bit flags that specify properties of the pixel buffer. Most of these flags are not mutually exclusive, but a few are used only when requesting or describing the pixel format. Table 13.1 lists the valid flags for this member.
<i>iPixelType</i>	The type of color buffer. Only two values are valid: <code>PFD_TYPE_RGBA</code> and <code>PFD_TYPE_COLORINDEX</code> . <code>PFD_TYPE_COLORINDEX</code> allows you to request or describe the pixel format as color index mode. This rendering mode should be considered obsolete on modern PC hardware.
<i>cColorBits</i>	The number of bits of color depth in the color buffer. Typical values are 8, 16, 24, and 32. The 32-bit color buffers may or may not be used to store destination alpha values. Only Microsoft's generic implementation on Windows 2000, Windows XP, and later supports destination alpha.
<i>cRedBits</i>	The number of bits in the color buffer dedicated for the red color component.
<i>cGreenBits</i>	The number of bits in the color buffer dedicated for the green color component.
<i>cBlueBits</i>	The number of bits in the color buffer dedicated for the blue color component.
<i>cAlphaBits</i>	The number of bits used for the alpha buffer. Destination alpha is not supported by Microsoft's generic implementation, but many hardware implementations are beginning to support it.
<i>cAccumBits</i>	The number of bits used for the accumulation buffer.
<i>cDepthBits</i>	The number of bits used for the depth buffer. Typical values are 0, 16, 24, and 32. The more bits dedicated to the depth buffer, the more accurate depth testing will be.
<i>cStencilBits</i>	The number of bits used for the stencil buffer.
<i>cAuxBuffers</i>	The number of auxiliary buffers. In implementations that support auxiliary buffers, rendering can be redirected to an auxiliary buffer from the color buffer and swapped to the screen at a later time.
<i>iLayerType</i>	Obsolete (ignored).
<i>bReserved</i>	The number of overlay and underlay planes supported by the implementation. Bits 0 through 3 specify the number of overlay planes (up to 15), and bits 4 through 7 specify the number of underlay planes (also up to 15).
<i>dwLayerMask</i>	Obsolete (ignored).
<i>dwVisibleMask</i>	The transparent color of an underlay plane.
<i>dwDamageMask</i>	Obsolete (ignored).

Table 13.1. Valid Flags to Describe the Pixel Rendering Buffer

Bit Flag	Description

PFD_DRAW_TO_WINDOW	The buffer's output is displayed in a window.
PFD_DRAW_TO_BITMAP	The buffer's output is written to a Windows bitmap.
PFD_SUPPORT_GDI	The buffer supports Windows GDI drawing. Most implementations allow this only for single-buffered windows or bitmaps.
PFD_SUPPORT_OPENGL	The buffer supports OpenGL drawing.
PFD_GENERIC_ACCELERATED	The buffer is accelerated by an MCD device driver that accelerates this format.
PFD_GENERIC_FORMAT	The buffer is a rendered by a software implementation. This bit is also set with <code>PFD_GENERIC_ACCELERATED</code> for MCD drivers. Only if this bit is clear is the hardware driver an ICD.
PFD_NEED_PALETTE	The buffer is on a palette-managed device. This flag is set on Windows when running in 8-bit (256-color) mode and requires a 3-3-2 color palette.
PFD_NEED_SYSTEM_PALETTE	This flag indicates that OpenGL hardware supports rendering in 256-color mode. A 3-3-2 palette must be realized to enable hardware acceleration. Although documented, this flag can be considered obsolete. No mainstream hardware accelerator that supported accelerated rendering in 256-color mode ever shipped for Windows.
PFD_DOUBLEBUFFER	The color buffer is double buffered.
PFD_STEREO	The color buffer is stereoscopic. This is not supported by Microsoft's generic implementation. Most PC vendors that support stereo do so with custom extensions for their hardware.
PFD_SWAP_LAYER_BUFFERS	This flag is used if overlay and underlay planes are supported. If set, these planes may be swapped independently of the color buffer.
PFD_DEPTH_DONTCARE	This flag is used only when requesting a pixel format. It indicates that you do not need a depth buffer. Some implementations can save memory and enhance performance by not allocating memory for the depth buffer.
PFD_DOUBLE_BUFFER_DONTCARE	This flag is used only when requesting a pixel format. It indicates that you do not plan to use double buffering. Although you can force rendering to the front buffer only, this flag allows an implementation to save memory and potentially enhance performance.

Enumerating Pixel Formats

The pixel format for a window is identified by a one-based integer index number. An implementation exports a number of pixel formats from which to choose. To set a pixel format for a window, you must select one of the available formats exported by the driver. You can use the `DescribePixelFormat` function to determine the characteristics of a given pixel format. You can also use this function to find out how many pixel formats are exported by the driver. The following code shows how to enumerate all the pixel formats available for a window:

```
PIXELFORMATDESCRIPTOR pfd;           // Pixel format descriptor
int nFormatCount;                  // How many pixel formats exported
...
// Get the number of pixel formats
// Will need a device context
pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);
```

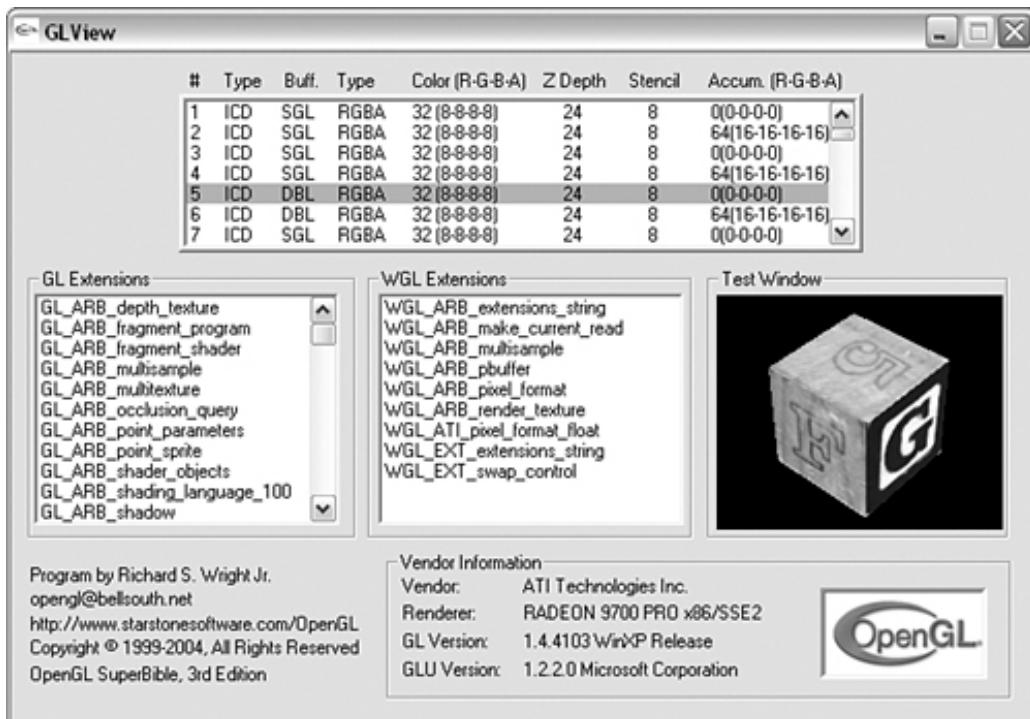
```

nFormatCount = DescribePixelFormat(hDC, 1, 0, NULL);
// Retrieve each pixel format
for(int i = 1; i <= nFormatCount; i++)
{
    // Get description of pixel format
    DescribePixelFormat(hDC, i, pfd.nSize, &pfd);
. . .
. . .
}

```

The `DescribePixelFormat` function returns the maximum pixel format index. You can use an initial call to this function as shown to determine how many are available. The CD includes an interesting utility program called `GLView` for this chapter. This program enumerates all pixel formats available for your display driver for the given resolution and color depths. [Figure 13.2](#) shows the output from this program when a double-buffered pixel format is selected. (A single-buffered pixel format would contain a blinking block animation.)

Figure 13.2. The GLView program shows all pixel formats for a given device.



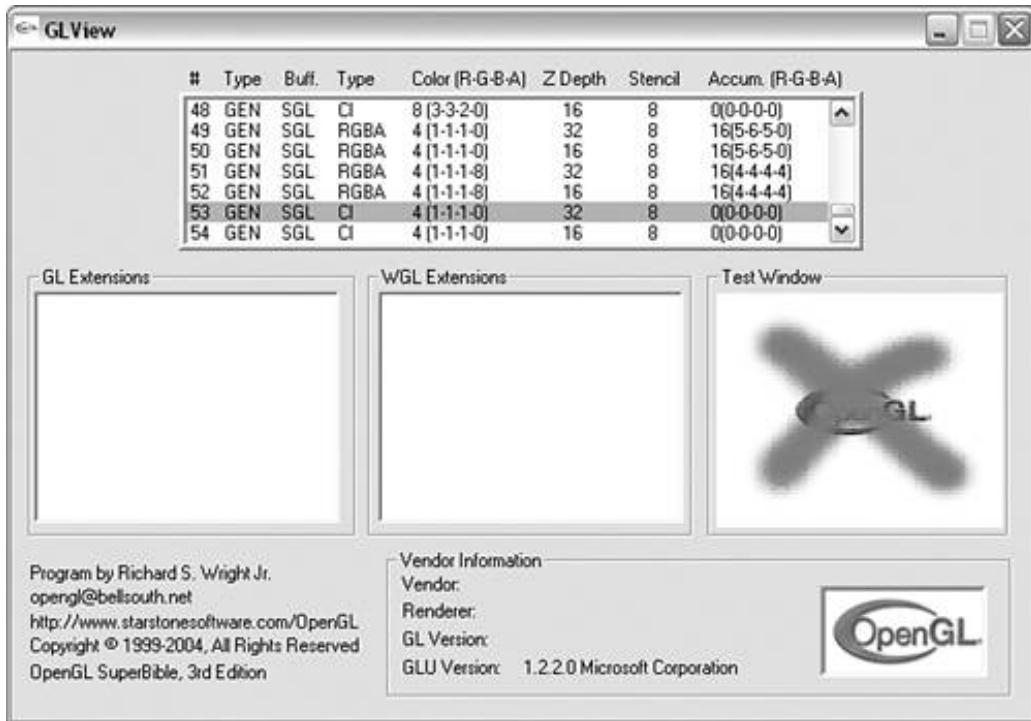
The Microsoft Foundation Classes (MFC) source code is included on the CD for `GLView`. This is a bit more complex than your typical sample program, and `GLView` is provided more as a tool for your use than as a programming example. The important code for enumerating pixel formats was presented earlier and is less than a dozen lines long. If you are familiar with MFC already, examination of this source code will show you how to integrate OpenGL rendering into any `CWnd` derived window class.

The list box lists all the available pixel formats and displays their characteristics (driver type, color depth, and so on). A sample window in the lower-right corner displays a rotating cube using a window created with the highlighted pixel format. The `glGetString` function enables you to find out the name of the vendor for the OpenGL driver, as well as other version information. Finally, a list box displays all the OpenGL and WGL extensions exported by the driver (WGL extensions are covered later in this chapter).

If you experiment with this program, you'll discover that not all pixel formats can be used to create an OpenGL window, as shown in [Figure 13.3](#). Even though the driver exports these pixel formats, it does not mean that you can create an OpenGL-enabled window with one of them. The

most important criterion is that the pixel format color depth must match the color depth of your desktop. That is, you can't create a 16-bit color pixel format for a 32-bit color desktop, or vice versa.

Figure 13.3. The GLView program showing an invalid pixel format.



Make special note of the fact that at least 24 pixel formats are always enumerated, sometimes more. If you are running the Microsoft generic implementation, you will see exactly 24 pixel formats listed (all belonging to Microsoft). If you have a hardware accelerator (either an MCD or ICD), you'll note that the accelerated pixel formats are listed first, followed by the 24 generic pixel formats belonging to Microsoft. This means that when hardware acceleration is present, you actually can choose from two implementations of OpenGL. The first are the hardware-accelerated pixel formats belonging to the hardware accelerator. The second are the pixel formats for Microsoft's software implementation.

Knowing this bit of information can be useful. For one thing, it means that a software implementation is always available for rendering to bitmaps or printer devices. It also means that if you so desire (for debugging purposes, perhaps), you can force software rendering, even when an application might typically select hardware acceleration.

Selecting and Setting a Pixel Format

Enumerating all the available pixel formats and examining each one to find one that meets your needs could turn out to be quite tedious. Windows provides a shortcut function that makes this process somewhat simpler. The `ChoosePixelFormat` function allows you to create a pixel format structure containing the desired attributes of your 3D window. The `ChoosePixelFormat` function then finds the closest match possible (with preference for hardware-accelerated pixel formats) and returns the most appropriate index. The pixel format is then set with a call to another new Windows function, `SetPixelFormat`. The following code segment shows the use of these two functions:

```
int nPixelFormat;  
. . .  
static PIXELFORMATDESCRIPTOR pfd = {  
    sizeof(PIXELFORMATDESCRIPTOR), // Size of this structure  
    1, // Version of this structure
```

```

PFD_DRAW_TO_WINDOW |          // Draw to window (not to bitmap)
PFD_SUPPORT_OPENGL |        // Support OpenGL calls in window
PFD_DOUBLEBUFFER,          // Double buffered mode
PFD_TYPE_RGBA,             // RGBA color mode
32,                        // Want 32-bit color
0,0,0,0,0,0,               // Not used to select mode
0,0,                      // Not used to select mode
0,0,0,0,0,0,               // Not used to select mode
16,                        // Size of depth buffer
0,                          // No stencil
0,                          // No auxiliary buffers
0,                          // Obsolete or reserved
0,                          // No overlay and underlay planes
0,                          // Obsolete or reserved layer mask
0,                          // No transparent color for underlay plane
0};                         // Obsolete

// Choose a pixel format that best matches that described in pfd
// for the given device context
nPixelFormat = ChoosePixelFormat(hDC, &pf);
// Set the pixel format for the device context
SetPixelFormat(hDC, nPixelFormat, &pf);

```

Initially, the `PIXELFORMATDESCRIPTOR` structure is filled with the desired characteristics of the 3D-enabled window. In this case, you want a double-buffered pixel format that renders into a window, so you request 32-bit color and a 16-bit depth buffer. If the current implementation supports 24-bit color at best, the returned pixel format will be a valid 24-bit color format. Depth buffer resolution is also subject to change. An implementation might support only a 24-bit or 32-bit depth buffer. In any case, `ChoosePixelFormat` always returns a valid pixel format, and if at all possible, it returns a hardware-accelerated pixel format.

Some programmers and programming needs might require more sophisticated selection of a pixel format. In these cases, you need to enumerate and inspect all available pixel formats or use the WGL extension presented later in this chapter. For most uses, however, the preceding code is sufficient to prime your window to receive OpenGL rendering commands.

The OpenGL Rendering Context

A typical Windows application can consist of many windows. You can even set a pixel format for each one (using that windows device context) if you want! But `SetPixelFormat` can be called only once per window. When you call an OpenGL command, how does it know which window to send its output to? In the previous chapters, we used the GLUT framework, which provided a single window to display OpenGL output. Recall that with normal Windows GDI-based drawing, each window has its own device context.

To accomplish the portability of the core OpenGL functions, each environment must implement some means of specifying a current rendering window before executing any OpenGL commands. Just as the Windows GDI functions use the windows device contexts, the OpenGL environment is embodied in what is known as the *rendering context*. Just as a device context remembers settings about drawing modes and commands for the GDI, the rendering context remembers OpenGL settings and commands.

You create an OpenGL rendering context by calling the `wglCreateContext` function. This function takes one parameter: the device context of a window with a valid pixel format. The data type of an OpenGL rendering context is `HGLRC`. The following code shows the creation of an OpenGL rendering context:

```

HGLRC hRC;      // OpenGL rendering context
HDC hDC;        // Windows device context
...
// Select and set a pixel format

```

```
    . . .
hRC = wglCreateContext(hDC);
```

A rendering context is created that is compatible with the window for which it was created. You can have more than one rendering context in your application—for instance, two windows that are using different drawing modes, perspectives, and so on. However, for OpenGL commands to know which window they are operating on, only one rendering context can be active at any one time per thread. When a rendering context is made active, it is said to be *current*.

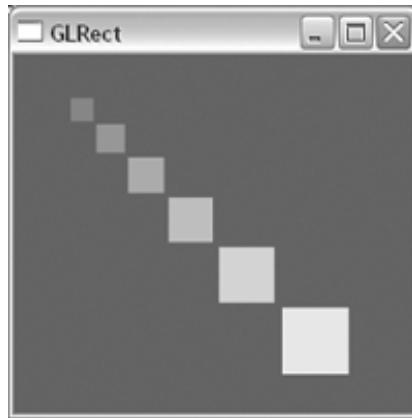
When made current, a rendering context is also associated with a device context and thus with a particular window. Now, OpenGL knows which window into which to render. You can even move an OpenGL rendering context from window to window, but each window must have the same pixel format. To make a rendering context current and associate it with a particular window, you call the `wglGetCurrent` function. This function takes two parameters, the device context of the window and the OpenGL rendering context:

```
void wglGetCurrent(HDC hDC, HGLRC hRC);
```

Putting It All Together

We've covered a lot of ground over the past several pages. We've described each piece of the puzzle individually, but now let's look at all the pieces put together. In addition to seeing all the OpenGL-related code, we should examine some of the minimum requirements for any Windows program to support OpenGL. Our sample program for this section is GLRECT. It should look somewhat familiar because it is also the first GLUT-based sample program from [Chapter 2](#). Now, however, the program is a full-fledged Windows program written with nothing but C and the Win32 API. [Figure 13.4](#) shows the output of the new program, complete with bouncing square.

Figure 13.4. Output from the GLRECT program with bouncing square.



Creating the Window

The starting place for any Windows-based GUI program is the `WinMain` function. In this function, you register the window type, create the window, and start the message pump. [Listing 13.1](#) shows the `WinMain` function for the first sample.

Listing 13.1. The `WinMain` Function of the GLRECT Sample Program

```
// Entry point of all Windows programs
int APIENTRY WinMain(    HINSTANCE      hInstance,
                        HINSTANCE      hPrevInstance,
                        LPSTR          lpCmdLine,
                        int            nCmdShow)
{
```

```

MSG      msg;          // Windows message structure
WNDCLASS  wc;          // Windows class structure
HWND      hWnd;         // Storage for window handle
// Register window style
wc.style      = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
wc.lpfnWndProc = (WNDPROC) WndProc;
wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
wc.hInstance   = hInstance;
wc.hIcon       = NULL;
wc.hCursor     = LoadCursor(NULL, IDC_ARROW);
// No need for background brush for OpenGL window
wc.hbrBackground = NULL;
wc.lpszMenuName = NULL;
wc.lpszClassName = lpszAppName;
// Register the window class
if(RegisterClass(&wc) == 0)
    return FALSE;

// Create the main application window
hWnd = CreateWindow(
    lpszAppName,
    lpszAppName,
    // OpenGL requires WS_CLIPCHILDREN and WS_CLIPSIBLINGS
    WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN | WS_CLIPSIBLINGS,
    // Window position and size
    100, 100,
    250, 250,
    NULL,
    NULL,
    hInstance,
    NULL);

// If window was not created, quit
if(hWnd == NULL)
    return FALSE;
// Display the window
ShowWindow(hWnd, SW_SHOW);
UpdateWindow(hWnd);
// Process application messages until the application closes
while( GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
return msg.wParam;
}

```

This listing pretty much contains your standard Windows GUI startup code. Only two points really bear mentioning here. The first is the choice of window styles set in `CreateWindow`. You can generally use whatever window styles you like, but you do need to set the `WS_CLIPCHILDREN` and `WS_CLIPSIBLINGS` styles. These styles were required in earlier versions of Windows, but later versions have dropped them as a strict requirement. The purpose of these styles is to keep the OpenGL rendering context from rendering into other windows, which can happen in GDI. However, an OpenGL rendering context must be associated with only one window at a time.

The second note you should make about this startup code is the use of `CS_OWNDC` for the window style. Why you need this innocent-looking flag requires a bit more explanation. You need a device context for both GDI rendering and for OpenGL double-buffered page flipping. To understand what `CS_OWNDC` has to do with this, you first need to take a step back and review the purpose and use of a windows device context.

First, You Need a Device Context

Before you can draw anything in a window, you first need a windows device context. You need it whether you're doing OpenGL, GDI, or even DirectX programming. Any drawing or painting operation in Windows (even if you're drawing on a bitmap in memory) requires a device context that identifies the specific object being drawn on. You retrieve the device context to a window with a simple function call:

```
HDC hDC = GetDC(hWnd);
```

The *hDC* variable is your handle to the device context of the window identified by the window handle *hWnd*. You use the device context for all GDI functions that draw in the window. You also need the device context for creating an OpenGL rendering context, making it current, and performing the buffer swap. You tell Windows that you don't need the device context for the window any longer with another simple function call, using the same two values:

```
ReleaseDC(hWnd, hDC);
```

The standard Windows programming wisdom is that you retrieve a device context, use it for drawing, and then release it again as soon as possible. This advice dates back to the pre-Win32 days; under Windows 3.1 and earlier, you had a small pool of memory allocated for system resources, such as the windows device context. What happened when Windows ran out of system resources? If you were lucky, you got an error message. If you were working on something really important, the operating system could somehow tell, and it would instead crash and take all your work with it. Well, at least it seemed that way!

The best way to spare your users this catastrophe was to make sure that the *GetDC* function succeeded. If you did get a device context, you did all your work as quickly as possible (typically within one message handler) and then released the device context so that other programs could use it. The same advice applied to other system resources such as pens, fonts, brushes, and so on.

Enter Win32

Windows NT and the subsequent Win32-based operating systems were a tremendous blessing for Windows programmers, in more ways than can be recounted here. Among their many benefits was that you could have all the system resources you needed until you exhausted available memory or your application crashed. (At least it wouldn't crash the OS!) It turns out that the *GetDC* function is, in computer time, quite an expensive function call to make. If you got the device context when the window was created and hung on to it until the window was destroyed, you could speed up your window painting considerably. You could hang on to brushes, fonts, and other resources that would have to be created or retrieved and potentially reinitialized each time the window was invalidated.

One popular example of this Win32 benefit was a program that created random rectangles and put them in random locations in the window. (This was a GDI sample.) The difference between code written the old way and code written the new way was astonishingly obvious. Wow! Win32 was great!

Three Steps Forward, Two Steps Back

Windows 95, 98, and ME brought Win32 programming to the mainstream, but still had a few of the old 16-bit limitations deep down in the plumbing. The situation with losing system resources was considerably improved, but it was not eliminated entirely. The operating system could still run out of resources, but (according to Microsoft) it was unlikely. Alas, life is not so simple. Under Windows NT, when an application terminates, all allocated system resources are automatically returned to the operating system. Under Windows 95, 98, or ME, you have a resource leak if the program crashes or the application fails to release the resources it allocated. Eventually, you will start to stress the system, and you can run out of system resources (or device contexts).

What happens when Windows doesn't have enough device contexts to go around? Well, it just takes one from someone who is being a hog with them. This means that if you call `GetDC` and don't call `ReleaseDC`, Windows 95, 98, or ME might just appropriate your device context when it becomes stressed. The next time you call `wglMakeCurrent` or `SwapBuffers`, your device context handle might not be valid. Your application might crash or mysteriously stop rendering. Ask someone in customer support how well it goes over when you try to explain to a customer that his or her problem with your application is really Microsoft's fault!

All Is Not Lost

You actually have a way to tell Windows to create a device context just for your window's use. This feature is useful because every time you call `GetDC`, you have to reselect your fonts, the mapping mode, and so on. If you have your own device context, you can do this sort of initialization only once. Plus, you don't have to worry about your device context handle being yanked out from under you. Doing this is simple: You simply specify `CS_OWNDC` as one of your class styles when you register the window. A common error is to use `CS_OWNDC` as a window style when you call `Create`. There are window styles and there are class styles, but you can't mix and match.

Code to register your window style generally looks something like this:

```
WNDCLASS wc; // Windows class structure
...
...
// Register window style
wc.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
wc.lpfnWndProc = (WNDPROC) WndProc;
...
...
wc.lpszClassName = lpszAppName;
// Register the window class
if(RegisterClass(&wc) == 0)
return FALSE;
```

You then specify the class name when you create the window:

```
hWnd = CreateWindow( wc.lpszClassName, szWindowName, ...
```

Graphics programmers should always use `CS_OWNDC` in the window class registration. This ensures that you have the most robust code possible on any Windows platform. Another consideration is that many older OpenGL hardware drivers have bugs because they expect `CS_OWNDC` to be specified. They might have been originally written for NT, so the drivers do not account for the possibility that the device context might become invalid. The driver might also trip up if the device context does not retain its configuration (as is the case in the `GetDC/ReleaseDC` scenario).

Regardless of the specifics, some drivers are not very stable unless you specify the `CS_OWNDC` flag. Many, if not most, vendors are addressing this well-known issue as their drivers mature. Still, the other reasons outlined here provide plenty of incentive to make what is basically a minor code adjustment.

Using the OpenGL Rendering Context

The real meat of the GLRECT sample program is in the window procedure, `WndProc`. The window procedure receives window messages from the operating system and responds appropriately. This model of programming, called *message* or *event-driven programming*, is the foundation of the modern Windows GUI.

When a window is created, it first receives a `WM_CREATE` message from the operating system. This is the ideal location to create and set up the OpenGL rendering context. A window also receives a `WM_DESTROY` message when it is being destroyed. Naturally, this is the ideal place to put cleanup code. [Listing 13.2](#) shows the `SetDCPixelFormat` format, which is used to select and set the pixel format, along with the window procedure for the application. This function contains the same basic functionality that we have been using with the GLUT framework.

Listing 13.2. Setting the Pixel Format and Handling the Creation and Deletion of the OpenGL Rendering Context

```
///////////
// Select the pixel format for a given device context
void SetDCPixelFormat(HDC hDC)
{
    int nPixelFormat;
    static PIXELFORMATDESCRIPTOR pfd = {
        sizeof(PIXELFORMATDESCRIPTOR), // Size of this structure
        1,                          // Version of this structure
        PFD_DRAW_TO_WINDOW |        // Draw to window (not to bitmap)
        PFD_SUPPORT_OPENGL |       // Support OpenGL calls in window
        PFD_DOUBLEBUFFER,           // Double-buffered mode
        PFD_TYPE_RGBA,              // RGBA color mode
        32,                         // Want 32-bit color
        0,0,0,0,0,0,                // Not used to select mode
        0,0,                      // Not used to select mode
        0,0,0,0,0,0,                // Not used to select mode
        16,                         // Size of depth buffer
        0,                          // Not used here
        0,0,0 };                   // Not used here

    // Choose a pixel format that best matches that described in pfd
    nPixelFormat = ChoosePixelFormat(hDC, &pfd);
    // Set the pixel format for the device context
    SetPixelFormat(hDC, nPixelFormat, &pfd);
}

///////////
// Window procedure, handles all messages for this program
LRESULT CALLBACK WndProc(HWND hWnd,
    UINT message,
    WPARAM wParam,
    LPARAM lParam)
{
    static HGLRC hRC = NULL;      // Permanent rendering context
    static HDC hDC = NULL;        // Private GDI device context
    switch (message)
    {
        // Window creation, set up for OpenGL
        case WM_CREATE:
            // Store the device context
            hDC = GetDC(hWnd);
            // Select the pixel format
            SetDCPixelFormat(hDC);
            // Create the rendering context and make it current
            hRC = wglCreateContext(hDC);
            wglMakeCurrent(hDC, hRC);
            // Create a timer that fires 30 times a second
            SetTimer(hWnd, 33, 1, NULL);
            break;
        // Window is being destroyed, clean up
    }
}
```

```

case WM_DESTROY:
    // Kill the timer that we created
    KillTimer(hWnd,101);
    // Deselect the current rendering context and delete it
    wglGetCurrentContext(hDC,NULL);
    wglDeleteContext(hRC);
    // Tell the application to terminate after the window
    // is gone.
    PostQuitMessage(0);
    break;
// Window is resized.
case WM_SIZE:
    // Call our function which modifies the clipping
    // volume and viewport
    ChangeSize(LOWORD(lParam), HIWORD(lParam));
    break;
// Timer moves and bounces the rectangle, simply calls
// our previous OnIdle function, then invalidates the
// window so it will be redrawn.
case WM_TIMER:
{
    IdleFunction();
    InvalidateRect(hWnd,NULL,FALSE);
}
break;
// The painting function. This message is sent by Windows
// whenever the screen needs updating.
case WM_PAINT:
{
    // Call OpenGL drawing code
    RenderScene();
    // Call function to swap the buffers
    SwapBuffers(hDC);
    // Validate the newly painted client area
    ValidateRect(hWnd,NULL);
}
break;
. . .
default: // Passes it on if unprocessed
    return (DefWindowProc(hWnd, message, wParam, lParam));
}
return (0L);
}

```

Initializing the Rendering Context

The first thing you do when the window is being created is retrieve the device context (remember, you hang on to it) and set the pixel format:

```

// Store the device context
hDC = GetDC(hWnd);
// Select the pixel format
SetDCPixelFormat(hDC);

```

Then you create the OpenGL rendering context and make it current:

```

// Create the rendering context and make it current
hRC = wglCreateContext(hDC);
wglGetCurrentContext(hDC, hRC);

```

The last task you handle while processing the `WM_CREATE` message is creating a Windows timer for the window. You will use this shortly to affect the animation loop:

```
// Create a timer that fires 30 times a second
SetTimer(hWnd, 33, 1, NULL);
break;
```

At this point, the OpenGL rendering context has been created and associated with a window with a valid pixel format. From this point forward, all OpenGL rendering commands will be routed to this context and window.

Shutting Down the Rendering Context

When the window procedure receives the `WM_DESTROY` message, the OpenGL rendering context must be deleted. Before you delete the rendering context with the `wglDeleteContext` function, you must first call `wglGetCurrent` again, but this time with `NULL` as the parameter for the OpenGL rendering context:

```
// Deselect the current rendering context and delete it
wglGetCurrent(hdc, NULL);
wglDeleteContext(hrc);
```

Before deleting the rendering context, you should delete any display lists, texture objects, or other OpenGL-allocated memory.

Other Windows Messages

All that is required to enable OpenGL to render into a window is creating and destroying the OpenGL rendering context. However, to make your application well behaved, you need to follow some conventions with respect to message handling. For example, you need to set the viewport when the window changes size, by handling the `WM_SIZE` message:

```
// Window is resized.
case WM_SIZE:
    // Call our function which modifies the clipping
    // volume and viewport
    ChangeSize(LOWORD(lParam), HIWORD(lParam));
    break;
```

The processing that happens in response to the `WM_SIZE` message is the same as in the function you handed off to `glutReshapeFunc` in GLUT-based programs. The window procedure also receives two parameters: `lParam` and `wParam`. The low word of `lParam` is the new width of the window, and the high word is the height.

This example uses the `WM_TIMER` message handler to do the idle processing. The process is not really idle, but the previous call to `SetTimer` causes the `WM_TIMER` message to be received on a fairly regular basis (*fairly* because the exact interval is not guaranteed).

Other Windows messages handle things such as keyboard activity (`WM_CHAR`, `WM_KEYDOWN`), mouse movements (`WM_MOUSEMOVE`), and palette management. (We discuss these messages shortly.)

The `WM_PAINT` Message

The `WM_PAINT` message bears a closer examination. This message is sent to a window whenever Windows needs to draw or redraw its contents. To tell Windows to redraw a window anyway, you invalidate the window with one function call in the `WM_TIMER` message handler:

```

IdleFunction();
InvalidateRect(hWnd,NULL, FALSE);

```

Here, `IdleFunction` updates the position of the square, and `InvalidateRect` tells Windows to redraw the window (now that the square has moved).

Most Windows programming books show you a `WM_PAINT` message handler with the well-known `BeginPaint/EndPaint` function pairing. `BeginPaint` retrieves the device context so it can be used for GDI drawing, and `EndPaint` releases the context and validates the window. In our previous discussion of why you need the `CS_OWNDC` class style, we pointed out that using this function pairing is generally a bad idea for high-performance graphics applications. The following code shows roughly the equivalent functionality, without quite so much overhead:

```

// The painting function. This message is sent by Windows
// whenever the screen needs updating.
case WM_PAINT:
{
    // Call OpenGL drawing code
    RenderScene();
    // Call function to swap the buffers
    SwapBuffers(hDC);
    // Validate the newly painted client area
    ValidateRect(hWnd,NULL);
}
break;

```

Because this example has a device context (`hDC`), you don't need to continually get and release it. We've mentioned the `SwapBuffers` function previously but not fully explained it. This function takes the device context as an argument and performs the buffer swap for double-buffered rendering. This is why you need the device context readily available when rendering.

Notice that you must manually validate the window with the call to `ValidateRect` after rendering. Without the `BeginPaint/EndPaint` functionality in place, there is no way to tell Windows that you have finished drawing the window contents. One alternative to using `WM_TIMER` to invalidate the window (thus forcing a redraw) is to simply not validate the window. If the window procedure returns from a `WM_PAINT` message and the window is not validated, the operating system generates another `WM_PAINT` message. This chain reaction causes an endless stream of repaint messages. One problem with this approach to animation is that it can leave little opportunity for other window messages to be processed. Although rendering might occur very quickly, the user might find it difficult or impossible to resize the window or use the menu, for example.

Windows Palettes

In [Chapter 5](#), "Color, Materials, and Lighting: The Basics," we discussed the various color modes available on the modern PC running Windows. Hardware-accelerated 3D graphics cards for the PC support 16-bit or higher color resolutions. If you drop down to 8-bit color (256 colors), you most likely are running Microsoft's generic software implementation. Although this graphics mode is becoming less common, your application could still find itself running in such an environment. Not all 3D applications require hardware acceleration, and many users might not even care about hardware versus software rendering.

Color Matching

What happens when you try to draw a pixel of a particular color using the RGB values in `glColor`? If the PC graphics card is in 24-bit color mode, each pixel is displayed precisely in the color specified by the 24-bit value (three 8-bit intensities). In the 15- and 16-bit color modes, Windows passes the 24-bit color value to the display driver, which reduces the color to a 15- or 16-bit color

value before displaying it. Internal color calculations due to lighting and texturing are usually (depending on the implementation) done at full precision. Reducing a color's precision from 24-bit to 16-bit results in some loss of visual fidelity but can be acceptable for many applications.

On a Windows display with only 8 bits of color resolution (256 colors), Windows creates a palette of colors for the display device. A palette is a list of color values specified at full color. When an application needs to specify one of these colors, it does so by index rather than by specifying the exact color. In practice, the color entries in a palette can be arbitrary and are often chosen to match a particular application's needs.

When Windows is running in a color mode that supports 256 colors, it would make sense if those colors were evenly distributed across RGB colorspace. (See [the color cube example in Chapter 5](#).) Then all applications would have a relatively wide choice of colors, and when a color was selected, the nearest available color would be used. This is exactly the type of palette that OpenGL requires when running in a palettized color mode. Unfortunately, this arrangement is not always practical for other applications.

Because the 256 colors in the palette for the device can be selected from more than 16 million different colors, an application can substantially improve the quality of its graphics by carefully selecting those colors—and many do. For example, to produce a seascape, additional shades of blue might be needed. CAD and modeling applications can modify the palette to produce smooth shading of a surface of a particular single color. For example, the scene might require as many as 200 shades of gray to accurately render the image of a pipe's cross-section. Thus, applications for the PC typically change the palette to meet their needs, resulting in near-photographic quality for many images and scenes. For 256-color bitmaps, the Windows .BMP format even has an array that's 256 entries long, containing 24-bit RGB values specifying the palette for the stored image.

An application can create a palette with the Windows `CreatePalette` function, identifying the palette by a handle of type `HPALETTE`. This function takes a logical palette structure (`LOGPALETTE`) that contains 256 entries, each specifying 8-bit values for red, green, and blue components. Before we examine palette creation, let's look at how multitasked applications can share the single system palette in 8-bit color mode.

Palette Arbitration

Windows multitasking allows many applications to be onscreen at once. If the hardware supports only 256 colors onscreen at once, all applications must share the same system palette. If one application changes the system palette, images in the other windows might have scrambled colors, producing some undesired psychedelic effects. To arbitrate palette usage among applications, Windows sends a set of messages. Applications are notified when another application has changed the system palette, and they are notified when their window has received focus and palette modification is allowed.

When an application receives keyboard or mouse input focus, Windows sends a `WM_QUERYNEWPALETTE` message to the main window of the application. This message asks the application whether it wants to realize a new palette. Realizing a palette means the application copies the palette entries from its private palette to the system palette. To do this, the application must first select the palette into the device context for the window being updated and then call `RealizePalette`.

Another message sent by Windows for palette realization is `WM_PALETTECHANGED`. This message is sent to windows that can realize their palette but might not have the current focus. When this message is sent, you must also check the value of `wParam`. If `wParam` contains the handle to the current window receiving the message, then `WM_QUERYNEWPALETTE` has already been processed, and the palette does not need to be realized again. [Listing 13.3](#) shows the message handler for these two messages.

Listing 13.3. Message Handlers for Windows Palette Management

```

///////////
// Windows is telling the application that it may modify
// the system palette. This message in essence asks the
// application for a new palette.
case WM_QUERYNEWPALETTE:
    // If the palette was created.
    if(hPalette)
    {
        int nRet;
        // Selects the palette into the current device context
        SelectPalette(hDC, hPalette, FALSE);
        // Map entries from the currently selected palette to
        // the system palette. The return value is the number
        // of palette entries modified.
        nRet = RealizePalette(hDC);
        // Repaint, forces remap of palette in current window
        InvalidateRect(hWnd,NULL,FALSE);
        return nRet;
    }
break;

///////////
// This window may set the palette, even though it is not the
// currently active window.
case WM_PALETTECHANGED:
    // Don't do anything if the palette does not exist or if
    // this is the window that changed the palette.
    if((hPalette != NULL) && ((HWND)wParam != hWnd))
    {
        // Select the palette into the device context
        SelectPalette(hDC,hPalette,FALSE);
        // Map entries to system palette
        RealizePalette(hDC);
        // Remap the current colors to the newly realized palette
        UpdateColors(hDC);
        return 0;
    }
break;

```

A Windows palette is identified by a handle of type `HPALETTE`. The `hPalette` variable shown in [Listing 13.3](#) is this type. Note that the value of `hPalette` is checked against `NULL` before either of these palette-realization messages is processed to check for a potential error. If the application is not running in 8-bit color mode, these messages are not posted to your application.

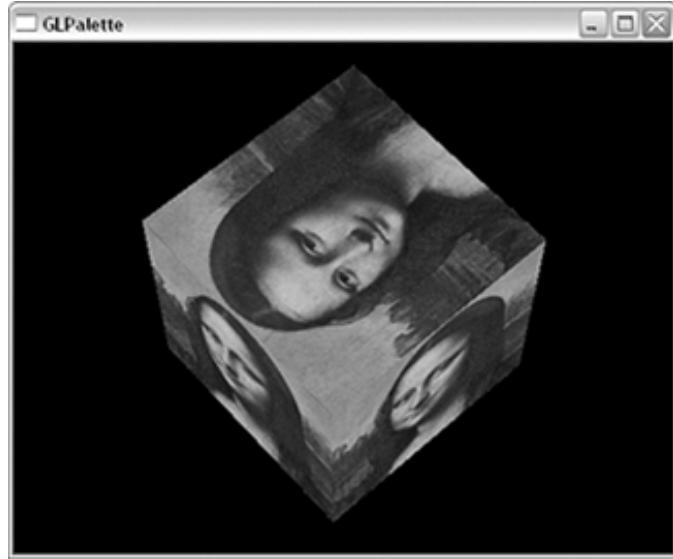
Creating a Palette for OpenGL

Unfortunately, palette considerations are a necessary evil if your application is to run on the 8-bit hardware that's still widely in use. What do you do if your code is executing on a machine that supports only 256 colors?

For an application such as image reproduction, we recommend selecting a range of colors that closely match the original colors. Selecting the best reduced palette for a given full-color image has been the subject of much study over the years and is well beyond the scope of this book. For OpenGL rendering under most circumstances, you want the widest possible range of colors for general-purpose use. The trick is to select the palette colors so that they're evenly distributed throughout the color cube. Then, whenever a color not already in the palette is specified, Windows will select the nearest color in the color cube. As mentioned earlier, this arrangement is not ideal for some applications, but for OpenGL-rendered scenes, it is the best you can do. Unless the scene has substantial texture mapping with a wide variety of colors, the results are usually acceptable.

The sample program GLPALETTE, shown in [Figure 13.5](#), demonstrates the results. This program creates a spinning cube textured on each side with a familiar face. Run this program on your PC in both full-color (16-bit or higher) and 256-color mode. The effect can't be accurately reproduced as a grayscale image in this book, but you can see that even in 8-bit color mode, OpenGL is able to reproduce the image quite well, despite the limited range of colors available.

Figure 13.5. The Mona Lisa cube, quite recognizable even in 8-bit color mode.



Do You Need a Palette?

To determine whether your application needs a palette, you examine `PIXELFORMATDESCRIPTOR` returned by a call to `DescribePixelFormat`. Test the `dwFlags` member of the `PIXELFORMATDESCRIPTOR` structure, and if the bit value `PFD_NEED_PALETTE` is set, you need to create a palette for your application:

```
DescribePixelFormat(hDC, nPixelFormat, sizeof(PIXELFORMATDESCRIPTOR), &pf);
// Does this pixel format require a palette?
if(!(pf.dwFlags & PFD_NEED_PALETTE))
    return NULL; // Does not need a palette
// Palette creation code
...
...
```

The Palette's Structure

To create a palette, you must first allocate memory for a Windows `LOGPALETTE` structure. This structure is filled with the information that describes the palette and then is passed to the Win32 function `CreatePalette`. The `LOGPALETTE` structure is defined as follows:

```
typedef struct tagLOGPALETTE { // lgpl
    WORD      palVersion;
    WORD      palNumEntries;
    PALETTEENTRY palPalEntry[1];
} LOGPALETTE;
```

The first two members are the palette header and contain the palette version (always set to `0x300`) and number of color entries (256 for 8-bit modes). Each entry is then defined as a `PALETTEENTRY` structure that contains the RGB components of the color entry. Additional entries are located at the end of the structure in memory.

The following code allocates space for the logical palette:

```
LOGPALETTE *pPal;           // Pointer to memory for logical palette
...
...
// Allocate space for a logical palette structure plus all the palette
// entries
pPal = (LOGPALETTE*)malloc(sizeof(LOGPALETTE) + nColors*sizeof(PALETTEENTRY));
```

Here, `nColors` specifies the number of colors to place in the palette, which for our purposes is always 256.

Each entry in the palette, then, is a `PALETTEENTRY` structure, which is defined as follows:

```
typedef struct tagPALETTEENTRY { // pe
    BYTE peRed;
    BYTE peGreen;
    BYTE peBlue;
    BYTE peFlags;
} PALETTEENTRY;
```

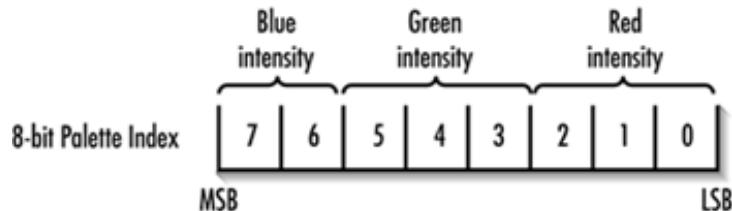
The `peRed`, `peGreen`, and `peBlue` members specify an 8-bit value that represents the relative intensities of each color component. In this way, each of the 256 palette entries contains a 24-color definition. The `peFlags` member describes advanced use of the palette entries. For OpenGL purposes, you can just set it to `NULL`.

The 3-3-2 Palette

Now comes the tricky part: Not only must the 256 palette entries be spread evenly throughout the RGB color cube, but also they must be in a certain order. It is this order that enables OpenGL to find the color it needs or the closest available color in the palette. In 8-bit color mode, you have 3 bits each for red and green color components and 2 bits for the blue component. This is commonly referred to as a *3-3-2 palette*. So the RGB color cube measures 8x8x3 along the red, green, and blue axes, respectively.

To find the color needed in the palette, an 8-8-8 color reference (the 24-bit color mode setup) is scaled to a 3-3-2 color value. This 8-bit value is then the index into the palette array. The red intensities of 0 to 7 in the 3-3-2 palette must correspond to the intensities 0 to 255 in the 8-8-8 palette. [Figure 13.6](#) illustrates how the red, green, and blue components are combined to make the palette index.

Figure 13.6. Sample of 3-3-2 palette packing.



When you build the palette, you loop through all values from 0 to 255. You then decompose the index into the red, green, and blue intensities represented by these values (in terms of the 3-3-2 palette). Each component is multiplied by 255 and divided by the maximum value represented, which has the effect of smoothly stepping the intensities from 0 to 7 for red and green and from 0 to 3 for the blue. [Table 13.2](#) shows some sample palette entries to demonstrate component calculation.

Table 13.2. A Few Sample Palette Entries for a 3-3-2 Palette

Palette Entry	Binary (B G R)	Blue Component	Green Component	Red Component
0	00 000 000	0	0	0
1	00 000 001	0	0	1*255/7
2	00 000 010	0	0	2*255/7
3	00 000 011	0	0	3*255/7
9	00 001 001	0	1*255/7	1*255/7
10	00 001 010	0	1*255/7	2*255/7
137	10 001 001	2*255/3	1*255/7	1*255/7
138	10 001 010	2*255/7	1*255/7	2*255/3
255	11 111 111	3*255/3	7*255/7	7*255/7

Building the Palette

The 3-3-2 palette is actually specified by `PIXELFORMATDESCRIPTOR` returned by `DescribePixelFormat`. The members `cRedBits`, `cGreenBits`, and `cBlueBits` specify 3, 3, and 2, respectively, for the number of bits that can represent each component. Furthermore, the `cRedShift`, `cGreenShift`, and `cBlueShift` values specify how much to shift the respective component value to the left (in this case, 0, 3, and 6 for red, green, and blue shifts). These sets of values compose the palette index (see [Figure 13.6](#)).

The code in [Listing 13.4](#) creates a palette if needed and returns its handle. This function makes use of the component bit counts and shift information in `PIXELFORMATDESCRIPTOR` to accommodate any subsequent palette requirements, such as a 2-2-2 palette.

Listing 13.4. Function to Create a Palette for OpenGL

```
// If necessary, creates a 3-3-2 palette for the device context listed.
HPALETTE GetOpenGLPalette(HDC hDC)
{
    HPALETTE hRetPal = NULL;           // Handle to palette to be created
    PIXELFORMATDESCRIPTOR pfd;          // Pixel format descriptor
    LOGPALETTE *pPal;                 // Pointer to memory for logical palette
    int nPixelFormat;                  // Pixel format index
    int nColors;                      // Number of entries in palette
    int i;                            // Counting variable
    BYTE RedRange,GreenRange,BlueRange;
                                // Range for each color entry (7,7, and 3)
    // Get the pixel format index and retrieve the pixel format description
    nPixelFormat = GetPixelFormat(hDC);
    DescribePixelFormat(hDC, nPixelFormat,
                        sizeof(PIXELFORMATDESCRIPTOR), &pfd);
    // Does this pixel format require a palette?  If not, do not create a
    // palette and just return NULL
    if(!(pfd.dwFlags & PFD_NEED_PALETTE))
        return NULL;
    // Number of entries in palette. 8 bits yields 256 entries
    nColors = 1 << pfd.cColorBits;
```

```

// Allocate space for a logical palette structure plus all the palette
// entries
pPal = (LOGPALETTE*)malloc(sizeof(LOGPALETTE) +
                           nColors*sizeof(PALETTEENTRY));
// Fill in palette header
pPal->palVersion = 0x300;           // Windows 3.0
pPal->palNumEntries = nColors;      // table size
// Build mask of all 1s. This creates a number represented by having
// the low order x bits set, where x = pfd.cRedBits, pfd.cGreenBits, and
// pfd.cBlueBits.
RedRange = (1 << pfd.cRedBits) -1;      // 7 for 3-3-2 palettes
GreenRange = (1 << pfd.cGreenBits) - 1;  // 7 for 3-3-2 palettes
BlueRange = (1 << pfd.cBlueBits) -1;      // 3 for 3-3-2 palettes
// Loop through all the palette entries
for(i = 0; i < nColors; i++)
{
    // Fill in the 8-bit equivalents for each component
    pPal->palPalEntry[i].peRed = (i >> pfd.cRedShift) & RedRange;
    pPal->palPalEntry[i].peRed = (unsigned char)(
        (double)pPal->palPalEntry[i].peRed * 255.0 / RedRange);
    pPal->palPalEntry[i].peGreen = (i >> pfd.cGreenShift) & GreenRange;
    pPal->palPalEntry[i].peGreen = (unsigned char)(
        (double)pPal->palPalEntry[i].peGreen * 255.0 / GreenRange);
    pPal->palPalEntry[i].peBlue = (i >> pfd.cBlueShift) & BlueRange;
    pPal->palPalEntry[i].peBlue = (unsigned char)(
        (double)pPal->palPalEntry[i].peBlue * 255.0 / BlueRange);
    pPal->palPalEntry[i].peFlags = (unsigned char) NULL;
}
// Create the palette
hRetPal = CreatePalette(pPal);
// Go ahead and select and realize the palette for this device context
SelectPalette(hDC,hRetPal, FALSE);
RealizePalette(hDC);
// Free the memory used for the logical palette structure
free(pPal);
// Return the handle to the new palette
return hRetPal;
}

```

Palette Creation and Disposal

The Windows palette should be created and realized before the OpenGL rendering context is created or made current. The function in [Listing 13.4](#) requires only the device context after the pixel format has been set. It then returns a handle to a palette if one is needed. [Listing 13.5](#) shows the sequence of operations when the window is created and destroyed. This listing is similar to code presented previously for the creation and destruction of the rendering context, but now it also takes into account the possible existence of a palette.

Listing 13.5. Creating and Destroying a Palette

```

// Window creation, set up for OpenGL
case WM_CREATE:
    // Store the device context
    hDC = GetDC(hWnd);
    // Select the pixel format
    SetDCPixelFormat(hDC);
    // Create the palette if needed
    hPalette = GetOpenGLPalette(hDC);
    // Create the rendering context and make it current
    hRC = wglCreateContext(hDC);

```

```
wglMakeCurrent(hDC, hRC);
break;
// Window is being destroyed, clean up
case WM_DESTROY:
    // Deselect the current rendering context and delete it
    wglMakeCurrent(hDC, NULL);
    wglDeleteContext(hRC);
    // If a palette was created, destroy it here
    if(hPalette != NULL)
        DeleteObject(hPalette);
    // Tell the application to terminate after the window
    // is gone.
    PostQuitMessage(0);
break;
```

Some Restrictions Apply

Not all your 256 palette entries are actually mapped to the system palette. Windows reserves 20 entries for static system colors that include the standard 16 VGA/EGA colors. This protects the standard Windows components (title bars, buttons, and so on) from alteration whenever an application changes the system palette. When your application realizes its palette, these 20 colors are not overwritten. Fortunately, some of these colors already exist or are closely matched in the 3-3-2 palette. Those that don't are matched closely enough that you shouldn't be able to tell the difference for most purposes.

We need to add one last important note about paletted rendering with OpenGL. The methods presented here enable you to specify colors as full RGBA components, and only at the point of rasterization are they converted to the nearest available palette entry. OpenGL does have an older and obsolete rendering mode called *color index mode* in which you can specify colors as actual palette entry indexes. Color index mode does not support modern features such as texture mapping and is not hardware accelerated on the PC (or the Mac, or Linux...). For these reasons, color index mode should be considered dead and is not covered by this text.

OpenGL and Windows Fonts

One of the nicer features of Windows is its support for TrueType fonts. These fonts have been native to Windows since before Windows became a 32-bit operating system. TrueType fonts enhance text appearance because they are device independent and can be easily scaled while still keeping a smooth shape. TrueType fonts are vector fonts, not bitmap fonts. What this means is that the character definitions consist of a series of point and curve definitions. When a character is scaled, the overall shape and appearance remain smooth.

Textual output is a part of nearly any Windows application, and 3D applications are no exception. Microsoft provided support for TrueType fonts in OpenGL with two new wiggle functions. You can use the first, `wglUseFontOutlines`, to create 3D font models that can be used to create 3D text effects. The second, `wglUseFontBitmaps`, creates a series of font character bitmaps that can be used for 2D text output in a double-buffered OpenGL window.

3D Fonts and Text

The `wglUseFontOutlines` function takes a handle to a device context. It uses the TrueType font currently selected into that device context to create a set of display lists for that font. Each display list renders just one character from the font. Listing 13.6 shows the `SetupRC` function from the sample program TEXT3D, where you can see the entire process of creating a font, selecting it into the device context, creating the display lists, and finally, deleting the (Windows) font.

Listing 13.6. Creating a Set of 3D Characters

```

void SetupRC(HDC hDC)
{
    // Set up the font characteristics
    HFONT hFont;
    GLYPHMETRICSFLOAT agmf[128]; // Throw away
    LOGFONT logfont;
    logfont.lfHeight = -10;
    logfont.lfWidth = 0;
    logfont.lfEscapement = 0;
    logfont.lfOrientation = 0;
    logfont.lfWeight = FW_BOLD;
    logfont.lfItalic = FALSE;
    logfont.lfUnderline = FALSE;
    logfont.lfStrikeOut = FALSE;
    logfont.lfCharSet = ANSI_CHARSET;
    logfont.lfOutPrecision = OUT_DEFAULT_PRECIS;
    logfont.lfClipPrecision = CLIP_DEFAULT_PRECIS;
    logfont.lfQuality = DEFAULT_QUALITY;
    logfont.lfPitchAndFamily = DEFAULT_PITCH;
    strcpy(logfont.lfFaceName, "Arial");
    // Create the font and display list
    hFont = CreateFontIndirect(&logfont);
    SelectObject (hDC, hFont);
    // Create display lists for glyphs 0 through 128 with 0.1 extrusion
    // and default deviation. The display list numbering starts at 1000
    // (it could be any number).
    nFontList = glGenLists(128);
    wglUseFontOutlines(hDC, 0, 128, nFontList, 0.0f, 0.5f,
                       WGL_FONT_POLYGONS, agmf);
    DeleteObject(hFont);
    . . .
    . . .
}

```

The function call to `wglUseFontOutlines` is the key function call to create your 3D character set:

```
wglUseFontOutlines(hDC, 0, 128, nFontList, 0.0f, 0.5f,
                   WGL_FONT_POLYGONS, agmf);
```

The first parameter is the handle to the device context where the desired font has been selected. The next two parameters specify the range of characters (called *glyphs*) in the font to use. In this case, you use the 1st through 127th character. (The indexes are zero based.) The third parameter, `nFontList`, is the beginning of the range of display lists created previously. It is important to allocate your display list space before using either of the `WGL` font functions. The next parameter is the chordal deviation. Think of it as specifying how smooth you want the font to appear, with 0.0 being the most smooth.

The `0.5f` is the extrusion of the character set. The 3D characters are defined to lay in the *xy* plane. The extrusion determines how far along the *z*-axis the characters extend. `WGL_FONT_POLYGONS` tells OpenGL to create the characters out of triangles and quads so that they are solid. When this information is specified, normals are also calculated and supplied for each letter. Only one other value is valid for this parameter: `WGL_FONT_LINES`. It produces a wireframe version of the character set and does not generate normals.

The last argument is an array of type `GLYPHMETRICSFLOAT`, which is defined as follows:

```
typedef struct _GLYPHMETRICSFLOAT {
```

```

FLOAT      gmfBlackBoxX;      // Extent of character cell in x direction
FLOAT      gmfBlackBoxY;      // Extent of character cell in y direction
POINTFLOAT gmfptGlyphOrigin; // Origin of character cell
FLOAT      gmfCellIncX;       // Horizontal distance to origin of next cell
FLOAT      gmfCellIncY;       // Vertical distance to origin of next cell
}; GLYPHMETRICSFLOAT

```

Windows fills in this array according to the selected font's characteristics. These values can be useful when you want to determine the size of a string rendered with 3D characters.

Rendering 3D Text

When the display list for each character is called, it renders the character and advances the current position to the right (positive x direction) by the width of the character cell. This is like calling `glTranslate` after each character, with the translation in the positive x direction. You can use the `glCallLists` function in conjunction with `glListBase` to treat a character array (a string) as an array of offsets from the first display list in the font. A simple text output method is shown in Listing 13.7 . The output from the TEXT3D program appears in Figure 13.7 .

Listing 13.7. Rendering a 3D Text String

```

void RenderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // Blue 3D text
    glColor3ub(0, 0, 255);
    glPushMatrix();
    glListBase(nFontList);
    glCallLists (6, GL_UNSIGNED_BYTE, "OpenGL");
    glPopMatrix();
}

```

Figure 13.7. Sample 3D text in OpenGL.



2D Fonts and Text

The `wglUseFontBitmaps` function is similar to its 3D counterpart. This function does not extrude the bitmaps into 3D, however, but instead creates a set of bitmap images of the glyphs in the font. You output images to the screen using the bitmap functions discussed in Chapter 7 , "Imaging with OpenGL." Each character rendered advances the raster position to the right in a similar manner to the 3D text.

Listing 3.8 shows the code to set up the coordinate system for the window (`ChangeSize` function), create the bitmap font (`SetupRC` function), and finally render some text (`RenderScene` function). The output from the TEXT2D sample program is shown in Figure 13.8 .

Listing 3.8. Creating and Using a 2D Font

```
///////////
// Window has changed size. Reset to match window coordinates
void ChangeSize(GLsizei w, GLsizei h)
{
    GLfloat nRange = 100.0f;
    GLfloat fAspect;
    // Prevent a divide by zero
    if(h == 0)
        h = 1;
    fAspect = (GLfloat)w/(GLfloat)h;
    // Set Viewport to window dimensions
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0,400, 400, 0);
    // Viewing transformation
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

///////////
// Setup. Use a Windows font to create the bitmaps
void SetupRC(HDC hDC)
{
    // Setup the Font characteristics
    HFONT hFont;
    LOGFONT logfont;
    logfont.lfHeight = -20;
    logfont.lfWidth = 0;
    logfont.lfEscapement = 0;
    logfont.lfOrientation = 0;
    logfont.lfWeight = FW_BOLD;
    logfont.lfItalic = FALSE;
    logfont.lfUnderline = FALSE;
    logfont.lfStrikeOut = FALSE;
    logfont.lfCharSet = ANSI_CHARSET;
    logfont.lfOutPrecision = OUT_DEFAULT_PRECIS;
    logfont.lfClipPrecision = CLIP_DEFAULT_PRECIS;
    logfont.lfQuality = DEFAULT_QUALITY;
    logfont.lfPitchAndFamily = DEFAULT_PITCH;
    strcpy(logfont.lfFaceName, "Arial");
    // Create the font and display list
    hFont = CreateFontIndirect(&logfont);
    SelectObject (hDC, hFont);
    //Create display lists for glyphs 0 through 128
    nFontList = glGenLists(128);
    wglUseFontBitmaps(hDC, 0, 128, nFontList);
    DeleteObject(hFont); // Don't need original font anymore
    // Black Background
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f );
}

///////////
// Draw everything (just the text)
void RenderScene(void)
{
```

```

glClear(GL_COLOR_BUFFER_BIT);
// Blue 3D Text - Note color is set before the raster position
glColor3f(1.0f, 1.0f, 1.0f);
glRasterPos2i(0, 200);
glListBase(nFontList);
glCallLists (13, GL_UNSIGNED_BYTE, "OpenGL Rocks! ");
}

```

Figure 13.8. Output from the TEXT2D sample program.



Note that `wglUseFontBitmaps` is a much simpler function. It requires only the device context handle, the beginning and last characters, and the first display list name to be used:

```
wglUseFontBitmaps(hDC, 0, 128, nFontList);
```

Because bitmap fonts are created based on the actual font and map directly to pixels on the screen, the `lfHeight` member of the `LOGFONT` structure is used exactly in the same way it is for GDI font rasterization.

Full-Screen Rendering

With OpenGL becoming popular among PC game developers, a common question is "How do I do full-screen rendering with OpenGL?" The truth is, if you've read this chapter, you already know how to do full-screen rendering with OpenGL—it's just like rendering into any other window! The real question is "How do I create a window that takes up the entire screen and has no borders?" Once you do this, rendering into this window is no different from rendering into any other window in any other sample in this book.

Even though this issue isn't strictly related to OpenGL, it is of enough interest to a wide number of our readers that we give this topic some coverage here.

Creating a Frameless Window

The first task is to create a window that has no border or caption. This procedure is quite simple. Following is the window creation code from the GLRECT sample program. We've made one small change by making the window style `WS_POPUP` instead of `WS_OVERLAPPEDWINDOW`:

```

// Create the main application window
hWnd = CreateWindow(lpszAppName,
                     lpszAppName,
                     // OpenGL requires WS_CLIPCHILDREN and WS_CLIPSIBLINGS
                     WS_POPUP | WS_CLIPCHILDREN | WS_CLIPSIBLINGS,
                     // Window position and size
                     100, 100,
                     250, 250,
                     NULL,
                     NULL,
                     hInstance,
                     NULL);

```

The result of this change is shown in [Figure 13.9](#).

Figure 13.9. A window with no caption or border.



As you can see, without the proper style settings, the window has neither a caption nor a border of any kind. Don't forget to take into account that now the window no longer has a close button on it. The user will have to press Alt+F4 to close the window and exit the program. Most user-friendly programs watch for a keystroke such as the Esc key or Q to terminate the program.

Creating a Full-Screen Window

Creating a window the size of the screen is almost as trivial as creating a window with no caption or border. The parameters of the `CreateWindow` function allow you to specify where onscreen the upper-left corner of the window will be positioned and the width and height of the window. To create a full-screen window, you always use (0,0) as the upper-left corner. The only trick would be determining what size the desktop is so you know how wide and high to make the window. You can easily determine this information by using the Windows function `GetDeviceCaps`.

[Listing 13.9](#) shows the new `WinMain` function from GLRECT, which is now the new sample FSCREEN. To use `GetDeviceCaps`, you need a device context handle. Because you are in the process of creating the main window, you need to use the device context from the desktop window.

Listing 13.9. Creating a Full-Screen Window

```

// Entry point of all Windows programs
int APIENTRY WinMain( HINSTANCE     hInstance,
                      HINSTANCE     hPrevInstance,
                      LPSTR         lpCmdLine,
                      int          nCmdShow)
{
    MSG          msg;           // Windows message structure
    WNDCLASS    wc;            // Windows class structure
    HWND        hWnd;          // Storage for window handle
    HWND        hDesktopWnd;   // Storage for desktop window handle

```

```

HDC          hDesktopDC; // Storage for desktop window device context
int          nScreenX, nScreenY; // Screen Dimensions
// Register Window style
wc.style          = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
wc.lpfnWndProc    = (WNDPROC) WndProc;
wc.cbClsExtra     = 0;
wc.cbWndExtra     = 0;
wc.hInstance       = hInstance;
wc.hIcon           = NULL;
wc.hCursor          = LoadCursor(NULL, IDC_ARROW);
// No need for background brush for OpenGL window
wc.hbrBackground   = NULL;
wc.lpszMenuName    = NULL;
wc.lpszClassName    = lpszAppName;
// Register the window class
if(RegisterClass(&wc) == 0)
    return FALSE;
// Get the Window handle and Device context to the desktop
hDesktopWnd = GetDesktopWindow();
hDesktopDC = GetDC(hDesktopWnd);
// Get the screen size
nScreenX = GetDeviceCaps(hDesktopDC, HORZRES);
nScreenY = GetDeviceCaps(hDesktopDC, VERTRES);
// Release the desktop device context
ReleaseDC(hDesktopWnd, hDesktopDC);
// Create the main application window
hWnd = CreateWindow(lpszAppName,
                    lpszAppName,
                    // OpenGL requires WS_CLIPCHILDREN and WS_CLIPSIBLINGS
                    WS_POPUP | WS_CLIPCHILDREN | WS_CLIPSIBLINGS,
                    // Window position and size
                    0, 0,
                    nScreenX, nScreenY,
                    NULL,
                    NULL,
                    hInstance,
                    NULL);
// If window was not created, quit
if(hWnd == NULL)
    return FALSE;
// Display the window
ShowWindow(hWnd, SW_SHOW);
UpdateWindow(hWnd);
// Process application messages until the application closes
while( GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
return msg.wParam;
}

```

The key code here is the lines that get the desktop window handle and device context. The device context can then be used to obtain the screen's horizontal and vertical resolution:

```

hDesktopWnd = GetDesktopWindow();
hDesktopDC = GetDC(hDesktopWnd);
// Get the screen size
nScreenX = GetDeviceCaps(hDesktopDC, HORZRES);
nScreenY = GetDeviceCaps(hDesktopDC, VERTRES);
// Release the desktop device context

```

```
ReleaseDC(hDesktopWnd, hDesktopDC);
```

If your system has multiple monitors, you should note that the values returned here would be for the primary display device. You might also be tempted to force the window to be a topmost window (using the `WS_EX_TOPMOST` window style). However, doing so makes it possible for your window to lose focus but remain on top of other active windows. This may confuse the user when the program stops responding to keyboard strokes.

You may also want to take a look at the Win32 function `ChangeDisplaySettings` in your Windows SDK documentation. This function allows you to dynamically change the desktop size at runtime and restore it when your application terminates. This capability may be desirable if you want to have a full-screen window but at a lower or higher display resolution than the default. If you do change the desktop settings, you must not create the rendering window or set the pixelformat until after the desktop settings have changed. OpenGL rendering contexts created under one environment (desktop settings) are not likely to be valid in another.

Multithreaded Rendering

A powerful feature of the Win32 API is multithreading. The topic of threading is beyond the scope of a book on computer graphics. Basically, a thread is the unit of execution for an application. Most programs execute instructions sequentially from the start of the program until the program terminates. A thread of execution is the path through the machine code that the CPU traverses as it fetches and executes instructions. By creating multiple threads using the Win32 API, you can create multiple paths through your source code that are followed simultaneously.

Think of multithreading as being able to call two functions at the same time and then having them executed simultaneously. Of course, the CPU cannot actually execute two code paths simultaneously, so it switches between threads during normal program flow much the same way a multitasking operating system switches between tasks.

A program carefully designed for multithreaded execution can outperform a single-threaded application in many circumstances. On a single processor machine, one thread can service I/O requests, for example, while another handles the GUI. On a multiprocessor machine employing Symmetric Multi-Processing (SMP), more than one CPU can actually execute your program simultaneously. Note, however, that SMP processing is not supported by older versions of Windows (95/98/ME).

Multithreading requires careful planning and usually causes applications to run more slowly or inefficiently when used improperly on a single CPU system. In addition, if a program is not thoroughly tested, it might never fail on a single CPU machine but have new bugs manifest on a machine with multiple processors.

Some OpenGL implementations take advantage of a multiprocessor system. If, for example, the transformation and lighting units of the OpenGL pipeline are not hardware accelerated, a driver can create another thread so that these calculations are performed by one CPU while another CPU feeds the transformed data to the rasterizer.

You might think that using two threads to do your OpenGL rendering would speed up your rendering as well. You could perhaps have one thread draw the background objects in a scene while another thread draws the more dynamic elements. This configuration is almost always a bad idea. Although you can create two OpenGL rendering contexts for two different threads, most drivers fail if you try to render with both of them in the same window. Technically, this multithreading should be possible, and the Microsoft generic implementation will succeed if you try it, as might many hardware implementations. In the real world, the extra work you place on the driver with two contexts trying to share the same framebuffer will most likely outweigh any performance benefit you hope to gain from using multiple threads.

Multithreading can benefit your OpenGL rendering on a multiprocessor system or even on a single processor system in at least two ways. In the first scenario, you have two different windows, each

with its own rendering context and thread of execution. This case could still stress some drivers (some of the low-end game boards are stressed just by two applications using OpenGL simultaneously!), but many professional OpenGL implementations can handle it quite well.

The second example is if you are writing a game or a real-time simulation. You can have a worker thread perform physics calculations or artificial intelligence or handle player interaction while another thread does the OpenGL rendering. This scenario requires careful sharing of data between threads but can provide a substantial performance boost on a dual-processor machine, and even a single-processor machine can improve the responsiveness of your program. Although we've made the disclaimer that multithreaded programming is outside the scope of this book, we present for your use the sample program RTHREAD included on the CD for your examination, which creates and uses a rendering thread. This program also demonstrates the use of the OpenGL WGL extensions.

OpenGL and WGL Extensions

On the Windows platform, you do not have direct access to the OpenGL driver. All OpenGL function calls are routed through the `opengl32.dll` system file. Because this DLL understands only OpenGL 1.1 entrypoints (function names), you must have a mechanism to get a pointer to an OpenGL function supported directly by the driver. Fortunately, the Windows OpenGL implementation has a function named `wglGetProcAddress` that allows you to retrieve a pointer to an OpenGL function supported by the driver, but not necessarily natively supported by `opengl32.dll`:

```
PROC wglGetProcAddress(LPSTR lpszProc);
```

This function takes the name of an OpenGL function or extension and returns a function pointer that you can use to call that function directly. For this to work, you must know the function prototype for the function so you can create a pointer to it and subsequently call the function.

OpenGL extensions (and post-version 1.1 features) come in two flavors. Some are simply new constants and enumerants recognized by a vendor's hardware driver. Others require that you call new functions added to the API. The number of extensions is extensive, especially when you add in the newer OpenGL core functionality and vendor-specific extensions. Complete coverage of all OpenGL extensions would require an entire book in itself (if not an encyclopedia!). You can find a registry of extensions on the Internet and among the Web sites listed in Appendix A, "Further Reading."

Fortunately, the following two header files give you programmatic access to most OpenGL extensions:

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

These files can be found at the OpenGL extension registry Web site, but they are also maintained by most graphics card vendors (see their developer support Web sites), and the latest version as of this book's printing is included in the `\common` source code directory on the CD. The `wglext.h` header contains a number of extensions that are Windows specific, and the `glext.h` header contains both standard OpenGL extensions and many vendor-specific OpenGL extensions.

Simple Extensions

Because this book covers known OpenGL features up to version 2.0, you may have already discovered that many of the sample programs in this book use these extensions for Windows builds of the sample code found in previous chapters. For example, in Chapter 9, "Texture

Mapping: Beyond the Basics," we showed you how to add specular highlights to textured geometry using OpenGL's separate specular color with the following function call:

```
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR);
```

However, this capability is not present in OpenGL 1.1, and neither the `GL_LIGHT_MODEL_COLOR_CONTROL` or `GL_SEPARATE_SPECULAR_COLOR` constants are defined in the Windows version of `gl.h`. They are, however, found in `glext.h`, and this file is already included automatically in all the samples in this book via the `OpenGLSB.h` header file. The `glLightModeli` function, on the other hand, has been around since OpenGL 1.0. These kinds of simple extensions simply pass new tokens to existing entrypoints (functions) and require only that you have the constants defined and know that the extension or feature is supported by the hardware.

Even if the OpenGL version is still reported as 1.1, this capability may still be included in the driver. This feature was originally an extension that was later promoted to the OpenGL core functionality. You can check for this and other easy-to-access extensions (no function pointers needed) quickly by using the following `GltTools` function:

```
bool gltIsExtSupported(const char *szExtension);
```

In the case of separate specular color, you might just code something like this:

```
if(gltIsExtSupported(GL_EXT_separate_specular_color))
    RenderOnce();
else
    UseMultiPassTechnique();
```

Here, you call the `RenderOnce` function if the extension (or feature) is supported and the `UserMultiPassTechnique` function to render an alternate (drawn twice and blended together) and slower way to achieve the same effect.

Using New Entrypoints

A more complex extension example comes from the IMAGING sample program in Chapter 7. In this case, the optional imaging subset is not only missing from the Windows version of `gl.h`, but is optional in all subsequent versions of OpenGL as well. This is an example of the type of feature that either has to be there, or there is no point in continuing. Thus, you first check for the presence of the imaging subset by checking for its extension string:

```
// Check for imaging subset, must be done after window
// is created or there won't be an OpenGL context to query
if(gltIsExtSupported("GL_ARB_imaging") == 0)
{
    printf("Imaging subset not supported\r\n");
    return 0;
}
```

The function prototype `typedefs` for the functions used are found in `glext.h`, and you use them to create function pointers to each of the functions you want to call. On the Macintosh platform, the standard system headers already contain these functions:

```
#ifndef __APPLE__
// These typedefs are found in glext.h
PFNGLHISTOGRAMPROC           glHistogram = NULL;
```

```

PFNGLGETHISTOGRAMPROC      glGetHistogram = NULL;
PFNGLCOLORTABLEPROC        glColorTable = NULL;
PFNGLCONVOLUTIONFILTER2DPROC glConvolutionFilter2D = NULL;
#endif

```

Now you use the `glTools` function `gltGetExtensionPointer` to retrieve the function pointer to the function in question. This function is simply a portability wrapper for `wglGetProcAddress` on Windows and an admittedly more complex method on the Apple of getting the function pointers:

```

#ifndef __APPLE__
glHistogram = gltGetExtensionPointer("glHistogram");
glGetHistogram = gltGetExtensionPointer("glGetHistogram");
glColorTable = gltGetExtensionPointer("glColorTable");
glConvolutionFilter2D = gltGetExtensionPointer("glConvolutionFilter2D");
#endif

```

Then you simply use the extension as if it were a normally supported part of the API:

```

// Start collecting histogram data, 256 luminance values
glHistogram(GL_HISTOGRAM, 256, GL_LUMINANCE, GL_FALSE);
 glEnable(GL_HISTOGRAM);

```

WGL Extensions

Several Windows-specific WGL extensions are also available—for example, the swap interval extension introduced in Chapter 2. You access the WGL extensions' entrypoints in the same manner as the other extensions—using the `wglGetProcAddress` function. There is, however, an important exception. Typically, among the many WGL extensions, only two are advertised by using `glGetString(GL_EXTENSIONS)`. They are the previously mentioned swap interval extension and the `WGL_ARB_extensions_string` extension. This extension provides yet another entrypoint that is used exclusively to query for the WGL extensions. The ARB extensions string function is prototyped as follows:

```
const char *wglGetExtensionsStringARB(HDC hdc);
```

This function retrieves the list of WGL extensions in the same manner you previously would have used `glGetString`. Using the `wglext.h` header file, you can retrieve a pointer to this function like this:

```

PFNWGLGETEXTENSIONSSTRINGARBPROC *wglGetExtensionsStringARB;
wglGetExtensionsStringARB = (PFNWGLGETEXTENSIONSSTRINGARBPROC)
    wglGetProcAddress("wglGetExtensionsStringARB");

```

`glGetString` returns the `WGL_ARB_extensions_string` identifier, but often developers skip this check and simply look for the entrypoint, as shown in the preceding code fragment. This approach is generally safe with most OpenGL extensions, but you should realize that this is, strictly speaking, "coloring outside the lines." Some vendors export extensions on an "experimental" basis, and these extensions may not be officially supported, or the functions may not function properly if you skip the extension string check. Also, more than one extension may use the same function or functions. Testing only for function availability provides no information on the availability of the specific extension or extensions that are supported.

Extended Pixel Formats

Perhaps one of the most important WGL extensions available for Windows is the `WGL_ARB_pixel_format` extension. This extension provides a mechanism that allows you to check for and select pixelformat features that did not exist when `PIXELFORMATDESCRIPTOR` was first created. For example, if your driver supports multisampled rendering (for full-scene antialiasing, for example), there is no way to select a pixelformat with this support using the old `PIXELFORMATDESCRIPTOR` fields. If this extension is supported, the driver exports the following functions:

```
BOOL wglGetPixelFormatAttribivARB(HDC hdc, GLint iPixelFormat,
                                 GLint iLayerPlane, GLuint nAttributes,
                                 const GLint *piAttributes, GLint *piValues);
BOOL wglGetPixelFormatAttribfvARB(HDC hdc, GLint iPixelFormat,
                                 GLint iLayerPlane, GLuint nAttributes,
                                 const GLint *piAttributes, GLfloat *pfValues);
```

These two variations of the same function allow you to query a particular pixelformat index and retrieve an array containing the attribute data for that pixelformat. The first argument, `hdc`, is the device context of the window that the pixelformat will be used for, followed by the pixelformat index. The `iLayerPlane` argument specifies which layer plane to query (0 if your implementation does not support layer planes). Next, `nAttributes` specifies how many attributes are being queried for this pixelformat, and the array `piAttributes` contains the list of attribute names to be queried. The attributes that can be specified are listed in Table 13.3. The final argument is an array that will be filled with the corresponding pixelformat attributes.

`WGL_NUMBER_PIXEL_FORMATS_ARB`

The number of pixelformats for this device.

`WGL_DRAW_TO_WINDOW_ARB`

Nonzero if the pixelformat can be used with a window.

`WGL_DRAW_TO_BITMAP_ARB`

Nonzero if the pixelformat can be used with a memory Device Independent Bitmap (DIB).

`WGL_DEPTH_BITS_ARB`

The number of bits in the depth buffer.

`WGL_STENCIL_BITS_ARB`

The number of bits in the stencil buffer.

`WGL_ACCELERATION_ARB`

One of the values in Table 13.4 that specifies which, if any, hardware driver is used.

`WGL_NEED_PALETTE_ARB`

Nonzero if a palette is required.

`WGL_NEED_SYSTEM_PALETTE_ARB`

Nonzero if the hardware supports one palette only in 256-color mode.

WGL_SWAP_LAYER_BUFFERS_ARB

Nonzero if the hardware supports swapping layer planes.

WGL_SWAP_METHOD_ARB

The method by which the buffer swap is accomplished for double-buffered pixelformats. It is one of the values listed in Table 13.5 .

WGL_NUMBER_OVERLAYS_ARB

The number of overlay planes.

WGL_NUMBER_UNDERLAYS_ARB

The number of underlay planes.

WGL_TRANSPARENT_ARB

Nonzero if transparency is supported.

WGL_TRANSPARENT_RED_VALUE_ARB

Transparent red color.

WGL_TRANSPARENT_GREEN_VALUE_ARB

Transparent green color.

WGL_TRANSPARENT_BLUE_VALUE_ARB

Transparent blue color.

WGL_TRANSPARENT_ALPHA_VALUE_ARB

Transparent alpha color.

WGL_SHARE_DEPTH_ARB

Nonzero if layer planes share a depth buffer with the main plane.

WGL_SHARE_STENCIL_ARB

Nonzero if layer planes share a stencil buffer with the main plane.

WGL_SHARE_ACCUM_ARB

Nonzero if layer planes share an accumulation buffer with the main plane.

WGL_SUPPORT_GDI_ARB

Nonzero if GDI rendering is supported (front buffer only).

WGL_SUPPORT_OPENGL_ARB

Nonzero if OpenGL is supported.

WGL_DOUBLE_BUFFER_ARB

Nonzero if double buffered.

WGL_STEREO_ARB

Nonzero if left and right buffers are supported.

WGL_PIXEL_TYPE_ARB

WGL_TYPE_RGBA_ARB for RGBA color modes; **WGL_TYPE_COLORINDEX_ARB** for color index mode.

WGL_COLOR_BITS_ARB

Number of bit planes in the color buffer.

WGL_RED_BITS_ARB

Number of red bit planes in the color buffer.

WGL_RED_SHIFT_ARB

Shift count for red bit planes.

WGL_GREEN_BITS_ARB

Number of green bit planes in the color buffer.

WGL_GREEN_SHIFT_ARB

Shift count for green bit planes.

WGL_BLUE_BITS_ARB

Number of blue bit planes in the color buffer.

WGL_BLUE_SHIFT_ARB

Shift count for blue bit planes.

WGL_ALPHA_BITS_ARB

Number of alpha bit planes in the color buffer.

WGL_ALPHA_SHIFT_ARB

Shift count for alpha bit planes.

WGL_ACCUM_BITS_ARB

Number of bit planes in the accumulation buffer.

`WGL_ACCUM_RED_BITS_ARB`

Number of red bit planes in the accumulation buffer.

`WGL_ACCUM_GREEN_BITS_ARB`

Number of green bit planes in the accumulation buffer.

`WGL_ACCUM_BLUE_BITS_ARB`

Number of blue bit planes in the accumulation buffer.

`WGL_ACCUM_ALPHA_BITS_ARB`

Number of alpha bit planes in the accumulation buffer.

`WGL_AUX_BUFFERS_ARB`

The number of auxiliary buffers.

Table 13.3. Pixelformat Attributes

Constant	Description

`WGL_NO_ACCELERATION_ARB`

Software rendering, no acceleration

`WGL_GENERIC_ACCELERATION_ARB`

Acceleration via an MCD driver

`WGL_FULL_ACCELERATION_ARB`

Acceleration via an ICD driver

Table 13.4.
Acceleration Flags for
`WGL_ACCELERATION_ARB`

Constant	Description

WGL_SWAP_EXCHANGE_ARB

Swapping exchanges the front and back buffers.

WGL_SWAP_COPY_ARB

The back buffer is copied to the front buffer.

WGL_SWAP_UNDEFINED_ARB

The back buffer is copied to the front buffer, but the back buffer contents remain undefined after the buffer swap.

Table 13.5. Buffer**Swap Values for****WGL_SWAP_METHOD_ARB**

Constant	Description

If you want to call the `wglGetPixelFormatAttrib` function, however, just like any other extension, the OpenGL rendering context must be current. This means that you must first create a temporary window, set up a pixelformat using `PIXELFORMATDESCRIPTOR`, and then retrieve and use a function pointer to one of the `wglGetPixelFormatAttribARB` functions. A convenient place to do this might be the splash screen or perhaps an initial Options dialog box that is presented to the user. You should not, however, try to use the Windows desktop because your application does not own it!

The following simple example queries for a single attribute—the number of pixelformats supported—so that you know how many you may need to look at:

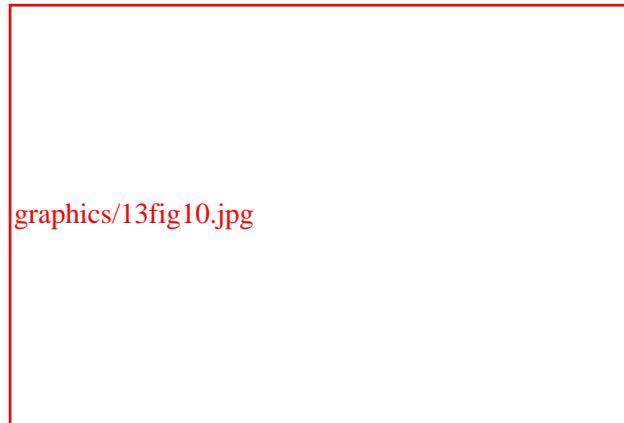
```
int attrib[] = { WGL_NUMBER_PIXEL_FORMATS_ARB };
int nResults[0];
wglGetPixelFormatAttributeivARB(hDC, 1, 0, 1, attrib, nResults);
// nResults[0] now contains the number of exported pixelformats
```

For a more detailed example showing how to look for a specific pixelformat (including a multisampled pixelformat), see the SPHEREWORLD32 sample program coming up next.

Win32 to the Max

SPHEREWORLD32 is a Win32-specific version of the Sphere World example we have returned to again and again throughout this book. SPHEREWORLD32 allows you to select windowed or full-screen mode, changes the display settings if necessary, and detects and allows you to select a multisampled pixelformat. Finally, you use the Windows-specific font features to display the frame rate and other information onscreen. When in full-screen mode, you can even Alt+Tab away from the program, and the window will be minimized until reselected.

The complete source to this "ultimate" Win32 sample program, provided in Listing 3.10, contains extensive comments to explain every aspect of the program. In the initial dialog box that is displayed (see Figure 3.10), you can select full-screen or windowed mode, multisampled rendering (if available), and whether you want to enable the swap interval extension. A sample screen of the running program is shown in Figure 3.11.

Figure 3.10. Initial Options dialog box for SPHEREWORLD32.**Figure 3.11. Output from the SPHEREWORLD32 sample program.****Listing 13.10. SPHEREWORLD32 Source Code**

```
// SphereWorld32.c
// OpenGL SuperBible
// Program by Richard S. Wright Jr.
// This program demonstrates a full featured robust Win32
// OpenGL framework
////////////////////////////////////////////////////////////////
// Include Files
#include <windows.h>           // Win32 Framework (No MFC)
#include <gl\gl.h>              // OpenGL
#include <gl\glu.h>             // GLU Library
#include <stdio.h>               // Standard IO (sprintf)
#include "..\..\common\wglext.h"  // WGL Extension Header
#include "..\..\common\glext.h"   // OpenGL Extension Header
#include "..\..\common\gltools.h" // GLTools library
#include "resource.h"            // Dialog resources
```

```

// Initial rendering options specified by the user.
struct STARTUPOPTIONS {
    DEVMODE     devMode;           // Display mode to use
    int         nPixelFormat;      // Pixel format to use
    int         nPixelFormatMS;    // Multisampled pixel format
    BOOL        bFullScreen;       // Full screen?
    BOOL        bFSAA;
    BOOL        bVerticalSync;
};

////////////////////////////////////////////////////////////////
// Module globals
static HPALETTE hPalette = NULL;           // Palette Handle
static HINSTANCE ghInstance = NULL;          // Module Instance Handle
static LPCTSTR lpszAppName = "SphereWorld32"; // Name of App
static GLint nFontList;                     // Base display list for font
static struct STARTUPOPTIONS startupOptions; // Startup options info
static LARGE_INTEGER CounterFrequency;
static LARGE_INTEGER FPSCount;
static LARGE_INTEGER CameraTimer;
#define NUM_SPHERES      30           // Number of Spheres
GLTFrame    spheres[NUM_SPHERES];          // Location of spheres
GLTFrame    frameCamera;                   // Location and orientation of camera
// Light and material Data
GLfloat fLightPos[4] = { -100.0f, 100.0f, 50.0f, 1.0f }; // Point source
GLfloat fNoLight[] = { 0.0f, 0.0f, 0.0f, 0.0f };
GLfloat fLowLight[] = { 0.25f, 0.25f, 0.25f, 1.0f };
GLfloat fBrightLight[] = { 1.0f, 1.0f, 1.0f, 1.0f };
// Shadow matrix
GLTMatrix mShadowMatrix;
// Textures identifiers
#define GROUND_TEXTURE 0
#define TORUS_TEXTURE 1
#define SPHERE_TEXTURE 2
#define NUM_TEXTURES 3
GLuint    textureObjects[NUM_TEXTURES];
const char *szTextureFiles[] = {"grass.tga", "wood.tga", "orb.tga"};
// Sphere and torus display lists
GLuint  lTorusList, lSphereList;
////////////////////////////////////////////////////////////////
// Forward Declarations
// Declaration for Window procedure
HRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                           WPARAM wParam, LPARAM lParam);

// Startup Dialog Procedure
BOOL APIENTRY StartupDlgProc (HWND hDlg, UINT message,
                               WPARAM wParam, LONG lParam);
// Find the best available pixelformat, including if Multisample is available
void FindBestPF(HDC hDC, int *nRegularFormat, int *nMSFormat);
BOOL ShowStartupOptions(void); // Initial startup dialog
void ChangeSize(GLsizei w, GLsizei h); // Change projection and viewport
void RenderScene(void); // Draw everything
void SetupRC(HDC hDC); // Set up the rendering context
void ShutdownRC(void); // Shutdown the rendering context
HPALETTE GetOpenGLPalette(HDC hDC); // Create a 3-3-2 palette
void DrawInhabitants(GLint nShadow); // Draw inhabitants of the world
void DrawGround(void); // Draw the ground
////////////////////////////////////////////////////////////////
// Extension function pointers
PFNWGLGETPIXELFORMATATTRIBIVARBPROC wglGetPixelFormatAttribivARB = NULL;
PFNGLWINDOWPOS2IPROC glWindowPos2i = NULL;
PFNWGLSWAPINTERVALEXTPROC wglSwapIntervalEXT = NULL;
////////////////////////////////////////////////////////////////

```

```

// Window has changed size. Reset to match window coordinates
void ChangeSize(GLsizei w, GLsizei h)
{
    GLfloat fAspect;
    // Prevent a divide by zero, when window is too short
    // (you can't make a window of zero width).
    if(h == 0)
        h = 1;
    glViewport(0, 0, w, h);
    fAspect = (GLfloat)w / (GLfloat)h;
    // Reset the coordinate system before modifying
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    // Set the clipping volume
    gluPerspective(35.0f, fAspect, 1.0f, 50.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
////////////////////////////////////////////////////////////////
// Draw the ground as a series of triangle strips
void DrawGround(void)
{
    GLfloat fExtent = 20.0f;
    GLfloat fStep = 1.0f;
    GLfloat y = -0.4f;
    GLint iStrip, iRun;
    GLfloat s = 0.0f;
    GLfloat t = 0.0f;
    GLfloat texStep = 1.0f / (fExtent * .075f);
    // Ground is a tiling texture
    glBindTexture(GL_TEXTURE_2D, textureObjects[GROUND_TEXTURE]);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    // Lay out strips and repeat textures coordinates
    for(iStrip = -fExtent; iStrip <= fExtent; iStrip += fStep)
    {
        t = 0.0f;
        glBegin(GL_TRIANGLE_STRIP);
        for(iRun = fExtent; iRun >= -fExtent; iRun -= fStep)
        {
            glTexCoord2f(s, t);
            glNormal3f(0.0f, 1.0f, 0.0f);    // All Point up
            glVertex3f(iStrip, y, iRun);
            glTexCoord2f(s + texStep, t);
            glNormal3f(0.0f, 1.0f, 0.0f);    // All Point up
            glVertex3f(iStrip + fStep, y, iRun);
            t += texStep;
        }
        glEnd();
        s += texStep;
    }
}
////////////////////////////////////////////////////////////////
// Draw random inhabitants and the rotating torus/sphere duo
void DrawInhabitants(GLint nShadow)
{
    static GLfloat yRot = 0.0f;           // Rotation angle for animation
    GLint i;
    if(nShadow == 0)
    {
        yRot += 0.5f;
        glColor4f(1.0f, 1.0f, 1.0f, 1.0f);

```

```

    }
else
    glColor4f(0.0f, 0.0f, .0f, .75f); // Shadow color
// Draw the randomly located spheres
glBindTexture(GL_TEXTURE_2D, textureObjects[SPHERE_TEXTURE]);
for(i = 0; i < NUM_SPHERES; i++)
{
    glPushMatrix();
    gltApplyActorTransform(&spheres[i]);
    glCallList(lSphereList);
    glPopMatrix();
}
glPushMatrix();
    glTranslatef(0.0f, 0.1f, -2.5f);
    glPushMatrix();
        glRotatef(-yRot * 2.0f, 0.0f, 1.0f, 0.0f);
        glTranslatef(1.0f, 0.0f, 0.0f);
        glCallList(lSphereList);
    glPopMatrix();
    if(nShadow == 0)
    {
        // Torus alone will be specular
        glMaterialfv(GL_FRONT, GL_SPECULAR, fBrightLight);
    }
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);
    glBindTexture(GL_TEXTURE_2D, textureObjects[TORUS_TEXTURE]);
    glCallList(lTorusList);
    glMaterialfv(GL_FRONT, GL_SPECULAR, fNoLight);
glPopMatrix();
}

///////////////////////////////
// Draw everything
void RenderScene(void)
{
    static int iFrames = 0; // Count frames to calculate fps every 100 frames
    static float fps = 0.0f; // Calculated fps
    // Clear the window
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
    glPushMatrix();
        gltApplyCameraTransform(&frameCamera); // Move camera/world
        // Position light before any other transformations
        glLightfv(GL_LIGHT0, GL_POSITION, fLightPos);
        // Draw the ground
        glColor3f(1.0f, 1.0f, 1.0f);
        DrawGround();
        // Draw shadows first
        glDisable(GL_DEPTH_TEST);
        glDisable(GL_LIGHTING);
        glDisable(GL_TEXTURE_2D);
        glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
        glEnable(GL_STENCIL_TEST);
    glPushMatrix();
        glMultMatrixf(mShadowMatrix);
        DrawInhabitants(1);
    glPopMatrix();
    glDisable(GL_STENCIL_TEST);
    glDisable(GL_BLEND);
    glEnable(GL_LIGHTING);
    glEnable(GL_TEXTURE_2D);
    glEnable(GL_DEPTH_TEST);
}

```

```

// Draw inhabitants normally
DrawInhabitants(0);
glPopMatrix();
// Calculate Frame Rate, once every 100 frames
iFrames++;
if(iFrames == 100)
{
    float fTime;
    // Get the current count
    LARGE_INTEGER lCurrent;
    QueryPerformanceCounter(&lCurrent);
    fTime = (float)(lCurrent.QuadPart - FPSCount.QuadPart) /
        (float)CounterFrequency.QuadPart;
    fps = (float)iFrames / fTime;
    // Reset frame count and timer
    iFrames = 0;
    QueryPerformanceCounter(&FPSCount);
}
// If we have the window position extension, display
// the frame rate, and tell if multisampling was enabled
// and if the VSync is turned on.
if(glWindowPos2i != NULL)
{
    int iRow = 10;
    char cBuffer[64];
    // Turn off depth test, lighting, and texture mapping
    glDisable(GL_DEPTH_TEST);
    glDisable(GL_LIGHTING);
    glDisable(GL_TEXTURE_2D);
    glColor3f(1.0f, 1.0f, 1.0f);
    // Set position and display message
    glWindowPos2i(0, iRow);
    glListBase(nFontList);
    glCallLists (13, GL_UNSIGNED_BYTE, "OpenGL Rocks!");
    iRow+= 20;
    // Display the frame rate
    sprintf(cBuffer,"FPS: %.1f", fps);
    glWindowPos2i(0, iRow);
    glCallLists(strlen(cBuffer), GL_UNSIGNED_BYTE, cBuffer);
    iRow += 20;
    // MultiSampled?
    if(startupOptions.bFSAA == TRUE && startupOptions.nPixelFormatMS != 0)
    {
        glWindowPos2i(0, iRow);
        glCallLists(25 ,GL_UNSIGNED_BYTE,"Multisampled Frame Buffer");
        iRow += 20;
    }
    // VSync?
    if(wglSwapIntervalEXT != NULL && startupOptions.bVerticalSync == TRUE)
    {
        glWindowPos2i(0, iRow);
        glCallLists(9 ,GL_UNSIGNED_BYTE, "VSync On");
        iRow += 20;
    }
    // Put everything back
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_LIGHTING);
    glEnable(GL_TEXTURE_2D);
}
}
////////////////////////////////////////////////////////////////
// Setup. Create font/bitmaps, load textures, create display lists

```

```

void SetupRC(HDC hDC)
{
    GLTVector3 vPoints[3] = {{ 0.0f, -0.4f, 0.0f },
                            { 10.0f, -0.4f, 0.0f },
                            { 5.0f, -0.4f, -5.0f }};

    int iSphere;
    int i;
    // Setup the Font characteristics
    HFONT hFont;
    LOGFONT logfont;
    logfont.lfHeight = -20;
    logfont.lfWidth = 0;
    logfont.lfEscapement = 0;
    logfont.lfOrientation = 0;
    logfont.lfWeight = FW_BOLD;
    logfont.lfItalic = FALSE;
    logfont.lfUnderline = FALSE;
    logfont.lfStrikeOut = FALSE;
    logfont.lfCharSet = ANSI_CHARSET;
    logfont.lfOutPrecision = OUT_DEFAULT_PRECIS;
    logfont.lfClipPrecision = CLIP_DEFAULT_PRECIS;
    logfont.lfQuality = DEFAULT_QUALITY;
    logfont.lfPitchAndFamily = DEFAULT_PITCH;
    strcpy(logfont.lfFaceName, "Arial");
    // Create the font and display list
    hFont = CreateFontIndirect(&logfont);
    SelectObject (hDC, hFont);
    //Create display lists for glyphs 0 through 128
    nFontList = glGenLists(128);
    wglUseFontBitmaps(hDC, 0, 128, nFontList);
    DeleteObject(hFont);           // Don't need original font anymore
    // Grayish background
    glClearColor(fLowLight[0], fLowLight[1], fLowLight[2], fLowLight[3]);
    // Clear stencil buffer with zero, increment by one whenever anybody
    // draws into it. When stencil function is enabled, only write where
    // stencil value is zero. This prevents the transparent shadow from drawing
    // over itself
    glStencilOp(GL_INCR, GL_INCR, GL_INCR);
    glClearStencil(0);
    glStencilFunc(GL_EQUAL, 0x0, 0x01);

    // Cull backs of polygons
    glCullFace(GL_BACK);
    glFrontFace(GL_CCW);
    glEnable(GL_CULL_FACE);
    glEnable(GL_DEPTH_TEST);
    // Setup light parameters
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, fNoLight);
    glLightfv(GL_LIGHT0, GL_AMBIENT, fLowLight);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, fBrightLight);
    glLightfv(GL_LIGHT0, GL_SPECULAR, fBrightLight);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    // Calculate shadow matrix
    gltMakeShadowMatrix(vPoints, fLightPos, mShadowMatrix);
    // Mostly use material tracking
    glEnable(GL_COLOR_MATERIAL);
    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
    glMateriali(GL_FRONT, GL_SHININESS, 128);
    gltInitFrame(&frameCamera); // Initialize the camera
    // Randomly place the sphere inhabitants
    for(iSphere = 0; iSphere < NUM_SPHERES; iSphere++)

```

```

{
    gltInitFrame(&spheres[iSphere]);      // Initialize the frame
    // Pick a random location between -20 and 20 at .1 increments
    spheres[iSphere].vLocation[0] = (float)((rand() % 400) - 200) * 0.1f;
    spheres[iSphere].vLocation[1] = 0.0f;
    spheres[iSphere].vLocation[2] = (float)((rand() % 400) - 200) * 0.1f;
}
// Set up texture maps
glEnable(GL_TEXTURE_2D);
glGenTextures(NUM_TEXTURES, textureObjects);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
// Load each texture
for(i = 0; i < NUM_TEXTURES; i++)
{
    GLubyte *pBytes;
    GLint iWidth, iHeight, iComponents;
    GLenum eFormat;
    glBindTexture(GL_TEXTURE_2D, textureObjects[i]);
    // Load this texture map
    pBytes = gltLoadTGA(szTextureFiles[i], &iWidth, &iHeight,
                        &iComponents, &eFormat);
    gluBuild2DMipmaps(GL_TEXTURE_2D, iComponents, iWidth,
                      iHeight, eFormat, GL_UNSIGNED_BYTE, pBytes);
    free(pBytes);
    // Trilinear mipmapping
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_LINEAR_MIPMAP_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
}
// Get window position function pointer if it exists
glWindowPos2i = (PFNGLWINDOWPOS2IPROC)wglGetProcAddress("glWindowPos2i");
// Get swap interval function pointer if it exists
wglSwapIntervalEXT = (PFNGLSWAPINTERVALEXTPROC)
                     wglGetProcAddress("wglSwapIntervalEXT");
if(wglSwapIntervalEXT != NULL && startupOptions.bVerticalSync == TRUE)
    wglSwapIntervalEXT(1);
// If multisampling was available and was selected, enable
if(startupOptions.bFSAA == TRUE && startupOptions.nPixelFormatMS != 0)
    glEnable(GL_MULTISAMPLE_ARB);
// If separate specular color is available, make torus shiny
if(gltIsExtSupported("GL_EXT_separate_specular_color"))
    glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR);

// Initialize the timers
QueryPerformanceFrequency(&CounterFrequency);
QueryPerformanceCounter(&FPSCount);
CameraTimer = FPSCount;
// Build display lists for the torus and spheres
// (You could do one for the ground as well)
lTorusList = glGenLists(2);
lSphereList = lTorusList + 1;
glNewList(lTorusList, GL_COMPILE);
    gltDrawTorus(0.35f, 0.15f, 61, 37);
glEndList();
glNewList(lSphereList, GL_COMPILE);
    gltDrawSphere(0.3f, 31, 16);
glEndList();
}
///////////////////////////////
// Shutdown the rendering context

```

```

void ShutdownRC(void)
{
    glDeleteLists(nFontList, 128); // Delete font display list
    glDeleteLists(lTorusList, 2); // Delete object display lists
    glDeleteTextures(NUM_TEXTURES, textureObjects); // Release textures
}
////////////////////////////////////////////////////////////////
// If necessary, creates a 3-3-2 palette for the device context listed.
HPALETTE GetOpenGLPalette(HDC hDC)
{
    HPALETTE hRetPal = NULL; // Handle to palette to be created
    PIXELFORMATDESCRIPTOR pfd; // Pixel Format Descriptor
    LOGPALETTE *pPal; // Pointer to memory for logical palette
    int nPixelFormat; // Pixel format index
    int nColors; // Number of entries in palette
    int i; // Counting variable
    BYTE RedRange,GreenRange,BlueRange;
                                // Range for each color entry (7,7, and 3)
    // Get the pixel format index and retrieve the pixel format description
    nPixelFormat = GetPixelFormat(hDC);
    DescribePixelFormat(hDC, nPixelFormat, sizeof(PIXELFORMATDESCRIPTOR),&pfd);
    // Does this pixel format require a palette? If not, do not create a
    // palette and just return NULL
    if(!(pfd.dwFlags & PFD_NEED_PALETTE))
        return NULL;
    // Number of entries in palette. 8 bits yields 256 entries
    nColors = 1 << pfd.cColorBits;
    // Allocate space for a logical palette structure plus all palette entries
    pPal = (LOGPALETTE*)malloc(sizeof(LOGPALETTE)+nColors*sizeof(PALETTEENTRY));
    // Fill in palette header
    pPal->palVersion = 0x300; // Windows 3.0
    pPal->palNumEntries = nColors; // table size
    // Build mask of all 1's. This creates a number represented by having
    // the low order x bits set, where x = pfd.cRedBits, pfd.cGreenBits, and
    // pfd.cBlueBits.
    RedRange = (1 << pfd.cRedBits) -1;
    GreenRange = (1 << pfd.cGreenBits) - 1;
    BlueRange = (1 << pfd.cBlueBits) -1;
    // Loop through all the palette entries
    for(i = 0; i < nColors; i++)
    {
        // Fill in the 8-bit equivalents for each component
        pPal->palPalEntry[i].peRed = (i >> pfd.cRedShift) & RedRange;
        pPal->palPalEntry[i].peRed = (unsigned char)(
            (double) pPal->palPalEntry[i].peRed * 255.0 / RedRange);

        pPal->palPalEntry[i].peGreen = (i >> pfd.cGreenShift) & GreenRange;
        pPal->palPalEntry[i].peGreen = (unsigned char)(
            (double)pPal->palPalEntry[i].peGreen * 255.0 / GreenRange);
        pPal->palPalEntry[i].peBlue = (i >> pfd.cBlueShift) & BlueRange;
        pPal->palPalEntry[i].peBlue = (unsigned char)(
            (double)pPal->palPalEntry[i].peBlue * 255.0 / BlueRange);
        pPal->palPalEntry[i].peFlags = (unsigned char) NULL;
    }
    // Create the palette
    hRetPal = CreatePalette(pPal);
    // Go ahead and select and realize the palette for this device context
    SelectPalette(hDC,hRetPal, FALSE);
    RealizePalette(hDC);
    // Free the memory used for the logical palette structure
    free(pPal);
    // Return the handle to the new palette
}

```

```

return hRetVal;
}
///////////////////////////////////////////////////////////////////
// Entry point of all Windows programs
int APIENTRY WinMain(      HINSTANCE      hInstance,
                           HINSTANCE      hPrevInstance,
                           LPSTR         lpCmdLine,
                           int           nCmdShow)
{
MSG         msg;           // Windows message structure
WNDCLASS    wc;            // Windows class structure
HWND        hWnd;          // Storage for window handle
UINT        uiStyle,uiStyleX;
ghInstance = hInstance;    // Save instance handle
// Get startup options, or shutdown
if(ShowStartupOptions() == FALSE)
    return 0;
if(startupOptions.bFullScreen == TRUE)
    if(ChangeDisplaySettings(&startupOptions.devMode, CDS_FULLSCREEN)
        != DISP_CHANGE_SUCCESSFUL)
    {
// Replace with string resource, and actual width and height
MessageBox(NULL, TEXT("Cannot change to selected desktop resolution."),
           NULL, MB_OK | MB_ICONSTOP);
    return -1;
    }
// Register Window style
wc.style          = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
wc.lpfnWndProc   = (WNDPROC) WndProc;
wc.cbClsExtra    = 0;
wc.cbWndExtra    = 0;
wc.hInstance     = hInstance;
wc.hIcon         = NULL;
wc.hCursor        = LoadCursor(NULL, IDC_ARROW);
// No need for background brush for OpenGL window
wc.hbrBackground = NULL;
wc.lpszMenuName  = NULL;
wc.lpszClassName = lpszAppName;
// Register the window class
if(RegisterClass(&wc) == 0)
return FALSE;
// Select window styles
if(startupOptions.bFullScreen == TRUE)
{
    uiStyle = WS_POPUP;
    uiStyleX = WS_EX_TOPMOST;
}
else
{
    uiStyle = WS_OVERLAPPEDWINDOW;
    uiStyleX = 0;
}
// Create the main 3D window
hWnd = CreateWindowEx(uiStyleX, wc.lpszClassName, lpszAppName, uiStyle,
                      0, 0, startupOptions.devMode.dmPelsWidth,
                      startupOptions.devMode.dmPelsHeight, NULL, NULL, hInstance, NULL);
// If window was not created, quit
if(hWnd == NULL)
    return FALSE;
// Make sure window manager stays hidden
ShowWindow(hWnd, SW_SHOW);
UpdateWindow(hWnd);

```

```

// Process application messages until the application closes
while( GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
// Restore Display Settings
if(startupOptions.bFullScreen == TRUE)
    ChangeDisplaySettings(NULL, 0);
return msg.wParam;
}
///////////
// Window procedure, handles all messages for this program
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static HGLRC hRC;           // Permanent Rendering context
    static HDC hDC;             // Private GDI Device context
    switch (message)
    {
        // Window creation, setup for OpenGL
        case WM_CREATE:
            // Store the device context
            hDC = GetDC(hWnd);
            // The screen and desktop may have changed, so do this again
            FindBestPF(hDC, &startupOptions.nPixelFormat,
                        &startupOptions.nPixelFormatMS);

            // Set pixelformat
            if(startupOptions.bFSAA == TRUE &&
               (startupOptions.nPixelFormatMS != 0))
                SetPixelFormat(hDC, startupOptions.nPixelFormatMS, NULL);
            else
                SetPixelFormat(hDC, startupOptions.nPixelFormat, NULL);
            // Create the rendering context and make it current
            hRC = wglCreateContext(hDC);
            wglMakeCurrent(hDC, hRC);
            // Create the palette
            hPalette = GetOpenGLPalette(hDC);
            SetupRC(hDC);
            break;
        // Check for ESC key
        case WM_CHAR:
            if(wParam == 27)
                DestroyWindow(hWnd);
            break;
        // Window is either full screen, or not visible
        case WM_ACTIVATE:
        {
            // Ignore this altogether unless we are in full screen mode
            if(startupOptions.bFullScreen == TRUE)
            {
                // Construct windowplacement structure
                WINDOWPLACEMENT wndPlacement;
                wndPlacement.length = sizeof(WINDOWPLACEMENT);
                wndPlacement.flags = WPF_RESTORETOMAXIMIZED;
                wndPlacement.ptMaxPosition.x = 0;
                wndPlacement.ptMaxPosition.y = 0;
                wndPlacement.ptMinPosition.x = 0;
                wndPlacement.ptMinPosition.y = 0;
                wndPlacement.rcNormalPosition.bottom =
                                startupOptions.devMode.dmPelsHeight;
                wndPlacement.rcNormalPosition.left = 0;
                wndPlacement.rcNormalPosition.top = 0;
            }
        }
    }
}

```

```

wndPlacement.rcNormalPosition.right =
    startupOptions.devMode.dmPelsWidth;
// Switching away from window
if(LOWORD(wParam) == WA_INACTIVE)
{
    wndPlacement.showCmd = SW_SHOWMINNOACTIVE;
    SetWindowPlacement(hWnd, &wndPlacement);
    ShowCursor(TRUE);
}
else // Switching back to window
{
    wndPlacement.showCmd = SW_RESTORE;
    SetWindowPlacement(hWnd, &wndPlacement);
    ShowCursor(FALSE);
}
}
break;
// Window is being destroyed, cleanup
case WM_DESTROY:
    ShutdownRC();
    // Deselect the current rendering context and delete it
    wglGetCurrentContext(hDC, NULL);
    wglDeleteContext(hRC);
    // Delete the palette
    if(hPalette != NULL)
        DeleteObject(hPalette);
    // Tell the application to terminate after the window
    // is gone.
    PostQuitMessage(0);
break;
// Window is resized.
case WM_SIZE:
    // Call our function which modifies the clipping
    // volume and viewport
    ChangeSize(LOWORD(lParam), HIWORD(lParam));
break;
// The painting function. This message sent by Windows
// whenever the screen needs updating.
case WM_PAINT:
{
    // Only poll keyboard when this window has focus
    if(GetFocus() == hWnd)
    {
        float fTime;
        float fLinear, fAngular;
        // Get the time since the last time we rendered a frame
        LARGE_INTEGER lCurrent;
        QueryPerformanceCounter(&lCurrent);
        fTime = (float)(lCurrent.QuadPart - CameraTimer.QuadPart) /
            (float)CounterFrequency.QuadPart;
        CameraTimer = lCurrent;
        // Camera motion will be time based. This keeps the motion constant
        // regardless of frame rate. Higher frame rates produce smoother
        // animation and motion, they should not produce "faster" motion.
        fLinear = fTime * 1.0f;
        fAngular = (float)gltDegToRad(60.0f * fTime);
        // Move the camera around, poll the keyboard
        if(GetAsyncKeyState(VK_UP))
            gltMoveFrameForward(&frameCamera, fLinear);
        if(GetAsyncKeyState(VK_DOWN))
            gltMoveFrameForward(&frameCamera, -fLinear);
}
}

```

```

        if(GetAsyncKeyState(VK_LEFT))
            gltRotateFrameLocalY(&frameCamera, fAngular);
        if(GetAsyncKeyState(VK_RIGHT))
            gltRotateFrameLocalY(&frameCamera, -fAngular);
    }
    // Call OpenGL drawing code
    RenderScene();
    // Call function to swap the buffers
    SwapBuffers(hDC);
    // Not validated on purpose, gives an endless series
    // of paint messages... this is akin to having
    // a rendering loop
    //ValidateRect(hWnd,NULL);
}
break;
// Windows is telling the application that it may modify
// the system palette. This message in essence asks the
// application for a new palette.
case WM_QUERYNEWPALETTE:
    // If the palette was created.
    if(hPalette)
    {
        int nRet;
        // Selects the palette into the current device context
        SelectPalette(hDC, hPalette, FALSE);
        // Map entries from the currently selected palette to
        // the system palette. The return value is the number
        // of palette entries modified.
        nRet = RealizePalette(hDC);
        // Repaint, forces remap of palette in current window
        InvalidateRect(hWnd,NULL,FALSE);
        return nRet;
    }
break;
// This window may set the palette, even though it is not the
// currently active window.
case WM_PALETTECHANGED:
    // Don't do anything if the palette does not exist, or if
    // this is the window that changed the palette.
    if((hPalette != NULL) && ((HWND)wParam != hWnd))
    {
        // Select the palette into the device context
        SelectPalette(hDC,hPalette,FALSE);
        // Map entries to system palette
        RealizePalette(hDC);
        // Remap the current colors to the newly realized palette
        UpdateColors(hDC);
        return 0;
    }
break;
default: // Passes it on if unprocessed
    return (DefWindowProc(hWnd, message, wParam, lParam));
}

return (0L);
}
////////////////////////////////////////////////////////////////
// Dialog procedure for the startup dialog
BOOL APIENTRY StartupDlgProc (HWND hDlg, UINT message, UINT wParam, LONG lParam)
{
switch (message)
{

```

```

// Initialize the dialog box
case WM_INITDIALOG:
{
    int nPF;
    HDC hDC;                                // Dialogs device context
    HGLRC hRC;
    DEVMODE devMode;
    unsigned int iMode;
    unsigned int nWidth;      // Current settings
    unsigned int nHeight;
    char cBuffer[64];
    HWND hListBox;
    PIXELFORMATDESCRIPTOR pfd = {    // Not going to be too picky
        sizeof(PIXELFORMATDESCRIPTOR),
        1,
        PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER,
        PFD_TYPE_RGBA,                      // Full color
        32,                                // Color depth
        0,0,0,0,0,0,0,                      // Ignored
        0,0,0,0,0,0,0,                      // Accumulation buffer
        16,                                // Depth bits
        8,                                 // Stencil bits
        0,0,0,0,0,0,0 };                  // Some used, some not
    // Initialize render options
    startupOptions.bFSAA = FALSE;
    startupOptions.bFullScreen = FALSE;
    startupOptions.bVerticalSync = FALSE;
    // Create a "temporary" OpenGL rendering context
    hDC = GetDC(hDlg);
    // Set pixel format one time....
    nPF = ChoosePixelFormat(hDC, &pfd);
    SetPixelFormat(hDC, nPF, &pfd);
    DescribePixelFormat(hDC, nPF, sizeof(PIXELFORMATDESCRIPTOR), &pfd);
    // Create the GL context
    hRC = wglCreateContext(hDC);
    wglMakeCurrent(hDC, hRC);
    // Set text in dialog
    SetDlgItemText(hDlg, IDC_VENDOR,
                    (const char *)glGetString(GL_VENDOR));
    SetDlgItemText(hDlg, IDC_RENDERER,
                    (const char *)glGetString(GL_RENDERER));
    SetDlgItemText(hDlg, IDC_VERSION,
                    (const char *)glGetString(GL_VERSION));
    // Vertical Sync off by default
    if(gltIsExtSupported("WGL_EXT_swap_control"))
        EnableWindow(GetDlgItem(hDlg, IDC_VSYNC_CHECK), TRUE);
    // Find a multisampled and non-multisampled pixel format
    FindBestPF(hDC, &startupOptions.nPixelFormat,
               &startupOptions.nPixelFormatMS);
    // Done with GL context
    wglMakeCurrent(hDC, NULL);
    wglDeleteContext(hRC);
    // Enumerate display modes
    iMode = 0;
    nWidth = GetSystemMetrics(SM_CXSCREEN);    // Current settings
    nHeight = GetSystemMetrics(SM_CYSCREEN);
    hListBox = GetDlgItem(hDlg, IDC_DISPLAY_COMBO);
    while(EnumDisplaySettings(NULL, iMode, &devMode))
    {
        int iItem;
        sprintf(cBuffer, "%d x %d x %dbpp @%dhz", devMode.dmPelsWidth,
                devMode.dmPelsHeight, devMode.dmBitsPerPel,

```

```

        devMode.dmDisplayFrequency);
iItem = SendMessage(hListBox, CB_ADDSTRING, 0, (LPARAM)cBuffer);
SendMessage(hListBox, CB_SETITEMDATA, iItem, iMode);
if(devMode.dmPelsHeight == nHeight &&
   devMode.dmPelsWidth == nWidth)
    SendMessage(hListBox, CB_SETCURSEL, iItem, 0);
iMode++;
}
// Set other defaults /////////////
// Windowed or full screen
CheckDlgButton(hDlg, IDC_FS_CHECK, BST_CHECKED);
// FSAA, but only if support detected
if(startupOptions.nPixelFormatMS != 0)
    EnableWindow(GetDlgItem(hDlg, IDC_MULTISAMPLED_CHECK), TRUE);
return (TRUE);
}
break;
// Process command messages
case WM_COMMAND:
{
// Validate and Make the changes
if(LOWORD(wParam) == IDOK)
{
// Read options /////////////////////////////////
// Display mode
HWND hListBox = GetDlgItem(hDlg, IDC_DISPLAY_COMBO);
int iMode = SendMessage(hListBox, CB_GETCURSEL, 0, 0);
iMode = SendMessage(hListBox, CB_GETITEMDATA, iMode, 0);
EnumDisplaySettings(NULL, iMode, &startupOptions.devMode);
// Full screen or windowed?
if(IsDlgButtonChecked(hDlg, IDC_FS_CHECK))
    startupOptions.bFullScreen = TRUE;
else
    startupOptions.bFullScreen = FALSE;
// FSAA
if(IsDlgButtonChecked(hDlg, IDC_MULTISAMPLED_CHECK))
    startupOptions.bFSAA = TRUE;
else
    startupOptions.bFSAA = FALSE;
// Vertical Sync.
if(IsDlgButtonChecked(hDlg, IDC_VSYNC_CHECK))
    startupOptions.bVerticalSync = TRUE;
else
    startupOptions.bVerticalSync = FALSE;
EndDialog(hDlg,TRUE);
}
if(LOWORD(wParam) == IDCANCEL)
    EndDialog(hDlg, FALSE);
}
break;
// Closed from sysbox
case WM_CLOSE:
    EndDialog(hDlg, FALSE); // Same as cancel
break;
}
return FALSE;
}
///////////////////////////////
// Display the startup screen (just a modal dialog box)
BOOL ShowStartupOptions(void)
{
return DialogBox (ghInstance,

```

```

MAKEINTRESOURCE(IDD_DLG_INTRO),
NULL,
StartupDlgProc);
}

// Select pixelformat with desired attributes
// Returns the best available "regular" pixel format, and the best available
// Multisampled pixelformat (0 if not available)
void FindBestPF(HDC hDC, int *nRegularFormat, int *nMSFormat)
{
    *nRegularFormat = 0;
    *nMSFormat = 0;
    // easy check, just look for the entrypoint
    if(gltIsWGLExtSupported(hDC, "WGL_ARB_pixel_format"))
        if(wglGetPixelFormatAttribivARB == NULL)
            wglGetPixelFormatAttribivARB = (PFNWGLGETPIXELFORMATATTRIBIVARBPROC)
                wglGetProcAddress("wglGetPixelFormatAttribivARB");

    // First try to use new extended wgl way
    if(wglGetPixelFormatAttribivARB != NULL)
    {
        // Only care about these attributes
        int nBestMS = 0;
        int i;
        int iResults[9];
        int iAttributes [9] = { WGL_SUPPORT_OPENGL_ARB, // 0
                               WGL_ACCELERATION_ARB, // 1
                               WGL_DRAW_TO_WINDOW_ARB, // 2
                               WGL_DOUBLE_BUFFER_ARB, // 3
                               WGL_PIXEL_TYPE_ARB, // 4
                               WGL_DEPTH_BITS_ARB, // 5
                               WGL_STENCIL_BITS_ARB, // 6
                               WGL_SAMPLE_BUFFERS_ARB, // 7
                               WGL_SAMPLES_ARB }; // 8

        // How many pixelformats are there?
        int nFormatCount[] = { 0 };
        int attrib[] = { WGL_NUMBER_PIXEL_FORMATS_ARB };
        wglGetPixelFormatAttribivARB(hDC, 1, 0, 1, attrib, nFormatCount);
        // Loop through all the formats and look at each one
        for(i = 0; i < nFormatCount[0]; i++)
        {
            // Query pixel format
            wglGetPixelFormatAttribivARB(hDC, i+1, 0, 9, iAttributes, iResults);
            // Match? Must support OpenGL AND be Accelerated AND draw to Window
            if(iResults[0] == 1 && iResults[1] == WGL_FULL_ACCELERATION_ARB
                && iResults[2] == 1)
                if(iResults[3] == 1) // Double buffered
                if(iResults[4] == WGL_TYPE_RGBA_ARB) // Full Color
                if(iResults[5] >= 16) // Any Depth greater than 16
                if(iResults[6] > 0) // Any Stencil depth (not zero)
                {
                    // We have a candidate, look for most samples if multisampled
                    if(iResults[7] == 1) // Multisampled
                    {
                        if(iResults[8] > nBestMS) // Look for most samples
                        {
                            *nMSFormat = i; // Multisamples
                            nBestMS = iResults[8]; // Looking for the best
                        }
                    }
                }
            else // Not multisampled
            {
                // Good enough for "regular". This will fall through
            }
        }
    }
}

```

```

        *nRegularFormat = i;
    }
}
else
{
    // Old fashioned way...
    // or multisample
    PIXELFORMATDESCRIPTOR pfd = {
        sizeof(PIXELFORMATDESCRIPTOR),
        1,
        PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER,
        PFD_TYPE_RGBA,           // Full color
        32,                      // Color depth
        0,0,0,0,0,0,             // Ignored
        0,0,0,0,                 // Accumulation buffer
        24,                      // Depth bits
        8,                       // Stencil bits
        0,0,0,0,0,0 };           // Some used, some not
    *nRegularFormat = ChoosePixelFormat(hDC, &pfd);
}
}
}

```

Summary

This chapter introduced you to using OpenGL on the Win32 platform. You read about the different driver models and implementations available for Windows and what to watch. You also learned how to enumerate and select a pixel format to get the kind of hardware-accelerated or software rendering support you want. You've now seen the basic framework for a Win32 program that replaces the GLUT framework, so you can write true native Win32 application code.

We also showed you how to create a 3-3-2 palette to enable OpenGL rendering with only 256 available colors for output, and we showed you how to create a full-screen window for games or simulation-type applications. Additionally, we discussed some of the Windows-specific features of OpenGL on Windows, such as support for TrueType fonts and multiple rendering threads.

Finally, we presented the ultimate OpenGL on Win32 sample program, SPHEREWORLD32. This program demonstrated how to use a number of Windows-specific features and WGL extensions if they were available. It also showed you how to construct a well-behaved program that will run on everything from an old 8-bit color display to the latest 32-bit full-color mega-3D game accelerator.

Reference

ChoosePixelFormat	
--------------------------	--

Purpose: Selects the pixel format closest to that specified by the `PIXELFORMATDESCRIPTOR` and that can be supported by the given device context.

Include File: `<wingdi.h>`

Syntax:

```
int ChoosePixelFormat(HDC hDC, CONST
    → PIXELFORMATDESCRIPTOR *ppfd);
```

Description: This function enables you to determine the best available pixel format for a given device context based on the desired characteristics described in the `PIXELFORMATDESCRIPTOR` structure. This returned format index is then used in the `SetPixelFormat` function.

Parameters:

`hDC` `HDC`: The device context for which this function seeks a best-match pixel format.

`ppfd` `PIXELFORMATDESCRIPTOR*`: A pointer to a structure that describes the ideal pixel format being sought. The entire contents of this structure are not pertinent to this function's use. For a complete description of the `PIXELFORMATDESCRIPTOR` structure, see the `DescribePixelFormat` function. The relevant members for this function are as follows:

<code>nSize</code>	<code>WORD</code> : The size of the structure, usually set to <code>sizeof(PIXELFORMATDESCRIPTOR)</code> .
<code>nVersion</code>	<code>WORD</code> : The version number of this structure, set to 1.
<code>dwFlags</code>	<code>DWORD</code> : A set of flags that specify properties of the pixel buffer.
<code>iPixelFormat</code>	<code>BYTE</code> : The color mode (RGBA or color index) type.
<code>cColorBits</code>	<code>BYTE</code> : The depth of the color buffer.
<code>cAlphaBits</code>	<code>BYTE</code> : The depth of the alpha buffer.
<code>cAccumBits</code>	<code>BYTE</code> : The depth of the accumulation buffer.
<code>cDepthBit</code>	<code>BYTE</code> : The depth of the depth buffer.
<code>cStencilBits</code>	<code>BYTE</code> : The depth of the stencil buffer.
<code>cAuxBuffers</code>	<code>BYTE</code> : The number of auxiliary buffers (not supported by Microsoft).
<code>iLayerType</code>	<code>BYTE</code> : The layer type (not supported by Microsoft).

Returns: The index of the nearest matching pixel format for the logical format specified or zero if no suitable pixel format can be found.

See Also: `DescribePixelFormat`, `SetPixelFormat`

DescribePixelFormat

Purpose: Obtains detailed information about a pixel format.

Include File: `<wingdi.h>`

Syntax:

```
int DescribePixelFormat(HDC hDC, int iPixelFormat,
    → UINT nBytes,
    LPIXELFORMATDESCRIPTOR ppfd);
```

Description: This function fills the `PIXELFORMATDESCRIPTOR` structure with information about the pixel format specified for the given device context. It also returns the maximum available pixel format for the device context. If `ppfd` is `NULL`, the function still returns the maximum valid pixel format for the device context. Some fields of the `PIXELFORMATDESCRIPTOR` are not supported by Microsoft's generic implementation of OpenGL, but these values might be supported by individual hardware manufacturers.

Parameters:

`hDC` `HDC`: The device context containing the pixel format of interest.

`iPixelFormat` `int`: The pixel format of interest for the specified device context.

`nBytes` `UINT`: The size of the structure pointed to by `ppfd`. If this value is 0 (zero), no data will be copied to the buffer. It should be set to `sizeof(PIXELFORMATDESCRIPTOR)`.

`ppfd` `LPPIXELFORMATDESCRIPTOR`: A pointer to the `PIXELFORMATDESCRIPTOR` that, on return, will contain the detailed information about the pixel format of interest. The `PIXELFORMATDESCRIPTOR` structure is defined as follows:

```
typedef struct tagPIXELFORMATDESCRIPTOR {
    WORD nSize;
    WORD nVersion;
    DWORD dwFlags;
    BYTE iPixelFormat;
    BYTE cColorBits;
    BYTE cRedBits;
    BYTE cRedShift;
    BYTE cGreenBits;
    BYTE cGreenShift;
    BYTE cBlueBits;
    BYTE cBlueShift;
    BYTE cAlphaBits;
    BYTE cAlphaShift;
    BYTE cAccumBits;
    BYTE cAccumRedBits;
    BYTE cAccumGreenBits;
    BYTE cAccumBlueBits;
    BYTE cAccumAlphaBits;
    BYTE cDepthBits;
    BYTE cStencilBits;
    BYTE cAuxBuffers;
    BYTE iLayerType;
    BYTE bReserved;
    DWORD dwLayerMask;
    DWORD dwVisibleMask;
    DWORD dwDamageMask;
} PIXELFORMATDESCRIPTOR;
```

`nSize` contains the size of the structure. It should always be set to `sizeof(PIXELFORMATDESCRIPTOR)`.

`nVersion` holds the version number of this structure. It should always be set to 1.

dwFlags contains a set of bit flags (see [Table 13.1](#)) that describe properties of the pixel format. Except as noted, these flags are not mutually exclusive.

iPixelFormat specifies the type of pixel data. More specifically, it specifies the color selection mode. Valid values are `GL_TYPE_RGBA` for RGBA color mode or `GL_TYPE_COLORINDEX` for color index mode.

cColorBits specifies the number of color bit planes used by the color buffer, excluding the alpha bit planes in RGBA color mode. In color index mode, it specifies the size of the color buffer.

cRedBits specifies the number of red bit planes in each RGBA color buffer.

cRedShift specifies the shift count for red bit planes in each RGBA color buffer.

cGreenBits specifies the number of green bit planes in each RGBA color buffer.

cGreenShift specifies the shift count for green bit planes in each RGBA color buffer.

cBlueBits specifies the number of blue bit planes in each RGBA color buffer.

cBlueShift specifies the shift count for blue bit planes in each RGBA color buffer.

cAlphaBits specifies the number of alpha bit planes in each RGBA color buffer. This is not supported by the Microsoft generic implementation on Windows versions earlier than Windows 2000.

cAlphaShift specifies the shift count for alpha bit planes in each RGBA color buffer.

cAccumBits is the total number of bit planes in the accumulation buffer. See [Chapter 7](#), "Imaging with OpenGL."

cAccumRedBits is the total number of red bit planes in the accumulation buffer.

cAccumGreenBits is the total number of green bit planes in the accumulation buffer.

cAccumBlueBits is the total number of blue bit planes in the accumulation buffer.

cAccumAlphaBits is the total number of alpha bit planes in the accumulation buffer.

cDepthBits specifies the depth of the depth buffer.

cStencilBits specifies the depth of the stencil buffer.

cAuxBuffers specifies the number of auxiliary buffers. This is not supported by the Microsoft generic implementation.

iLayerType is obsolete. Do not use.

bReserved contains the number of overlay and underlay planes supported by the implementation. Bits 0 through 3 specify the number of overlay planes (up to 15), and bits 4 through 7 specify the number of underlay planes (also up to 15).

dwLayerMask is obsolete. Do not use.

dwVisibleMask is used in conjunction with the *dwLayerMask* to determine whether one layer overlays another. Layers are not supported by the current Microsoft implementation.

dwDamageMask is obsolete. Do not use.

Returns: The maximum pixel format supported by the specified device context or 0 (zero) on failure.

See Also: [ChoosePixelFormat](#), [GetPixelFormat](#), [SetPixelFormat](#)

GetPixelFormat

Purpose: Retrieves the index of the pixel format currently selected for the given device context.

Include File: `<wingdi.h>`

Syntax:

```
int GetPixelFormat(HDC hDC);
```

Description: This function retrieves the selected pixel format for the device context specified. The pixel format index is a one-based positive value.

Parameters:

hDC **HDC:** The device context of interest.

Returns: The index of the currently selected pixel format for the given device or 0 (zero) on failure.

See Also: [DescribePixelFormat](#), [ChoosePixelFormat](#), [SetPixelFormat](#)

SetPixelFormat

Purpose: Sets a device context's pixel format.

Include File: `<wingdi.h>`

Syntax:

```
BOOL SetPixelFormat(HDC hDC, int nPixelFormat,
                   CONST PIXELFORMATDESCRIPTOR *
                   ↗ ppfd);
```

Description: This function actually sets the pixel format for a device context. After the pixel format has been selected for a given device, it cannot be changed. This function must be called before creating an OpenGL rendering context for the device.

Parameters:

hDC **HDC**: The device context whose pixel format is to be set.

nPixelFormat int: Index of the pixel format to be set.

ppfd **LPPIXELFORMATDESCRIPTOR**: A pointer to a **PIXELFORMATDESCRIPTOR** that contains the logical pixel format descriptor. This structure is used internally to record the logical pixel format specification. Its value does not influence the operation of this function.

Returns: **TRUE** if the specified pixel format was set for the given device context; **FALSE** if an error occurs.

See Also: [DescribePixelFormat](#), [GetPixelFormat](#), [ChoosePixelFormat](#)

SwapBuffers

Purpose: Quickly copies the contents of a window's back buffer to the front buffer (foreground).

Include File: `<wingdi.h>`

Syntax:

```
BOOL SwapBuffers(HDC hDC);
```

Description: When a double-buffered pixel format is chosen, a window has a front (displayed) and back (hidden) image buffer. Drawing commands are sent to the back buffer. This function enables you to copy the contents of the hidden back buffer to the displayed front buffer, to support smooth drawing or animation. Note that the buffers may be copied or just swapped by the implementation. After this command is executed, the contents of the back buffer are undefined.

Parameters:

hDC **HDC**: Specifies the device context of the window containing the offscreen and onscreen buffers.

Returns: **TRUE** if the buffers were swapped.

See Also: [glDrawBuffer](#)

wglCreateContext

Purpose: Creates a rendering context suitable for drawing on the specified device context.

Include File: `<wingdi.h>`

Syntax:

```
HGLRC wglCreateContext(HDC hDC);
```

Description: This function creates an OpenGL rendering context suitable for the given windows device context. The pixel format for the device context should be set before the creation of the rendering context. When an application is finished with the rendering context, it should call [wglDeleteContext](#).

Parameters:

- hDC* **HDC**: The device context that will be drawn on by the new rendering context.
- Returns:** The handle to the new rendering context or **NULL** if an error occurs.
- See Also:** [wglCreateLayerContext](#), [wglDeleteContext](#), [wglGetCurrentContext](#), [wglMakeCurrent](#)

wglCreateLayerContext

Purpose: Creates a new OpenGL rendering context suitable for drawing on the specified layer plane.

Include File: `<wingdi.h>`

Syntax:

```
HGLRC wglCreateLayerContext(HDC hDC, int iLayerPlane);
```

Description: This function creates an OpenGL rendering context suitable for the given layer plane. When overlay and underlay planes are supported (only by some hardware implementations), you need a separate OpenGL rendering context for each one. The layer plane with index 0 is the main plane (what you normally render to). Positive indexes are overlay planes, and negative values are underlay planes.

Parameters:

- hDC* **HDC**: The device context that will be drawn on by the new overlay or underlay rendering context.
- iLayerPlane* **int**: The index of the layer plane for which to create the context.
- Returns:** The handle to the new rendering context or **NULL** if an error occurs.
- See Also:** [wglCreateContext](#), [wglDeleteContext](#), [wglGetCurrentContext](#), [wglMakeCurrent](#)

wglCopyContext

Purpose: Copies selected groups of rendering states from one OpenGL context to another.

Include File: `<wingdi.h>`

Syntax:

```
BOOL wglCopyContext(HGLRC hSource, HGLRC hDest,
→      UINT mask);
```

Description: This function can be used to synchronize the rendering state of two OpenGL rendering contexts. Any valid state flags that can be specified with [glPushAttrib](#) can be copied with this function. You can use [GL_ALL_ATTRIB_BITS](#) to copy all attributes.

Parameters:

hSource HGLRC: The source rendering context from which to copy state information.

hDest HGLRC: The destination rendering context to which to copy state information.

mask **UINT**: The handle of the rendering context to be deleted.

Returns: **TRUE** if the rendering context state information is copied.

See Also: [glPushAttrib](#), [wglCreateContext](#), [wglGetCurrentContext](#), [wglMakeCurrent](#)

wglDeleteContext

Purpose: Deletes a rendering context after it is no longer needed by the application.

Include File: `<wingdi.h>`

Syntax:

```
BOOL wglDeleteContext(HGLRC hglrc);
```

Description: This function deletes an OpenGL rendering context. This frees any memory and resources held by the context.

Parameters:

hglrc HGLRC: The handle of the rendering context to be deleted.

Returns: **TRUE** if the rendering context is deleted; **FALSE** if an error occurs. It is an error for one thread to delete a rendering context that is the current context of another thread.

See Also: [wglCreateContext](#), [wglGetCurrentContext](#), [wglMakeCurrent](#)

wglDescribeLayerPlane

Purpose: Retrieves information about overlay and underlay planes of a given pixel format.

Include File: `<wingdi.h>`

Syntax:

```
BOOL wglDescribeLayerPlane(HDC hdc, int
  ↪ iPixelFormat, int iLayerPlane,
                           UINT nBytes,
  ↪ LPLAYERPLANEDESCRIPTOR plpd);
```

Description: This function serves a purpose similar to [DescribePixelFormat](#) but retrieves information about overlay and underlay planes. Layered planes are numbered with negative and positive integers. Plane 0 is the main plane, and negative plane numbers are underlays. Numbers greater than 0 are overlay planes.

Parameters:

hdc **HDC**: The handle of the device context whose layer planes are to be described.

<i>iPixelFormat</i>	<code>int</code> : The pixel format of the desired layer plane.
<i>iLayerPlane</i>	<code>int</code> : The overlay or underlay plane identifier. Negative values are underlays, positive values are overlays, and 0 is the main plane.
<i>nBytes</i>	<code>UINT</code> : The size in bytes of the <code>LAYERPLANEDESCRIPTOR</code> .
<i>plpd</i>	<code>LPLAYERPLANEDESCRIPTOR</code> : Pointer to a <code>LAYERPLANEDESCRIPTOR</code> structure.

Returns: `TRUE` if successful and fills in the data members of the `LAYERPLANEDESCRIPTOR` structure. This structure is defined as follows:

```
typedef struct tagLAYERPLANEDESCRIPTOR {
    WORD nSize;
    WORD nVersion;
    DWORD dwFlags;
    BYTE iPixelFormat;
    BYTE cColorBits;
    BYTE cRedBits;
    BYTE cRedShift;
    BYTE cGreenBits;
    BYTE cGreenShift;
    BYTE cBlueBits;
    BYTE cBlueShift;
    BYTE cAlphaBits;
    BYTE cAlphaShift;
    BYTE cAccumBits;
    BYTE cAccumRedBits;
    BYTE cAccumGreenBits;
    BYTE cAccumBlueBits;
    BYTE cAccumAlphaBits;
    BYTE cDepthBits;
    BYTE cStencilBits;
    BYTE cAuxBuffers;
    BYTE iLayerType;
    BYTE bReserved;
    COLOREF crTransparent;
} LAYERPLANEDESCRIPTOR;
```

nSize contains the size of the structure. It should always be set to `sizeof(LAYERPLANEDESCRIPTOR)`.

nVersion holds the version number of this structure. It should always be set to 1.

dwFlags contains a set of bit flags that describe properties of the pixel format. Except as noted, these flags are not mutually exclusive:

`LPD_SUPPORT_OPENGL` supports OpenGL rendering.

`LPD_SUPPORT_GDI` supports GDI drawing.

`LPD_DOUBLEBUFFER` indicates the layer plane is double-buffered.

`LPD_STEREO` indicates the layer plane is stereoscopic.

`LPD_SWAP_EXCHANGE` means that in double-buffering, the front and back buffers' contents are swapped.

`LPD_SWAP_COPY` means that in double-buffering, the back buffer is copied to the front buffer. The back buffer is unaffected.

`LPD_TRANSPARENT` indicates the `crTransparent` member of this structure contains a color value that should be considered the transparent color.

`LPD_SHARE_DEPTH` indicates the layer plane shares the depth buffer with the main plane.

`LPD_SHARE_STENCIL` indicates the layer plane shares the stencil buffer with the main plane.

`LPD_SHARE_ACCUM` indicates the layer plane shares the accumulation buffer with the main plane.

`iPixelFormat` specifies the type of pixel data. More specifically, it specifies the color selection mode. Valid values are `LPD_TYPE_RGBA` for RGBA color mode or `LPD_TYPE_COLORINDEX` for color index mode.

`cColorBits` specifies the number of color bit planes used by the color buffer, excluding the alpha bit planes in RGBA color mode. In color index mode, it specifies the size of the color buffer.

`cRedBits` specifies the number of red bit planes in each RGBA color buffer.

`cRedShift` specifies the shift count for red bit planes in each RGBA color buffer.

`cGreenBits` specifies the number of green bit planes in each RGBA color buffer.

`cGreenShift` specifies the shift count for green bit planes in each RGBA color buffer.

`cBlueBits` specifies the number of blue bit planes in each RGBA color buffer.

`cBlueShift` specifies the shift count for blue bit planes in each RGBA color buffer.

`cAlphaBits` specifies the number of alpha bit planes in each RGBA color buffer. This is not supported by the Microsoft generic implementation on Windows versions prior to Windows 2000.

`cAlphaShift` specifies the shift count for alpha bit planes in each RGBA color buffer. This is not supported by the Microsoft implementation.

`cAccumBits` is the total number of bit planes in the accumulation buffer.

`cAccumRedBits` is the total number of red bit planes in the accumulation buffer.

`cAccumGreenBits` is the total number of green bit planes in the accumulation buffer.

`cAccumBlueBits` is the total number of blue bit planes in the accumulation buffer.

`cAccumAlphaBits` is the total number of alpha bit planes in the accumulation buffer.

cDepthBits specifies the depth of the depth buffer.

cStencilBits specifies the depth of the stencil buffer.

cAuxBuffers specifies the number of auxiliary buffers. This is not supported by the Microsoft generic implementation.

iLayerType is the layer plane number. Positive values are overlays, and negative numbers are underlays.

bReserved is not used. It must be 0 (zero).

crTransparent indicates that when the `LPD_TRANSPARENT` flag is set, this is the transparent color value. Typically, it is set to black. The color is specified as a Windows `COLORREF` value. You can use the Windows RGB macro to construct this value.

See Also: [DescribePixelFormat](#), [wglCreateLayerContext](#)

wglGetCurrentContext

Purpose: Retrieves a handle to the current thread's OpenGL rendering context.

Include File: `<wingdi.h>`

Syntax:

```
HGLRC wglGetCurrentContext(void);
```

Description: Each thread of an application can have its own current OpenGL rendering context. You can use this function to determine which rendering context is currently active for the calling thread.

Returns: If the calling thread has a current rendering context, this function returns its handle. If not, the function returns `NULL`.

See Also: [wglCreateContext](#), [wglDeleteContext](#), [wglMakeCurrent](#), [wglGetCurrentDC](#)

wglGetCurrentDC

Purpose: Gets the windows device context associated with the current OpenGL rendering context.

Include File: `<wingdi.h>`

Syntax:

```
HDC wglGetCurrentDC(void);
```

Description: This function enables you to acquire the windows device context of the window associated with the current OpenGL rendering context. It is typically used to obtain a windows device context to combine OpenGL and GDI drawing functions in a single window.

Returns: If the current thread has a current OpenGL rendering context, this function returns the handle to the windows device context associated with it. Otherwise, the return value is `NULL`.

See Also: [wglGetCurrentContext](#)

wglGetProcAddress

Purpose: Gets the address of an extension function.

Include File: `<gl.h>`

Syntax:

```
PROC wglGetProcAddress(LPSTR lpszProc);
```

Description: This function retrieves the address of an extension function. If the extension function is not available, a `NULL` pointer is returned.

Parameters:

lpszProc `LPSTR`: The name of the extension function.

Returns: None.

wglGetCurrent

Purpose: Makes a given OpenGL rendering context current for the calling thread and associates it with the specified device context.

Include File: `<wingdi.h>`

Syntax:

```
BOOL wglGetCurrent(HDC hDC, HGLRC HRC);
```

Description: This function makes the specified rendering context the current rendering context for the calling thread. This rendering context is associated with the given windows device context. The device context need not be the same as that used in the call to `wglCreateContext`, as long as the pixel format is the same for both and they both exist on the same physical device (not one on the screen and one on a printer). Any outstanding OpenGL commands for the previous rendering context are flushed before the new rendering context is made current. You can also use this function to make no rendering context active, by calling it with `NULL` for the *HRC* parameter.

Parameters:

hDC `HDC`: The device context that will be used for all OpenGL drawing operations performed by the calling thread.

hRC **HGLRC:** The rendering context to make current for the calling thread.

Returns: **TRUE** on success or **FALSE** if an error occurs. If an error occurs, no rendering context will remain current for the calling thread.

See Also: [wglCreateContext](#), [wglDeleteContext](#), [wglGetCurrentContext](#), [wglGetCurrentDC](#)

wglShareLists

Purpose: Allows multiple rendering contexts to share display lists.

Include File: `<wingdi.h>`

Syntax:

```
BOOL wglShareLists(HGLRC hRC1, HGLRC hRC2);
```

Description: A display list is a list of "precompiled" OpenGL commands and functions (see [Chapter 11](#), "It's All About the Pipeline: Faster Geometry Throughput"). Memory is allocated for the storage of display lists within each rendering context. As display lists are created within that rendering context, it has access to its own display list memory. This function allows multiple rendering contexts to share this memory. This capability is particularly useful when large display lists are used by multiple rendering contexts or threads to save memory. Any number of rendering contexts can share the same memory for display lists. This memory is not freed until the last rendering context using that space is deleted. When using a shared display list space between threads, you should synchronize display list creation and usage.

Parameters:

hRC1 **HGLRC:** The rendering context with which to share display list memory.

hRC2 **HGLRC:** The rendering context that will share the display list memory with ***hRC1***. No display lists for ***hRC2*** should be created until after its display list memory is shared.

Returns: **TRUE** if the display list space is shared; **FALSE** if they are not.

See Also: [glIsList](#), [glNewList](#), [glCallList](#), [glCallLists](#), [glListBase](#), [glDeleteLists](#), [glEndList](#), [glGenLists](#)

wglSwapLayerBuffers

Purpose: Swaps the front and back buffers in the overlay, underlay, and main planes belonging to the specified device context.

Include File: `<wingdi.h>`

Syntax:

```
BOOL wglSwapLayerBuffers(HDC hDC, UINT fuPlanes);
```

Description: When a double-buffered pixel format is chosen, a window has a front (displayed) and back (hidden) image buffer. Drawing commands are sent to the back buffer. This function enables you to copy the contents of the hidden back buffer to the displayed front buffer, to support smooth drawing or animation. Note that the buffers are not really swapped. After this command is executed, the contents of the back buffer are undefined.

Parameters:

hDC **HDC:** The device context of the window containing the offscreen and onscreen buffers.

fuPlanes **UINT:** The device context of the window containing the offscreen and onscreen buffers.

Returns: True if the buffers were swapped.

See Also: [glSwapBuffers](#)

wglUseFontBitmaps

Purpose: Creates a set of OpenGL display list bitmaps for the currently selected GDI font.

Include File: `<wingdi.h>`

Syntax:

```
BOOL wglUseFontBitmaps(HDC hDC, DWORD dwFirst,
→ DWORD dwCount, DWORD dwListBase);
```

Description: This function takes the font currently selected in the device context specified by *hDC* and creates a bitmap display list for each character, starting at *dwFirst* and running for *dwCount* characters. The display lists are created in the currently selected rendering context and are identified by numbers starting at *dwListBase*. Typically, this function enables you to draw text into an OpenGL double-buffered scene because the Windows GDI will not allow operations to the back buffer of a double-buffered window. This function also enables you to label OpenGL objects onscreen.

Parameters:

hDC **HDC:** The Windows GDI device context from which the font definition is to be derived. You can change the font used by creating and selecting the desired font into the device context.

dwFirst **DWORD:** The ASCII value of the first character in the font to use for building the display lists.

dwCount **DWORD:** The consecutive number of characters in the font to use succeeding the character specified by *dwFirst*.

dwListBase **DWORD:** The display list base value to use for the first display list character.

Returns: **TRUE** if the display lists could be created; **FALSE** otherwise.

See Also: [wglUseFontOutlines](#), [glIsList](#), [glNewList](#), [glCallList](#), [glCallLists](#), [glListBase](#), [glDeleteLists](#), [glEndList](#), [glGenLists](#)

wglUseFontOutlines

Purpose: Creates a set of OpenGL 3D display lists for the currently selected GDI font.

Include File: `<wingdi.h>`

Syntax:

```
BOOL wglUseFontOutlines(HDC hDC, DWORD first,
→ DWORD count, DWORD listBase,
→ FLOAT deviation, FLOAT
→ extrusion, int format,
→ LPGLYPHMETRICSFLOAT lpgmf);
```

Description: This function takes the TrueType font currently selected into the GDI device context *hDC* and creates a 3D outline for *count* characters starting at *first*. The display lists are numbered starting at the value *listBase*. The outline can be composed of line segments or polygons as specified by the *format* parameter. The character cell used for the font extends 1.0 unit length along the x- and y-axes. The parameter *extrusion* supplies the length along the negative z-axis on which the character is extruded. The *deviation* is an amount 0 or greater that determines the chordal deviation from the original font outline. This function will work only with TrueType fonts. Additional character data is supplied in the *lpgmf* array of `GLYPHMETRICSFLOAT` structures.

Parameters:

hDC `HDC`: Device context of the font.

first `DWORD`: First character in the font to be turned into a display list.

count `DWORD`: Number of characters in the font to be turned into display lists.

listBase `DWORD`: The display list base value to use for the first display list character.

deviation `FLOAT`: The maximum chordal deviation from the true outlines.

extrusion `FLOAT`: Extrusion value in the negative z direction.

format `int`: A value that specifies whether the characters should be composed of line segments or polygons in the display lists. It may be one of the following values:

`WGL_FONT_LINES`: Use line segments to compose character.

`WGL_FONT_POLYGONS`: Use polygons to compose character.

lpgmf `LPGLYPHMETRICSFLOAT`: Address of an array to receive glyphs metric data. Each array element is filled with data pertaining to its character's display list. Each is defined as follows:

```
typedef struct _GLYPHMETRICSFLOAT { // gmf
    FLOAT    gmfBlackBoxX;
    FLOAT    gmfBlackBoxY;
    POINTFLOAT gmfptGlyphOrigin;
    FLOAT    gmfCellIncX;
    FLOAT    gmfCellIncY;
} GLYPHMETRICSFLOAT;
```

gmfBlackBoxX specifies the width of the smallest rectangle that completely encloses the character.

gmfBlackBoxY specifies the height of the smallest rectangle that completely encloses the character.

gmfptGlyphOrigin specifies the x and y coordinates of the upper-left corner of the rectangle that completely encloses the character. The **POINTFLOAT** structure is defined as

```
typedef struct _POINTFLOAT { // ptf
    FLOAT    x;           // The horizontal coordinate
    ↪ of a point
    FLOAT    y;           // The vertical coordinate of
    ↪ a point
} POINTFLOAT;
```

gmfCellIncX specifies the horizontal distance from the origin of the current character cell to the origin of the next character cell.

gmfCellIncY specifies the vertical distance from the origin of the current character cell to the origin of the next character cell.

Returns: `TRUE` if the display lists could be created; `FALSE` otherwise.

See Also: `wglUseFontBitmaps`, `glIsList`, `glNewList`, `glCallList`, `glCallLists`, `glListBase`, `glDeleteLists`, `glEndList`, `glGenLists`

Chapter 14. OpenGL on MacOS X

by Michael Sweet

WHAT YOU'LL LEARN IN THIS CHAPTER:

How To	Functions You'll Use
Choose appropriate pixel formats for OpenGL rendering	<code>aglChoosePixelFormat</code>
Manage OpenGL drawing contexts	<code>aglCreateContext</code> , <code>aglDestroyContext</code> , <code>aglGetCurrentContext</code> , <code>aglSetDrawable</code>
Do double-buffered drawing	<code>aglSwapBuffers</code>
Create bitmap text fonts	<code>aglUseFont</code>

This chapter discusses the OpenGL interfaces on MacOS X. We cover both AGL and NSOpenGL, the Carbon and Cocoa interfaces, respectively. You learn how to create and manage OpenGL contexts as well as how to create OpenGL drawing areas for Carbon and Cocoa applications. This chapter also shows you how to use GLUT on MacOS X with all the examples in this book.

The Basics

OpenGL on MacOS X is exposed via three APIs: AGL for Carbon applications, NSOpenGL for Cocoa applications, and CGL for applications that need direct access to the screen. This chapter discusses using AGL and NSOpenGL; the CGL API provides similar functionality, but because it bypasses the windowing system (no OpenGL in windows, only full screen), we do not cover it. There is, however, a tutorial for CGL on the book's Web site (see [Appendix A](#), "Further Reading").

In addition to the three standard APIs, the Apple developer tools also include a GLUT port that can be used to compile and use any of the GLUT-based examples in this book.

Frameworks

All the APIs discussed in this chapter are provided via *frameworks* in MacOS X. A framework is a collection of header files and libraries that provide specific functionality. [Table 14.1](#) lists the OpenGL-related frameworks discussed in this chapter.

Table 14.1. OpenGL-Related Frameworks in MacOS X

Name	Description
AGL	Provides the OpenGL interface to Carbon (QuickDraw) windows.
ApplicationServices	Provides a common API for all GUI application services. This framework is used for all OpenGL applications.
Carbon	Provides a common API for MacOS X and earlier. This framework is used in combination with the AGL and OpenGL frameworks.
Cocoa	Provides the Aqua user interface classes. This framework is used in combination with the OpenGL framework.
OpenGL	Provides the common OpenGL API to the graphics hardware. This framework is used with any of the other frameworks to provide OpenGL rendering support.

Using the GLUT API

The GLUT API is provided in the OpenGL examples folder of the Apple Developer Tools CD-ROM. To use the GLUT API, copy this folder to disk and then add the GLUT framework in the directory to the list of frameworks in the Xcode window for your application. Use the following linker options if you are building your software with a makefile:

```
-framework /path/to/GLUT.framework -framework AGL -framework
OpenGL -framework Carbon -framework ApplicationServices
```

Using the AGL and Carbon APIs

The AGL and Carbon APIs provide a relatively simple interface for C and C++ programs to display OpenGL graphics. Because Carbon-based applications are supported on MacOS 8 through X, you can use the AGL and Carbon APIs to develop applications that work on multiple versions of MacOS.

Programs of this type require the AGL, ApplicationServices, Carbon, and OpenGL frameworks; you can start with a Carbon-based application in the Xcode application and add the AGL and OpenGL frameworks or use the following linker options:

```
-framework AGL -framework OpenGL -framework Carbon -framework ApplicationServices
```

As you'll see in the following sections, AGL functions start with the prefix `agl`.

Pixel Formats

AGL exposes multiple *pixel formats* that provide different rendering capabilities. For example, a particular graphics card might be able to provide full-scene antialiasing with a 16-bit depth buffer but not with a 24-bit or 32-bit depth buffer due to limited memory or other implementation-specific issues.

The `aglChoosePixelFormat` function allows an application to choose an appropriate pixel format using a list of integer attributes describing the desired output capabilities. Table 14.2 lists the attribute constants defined by the AGL API.

`AGL_ACCELERATED`

This constant specifies that a hardware-accelerated pixel format is required.

`AGL_ACCUM_ALPHA_SIZE`

The number that follows specifies the minimum number of alpha bits that are required in the accumulation buffer.

`AGL_ACCUM_BLUE_SIZE`

The number that follows specifies the minimum number of blue bits that are required in the accumulation buffer.

`AGL_ACCUM_GREEN_SIZE`

The number that follows specifies the minimum number of green bits that are required in the accumulation buffer.

`AGL_ACCUM_RED_SIZE`

The number that follows specifies the minimum number of red bits that are required in the accumulation buffer.

`AGL_ALPHA_SIZE`

The number that follows specifies the minimum number of alpha bits that are required.

`AGL_AUX_BUFFERS`

The number that follows specifies the number of auxiliary buffers that are required.

`AGL_BACKING_STORE`

This constant specifies that only formats that support a full backing store capability should be used.

`AGL_BLUE_SIZE`

The number that follows specifies the minimum number of blue bits that are required.

AGL_BUFFER_SIZE

The number that follows specifies the number of color index bits that are desired.

AGL_CLOSEST_POLICY

This constant specifies that the sizes specified should be used as the ideal requirements for the pixel format, and the pixel format should use the closest number of bits possible.

AGL_DEPTH_SIZE

The number that follows specifies the minimum number of depth bits that are required.

AGL_DOUBLEBUFFER

This constant specifies that a double-buffered visual is desired.

AGL_FULLSCREEN

This constant specifies that the pixel format is for full-screen rendering.

AGL_GREEN_SIZE

The number that follows specifies the minimum number of green bits that are required.

AGL_LEVEL

This constant specifies the buffer level in the number that follows; 0 is the main buffer, 1 is the first overlay buffer, -1 is the first underlay buffer, and so forth.

AGL_MINIMUM_POLICY

This constant specifies that the sizes specified should be used as the minimum requirements for the pixel format, and the pixel format should use the minimum number of bits possible.

AGL_MAXIMUM_POLICY

This constant specifies that the sizes specified should be used as the minimum requirements for the pixel format, and the pixel format should use the maximum number of bits possible.

AGL_NONE

This constant specifies the end of the attribute list.

AGL_OFFSCREEN

This constant specifies that the pixel format is for an offscreen buffer.

AGL_PIXEL_SIZE

The number that follows specifies the total number of bits that are used for a pixel.

AGL_RED_SIZE

The number that follows specifies the minimum number of red bits that are required.

AGL_RGBA

This constant specifies that an RGBA visual is desired.

AGL_STENCIL_SIZE

The number that follows specifies the minimum number of stencil bits that are required.

AGL_STEREO

This constant specifies that a stereo visual is desired; stereo visuals provide separate left and right eye images.

Table 14.2. AGL Pixel Format Attribute List Constants

Constant	Description

You might use the following code to find a double-buffered RGB pixel format:

```
GLint attributes[] = {
    AGL_RGBA,
    AGL_DOUBLEBUFFER,
    AGL_RED_SIZE, 4,
    AGL_GREEN_SIZE, 4,
    AGL_BLUE_SIZE, 4,
    AGL_DEPTH_SIZE, 16,
    AGL_NONE
};

AGLPixelFormat format;
format = aglChoosePixelFormat(NULL, 0, attributes);
```

Note that the last value is required to be `AGL_NONE`. After `aglChoosePixelFormat` is called, an `AGLPixelFormat` value is returned; this value provides information on the correct pixel format to use. If no matching pixel format can be found, a `NULL` pointer is returned, and you should retry with different attributes or inform the users that the window cannot be displayed on their system.

Managing Contexts

AGL provides four functions for managing OpenGL rendering contexts: `aglCreateContext`, `aglDestroyContext`, `aglSetCurrentContext`, and `aglSetDrawable`. The `aglCreateContext` function creates an OpenGL context using the `AGLPixelFormat` value returned by the `aglChoosePixelFormat` function:

```
AGLContext context;
context = aglCreateContext(format, NULL);
```

The `aglCreateContext` function accepts two arguments: the pixel format and a context to share display list, texture object, and vertex array information with. Pass `NULL` if you do not want to share information with another context.

After you have created the context, it needs to be bound to a window or offscreen buffer using the `aglSetDrawable` function:

```
WindowPtr window;
aglSetDrawable(context, GetWindowPort(window));
```

Next, call the `aglSetCurrentContext` function to use the context to do OpenGL rendering:

```
aglSetCurrentContext(context);
```

Finally, when you are done using the context, call the `aglDestroyContext` function to release the resources that were used by the context:

```
aglDestroyContext(context);
```

Doing Double-Buffered Rendering

You enable double-buffered rendering by using the appropriate pixel format. The drawing code in your program then merely needs to call the `aglSwapBuffers` function after doing any OpenGL rendering to make it visible:

```
aglSwapBuffers(context);
```

Your First AGL Program

Listing 14.1 shows a basic Carbon application that creates a window for OpenGL rendering and displays a spinning cube. The application responds to mouse clicks and drags to alter the spinning of the cube and exits after the user closes the window. Figure 14.1 shows the result.

Listing 14.1. The CARBON Sample Program

```
/*
 * Include necessary headers...
 */
#include <stdio.h>
#include <stdlib.h>
#include <Carbon/Carbon.h>
#include <AGL/agl.h>
/*
 * Globals...
 */
float      CubeRotation[3],      /* Rotation of cube */
           CubeRate[3];      /* Rotation rate of cube */
int       CubeMouseButton,      /* Button that was pressed */
         CubeMouseX,      /* Start X position of mouse */
         CubeMouseY;      /* Start Y position of mouse */
int       CubeX,                /* X position of window */
         CubeY,                /* Y position of window */
         CubeWidth,           /* Width of window */
```

```

        CubeHeight;           /* Height of window */
int      CubeVisible;        /* Is the window visible? */
AGLContext CubeContext;      /* OpenGL context */

/*
 * Functions...
 */
void      DisplayFunc(void);
static pascal OSStatus
          EventHandler(EventHandlerCallRef nextHandler,
                        EventRef event, void *userData);

void      IdleFunc(void);
void      MotionFunc(int x, int y);
void      MouseFunc(int button, int state, int x, int y);
void      ReshapeFunc(int width, int height);

/*
 * 'main()' - Main entry for example program.
 */
int      main(int argc,
            char *argv[])
{
    AGLPixelFormat      format;           /* OpenGL pixel format */
    WindowPtr           window;          /* Window */
    int                 winattrs;         /* Window attributes */
    Str255              title;           /* Title of window */
    Rect                rect;            /* Rectangle definition */
    EventHandlerUPP    handler;          /* Event handler */
    EventLoopTimerUPP  thandler;         /* Timer handler */
    EventLoopTimerRef   timer;           /* Timer for animating the window */
    ProcessSerialNumber psn;            /* Process serial number */
    static EventTypeSpec events[] =      /* Events we are interested in... */
    {
        { kEventClassMouse, kEventMouseDown },
        { kEventClassMouse, kEventMouseUp },
        { kEventClassMouse, kEventMouseMoved },
        { kEventClassMouse, kEventMouseDragged },
        { kEventClassWindow, kEventWindowDrawContent },
        { kEventClassWindow, kEventWindowShown },
        { kEventClassWindow, kEventWindowHidden },
        { kEventClassWindow, kEventWindowActivated },
        { kEventClassWindow, kEventWindowDeactivated },
        { kEventClassWindow, kEventWindowClose },
        { kEventClassWindow, kEventWindowBoundsChanged }
    };
    static GLint          attributes[] = /* OpenGL attributes */
    {
        AGL_RGBA,
        AGL_GREEN_SIZE, 1,
        AGL_DOUBLEBUFFER,
        AGL_DEPTH_SIZE, 16,
        AGL_NONE
    };

        //Set initial values for window
const int      origWinHeight = 628;
const int      origWinWidth = 850;
const int      origWinXOffset = 50;
const int      origWinYOffset = 50;
/*
 * Create the window...
 */

```

```

CubeContext = 0;
CubeVisible = 0;
SetRect(&rect, origWinXOffset, origWinYOffset, origWinWidth, origWinHeight);
winattrs = kWindowStandardHandlerAttribute | kWindowCloseBoxAttribute |
           kWindowCollapseBoxAttribute | kWindowFullZoomAttribute |
           kWindowResizableAttribute | kWindowLiveResizeAttribute;
winattrs &= GetAvailableWindowAttributes(kDocumentWindowClass);
strcpy(title + 1, "Carbon OpenGL Example");
title[0] = strlen(title + 1);
CreateNewWindow(kDocumentWindowClass, winattrs, &rect, &window);
SetWTitle(window, title);
handler = NewEventHandlerUPP(EventHandler);
InstallWindowEventHandler(window, handler,
                           sizeof(events) / sizeof(events[0]),
                           events, NULL, 0L);
thandler =
    NewEventLoopTimerUPP((void (*)(EventLoopTimerRef, void *))IdleFunc);
InstallEventLoopTimer(GetMainEventLoop(), 0, 0, thandler,
                      0, &timer);
GetCurrentProcess(&psn);
SetFrontProcess(&psn);
DrawGrowIcon(window);

ShowWindow(window);
/*
 * Create the OpenGL context and bind it to the window...
 */
format      = aglChoosePixelFormat(NULL, 0, attributes);
CubeContext = aglCreateContext(format, NULL);
if (!CubeContext)
{
    puts("Unable to get OpenGL context!");
    return (1);
}
aglDestroyPixelFormat(format);
aglSetDrawable(CubeContext, GetWindowPort(window));
/*
 * Setup remaining globals...
 */
CubeX        = 50;
CubeY        = 50;
CubeWidth    = 400;
CubeHeight   = 400;
CubeRotation[0] = 45.0f;
CubeRotation[1] = 45.0f;
CubeRotation[2] = 45.0f;
CubeRate[0]   = 1.0f;
CubeRate[1]   = 1.0f;
CubeRate[2]   = 1.0f;
//Set the initial size of the cube
ReshapeFunc(origWinWidth - origWinXOffset, origWinHeight - origWinYOffset);
/*
 * Loop forever...
 */

for (++)
{
    if (CubeVisible)
        SetEventLoopTimerNextFireTime(timer, 0.05);
    RunApplicationEventLoop();
    if (CubeVisible)
    {

```

```

/*
 * Animate the cube...
 */
DisplayFunc();
}
}
/*
 * 'DisplayFunc()' - Draw a cube.
 */
void
DisplayFunc(void)
{
    int             i, j;           /* Looping vars */
float           aspectRatio,windowWidth, windowHeight;
    static const GLfloat  corners[8][3] = /* Corner vertices */
{
    {
        { 1.0f,  1.0f,  1.0f }, /* Front top right */
        { 1.0f, -1.0f,  1.0f }, /* Front bottom right */
        { -1.0f, -1.0f,  1.0f }, /* Front bottom left */
        { -1.0f,  1.0f,  1.0f }, /* Front top left */
        { 1.0f,  1.0f, -1.0f }, /* Back top right */
        { 1.0f, -1.0f, -1.0f }, /* Back bottom right */
        { -1.0f, -1.0f, -1.0f }, /* Back bottom left */
        { -1.0f,  1.0f, -1.0f } /* Back top left */
    };
    static const int    sides[6][4] = /* Sides */
{
    {
        { 0, 1, 2, 3 },           /* Front */
        { 4, 5, 6, 7 },           /* Back */
        { 0, 1, 5, 4 },           /* Right */
        { 2, 3, 7, 6 },           /* Left */
        { 0, 3, 7, 4 },           /* Top */
        { 1, 2, 6, 5 }            /* Bottom */
    };
    static const GLfloat colors[6][3] = /* Colors */
{
    {
        { 1.0f, 0.0f, 0.0f },     /* Red */
        { 0.0f, 1.0f, 0.0f },     /* Green */
        { 1.0f, 1.0f, 0.0f },     /* Yellow */
        { 0.0f, 0.0f, 1.0f },     /* Blue */
        { 1.0f, 0.0f, 1.0f },     /* Magenta */
        { 0.0f, 1.0f, 1.0f }      /* Cyan */
    };
/*
 * Set the current OpenGL context...
 */
aglSetCurrentContext(CubeContext);
/*
 * Clear the window...
 */
glViewport(0, 0, CubeWidth, CubeHeight);
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
/*
 * Setup the matrices...
 */
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
aspectRatio = (GLfloat)CubeWidth / (GLfloat)CubeHeight;
if(CubeWidth <= CubeHeight)

```

```

{
    windowHeight = 2.0f;
    windowHeight = 2.0f / aspectRatio;
    glOrtho(-2.0f, 2.0f, -windowHeight, windowHeight, 2.0f, -2.0f);
}
else
{
    windowHeight = 2.0f * aspectRatio;
    windowHeight = 2.0f;
    glOrtho(-windowHeight, windowHeight, -2.0f, 2.0f, 2.0f, -2.0f);
}

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glRotatef(CubeRotation[0], 1.0f, 0.0f, 0.0f);
glRotatef(CubeRotation[1], 0.0f, 1.0f, 0.0f);
glRotatef(CubeRotation[2], 0.0f, 0.0f, 1.0f);
/*
 * Draw the cube...
 */
glEnable(GL_DEPTH_TEST);
glBegin(GL_QUADS);
for (i = 0; i < 6; i++)
{
    glColor3fv(colors[i]);
    for (j = 0; j < 4; j++)
        glVertex3fv(corners[sides[i][j]]);
}
glEnd();
/*
 * Swap the front and back buffers...
 */
aglSwapBuffers(CubeContext);
}
/*
 * 'EventHandler()' - Handle window and mouse events from the window manager.
*/
static pascal OSStatus           // 0 - noErr on success or error code
EventHandler(EventHandlerCallRef nextHandler,    /* I - Next handler to call */
             EventRef                 /* I - Event reference */
             void                     /* I - User data (not used) */
{
    UInt32          kind;           /* Kind of event */
    Rect            rect;           /* New window size */
    EventMouseButton button;       /* Mouse button */
    Point           point;          /* Mouse position */
    kind = GetEventKind(event);
    if (GetEventClass(event) == kEventClassWindow)
    {
        switch (kind)
        {
            case kEventWindowDrawContent :
                if (CubeVisible && CubeContext)
                    DisplayFunc();
                break;
            case kEventWindowBoundsChanged :
                GetEventParameter(event, kEventParamCurrentBounds, typeQDRectangle,
                                  NULL, sizeof(Rect), NULL, &rect);
                CubeX = rect.left;
                CubeY = rect.top;
                if (CubeContext)

```

```

        aglUpdateContext(CubeContext);
        ReshapeFunc(rect.right - rect.left, rect.bottom - rect.top);
        if (CubeVisible && CubeContext)
            DisplayFunc();
        break;

    case kEventWindowShown :
        CubeVisible = 1;
        if (CubeContext)
            DisplayFunc();
        break;
    case kEventWindowHidden :
        CubeVisible = 0;
        break;
    case kEventWindowClose :
        ExitToShell();
        break;
    }
}
else
{
    switch (kind)
    {
        case kEventMouseDown :
            GetEventParameter(event, kEventParamMouseButton, typeMouseButton,
                             NULL, sizeof(EventMouseButton), NULL, &button);
            GetEventParameter(event, kEventParamMouseLocation, typeQDPoint,
                             NULL, sizeof(Point), NULL, &point);
            if (point.v < CubeY ||
                (point.v > (CubeY + CubeHeight - 8) &&
                 point.h > (CubeX + CubeWidth - 8)))
                return (CallNextEventHandler(nextHandler, event));
            MouseFunc(button, 0, point.h, point.v);
            break;
        case kEventMouseUp :
            GetEventParameter(event, kEventParamMouseButton, typeMouseButton,
                             NULL, sizeof(EventMouseButton), NULL, &button);
            GetEventParameter(event, kEventParamMouseLocation, typeQDPoint,
                             NULL, sizeof(Point), NULL, &point);

            if (point.v < CubeY ||
                (point.v > (CubeY + CubeHeight - 8) &&
                 point.h > (CubeX + CubeWidth - 8)))
                return (CallNextEventHandler(nextHandler, event));
            MouseFunc(button, 1, point.h, point.v);
            break;
        case kEventMouseDragged :
            GetEventParameter(event, kEventParamMouseLocation, typeQDPoint,
                             NULL, sizeof(Point), NULL, &point);
            if (point.v < CubeY ||
                (point.v > (CubeY + CubeHeight - 8) &&
                 point.h > (CubeX + CubeWidth - 8)))
                return (CallNextEventHandler(nextHandler, event));
            MotionFunc(point.h, point.v);
            break;
        default :
            return (CallNextEventHandler(nextHandler, event));
    }
}
/*
 * Return whether we handled the event...
 */

```

```

    return (noErr);
}
/*
 * 'IdleFunc()' - Rotate and redraw the cube.
 */
void
IdleFunc(void)
{
    CubeRotation[0] += CubeRate[0];
    CubeRotation[1] += CubeRate[1];
    CubeRotation[2] += CubeRate[2];

    QuitApplicationEventLoop();
}
/*
 * 'MotionFunc()' - Handle mouse pointer motion.
 */
void
MotionFunc(int x,                      /* I - X position */
           int y)                      /* I - Y position */
{
/*
 * Get the mouse movement...
 */
    x -= CubeMouseX;
    y -= CubeMouseY;
/*
 * Update the cube rotation rate based upon the mouse movement and
 * button...
 */
    switch (CubeMouseButton)
    {
        case 0 :                      /* Button 1 */
            CubeRate[0] = 0.01f * y;
            CubeRate[1] = 0.01f * x;
            CubeRate[2] = 0.0f;
            break;
        case 1 :                      /* Button 2 */
            CubeRate[0] = 0.0f;
            CubeRate[1] = 0.01f * y;
            CubeRate[2] = 0.01f * x;
            break;
        default :                     /* Button 3 */
            CubeRate[0] = 0.01f * y;
            CubeRate[1] = 0.0f;
            CubeRate[2] = 0.01f * x;
            break;
    }
}

/*
 * 'MouseFunc()' - Handle mouse button press/release events.
 */
void
MouseFunc(int button,                  /* I - Button that was pressed */
          int state,                 /* I - Button state (0 = down) */
          int x,                     /* I - X position */
          int y)                     /* I - Y position */
{
/*
 * Only respond to button presses...
 */
}

```

```

if (state)
    return;
/*
 * Save the mouse state...
 */
CubeMouseButton = button;
CubeMouseX      = x;
CubeMouseY      = y;
/*
 * Zero-out the rotation rates...
 */
CubeRate[0] = 0.0f;
CubeRate[1] = 0.0f;
CubeRate[2] = 0.0f;
}
/*
 * 'ReshapeFunc()' - Resize the window.
 */
void
ReshapeFunc(int width,           /* I - Width of window */
            int height)        /* I - Height of window */
{
    CubeWidth  = width;
    CubeHeight = height;
}

```

Figure 14.1. The AGL spinning cube example.



Using Bitmap Fonts

The AGL framework provides a single function for using bitmap fonts for OpenGL rendering: `aglUseFont`. This function works in conjunction with the Carbon `GetFNum` function and OpenGL `glGenLists` function to extract bitmaps and create display lists for each character you want in the font. The following code demonstrates how to extract the visible ASCII characters from the Courier

New Bold font at a 14-pixel-high size:

```
GLint listbase;
short font;
Str255 fontname;
strcpy(fontname + 1, "Courier New");
fontname[0] = strlen(fontname + 1);
GetFNum(fontname, &font);
listbase = glGenLists(96);
aglUseFont(context, font, bold, 14, ' ', 96, listbase);
```

The font number is used along with a font style (`bold` in this case) to select the actual font to render. The fifth and sixth arguments are the starting character and the number of characters to extract, respectively. The `listbase` variable in the example points to the start of a block of 96 consecutive display lists to use for each character.

After the characters are extracted, you can draw text using a combination of the `glRasterPos()`, `glPushAttrib()`, `glListBase()`, `glCallLists()`, and `glPopAttrib()` functions, as follows:

```
char *s = "Hello, World!";
glPushAttrib(GL_LIST_BIT);
glListBase(listbase - ' ');
glRasterPos3f(0.0f, 0.0f, 0.0f);
glCallLists(strlen(s), GL_BYTE, s);
glPopAttrib();
```

The example in Listing 14.2 uses this code to draw the names of each side of the cube. Figure 14.2 shows the result.

Listing 14.2. The CARBONFONTS Sample Program

```
/*
 * Include necessary headers...
 */
#include <stdio.h>
#include <stdlib.h>
#include <Carbon/Carbon.h>
#include <AGL/agl.h>
/*
 * Globals...
 */
float      CubeRotation[3],          /* Rotation of cube */
           CubeRate[3];          /* Rotation rate of cube */
int        CubeMouseButton,          /* Button that was pressed */
           CubeMouseX,           /* Start X position of mouse */
           CubeMouseY;           /* Start Y position of mouse */
int        CubeX,                  /* X position of window */
           CubeY,                  /* Y position of window */
           CubeWidth,             /* Width of window */
           CubeHeight;            /* Height of window */
int        CubeVisible;            /* Is the window visible? */
AGLContext CubeContext;            /* OpenGL context */
GLuint     CubeFont;               /* Display list base for font */
/*
 * Functions...
 */
void        DisplayFunc(void);
static pascal OSStatus
```

```

EventHandler(EventHandlerCallRef nextHandler,
             EventRef event, void *userData);

void IdleFunc(void);
void MotionFunc(int x, int y);
void MouseFunc(int button, int state, int x, int y);
void ReshapeFunc(int width, int height);

/*
 * 'main()' - Main entry for example program.
 */
int main(int argc,           /* O - Exit status */
         char *argv[])  /* I - Number of command-line args */
{
    AGLPixelFormat      format;      /* OpenGL pixel format */
    WindowPtr           window;     /* Window */
    int                 winattrs;    /* Window attributes */
    Str255              title;      /* Title of window */
    Rect                rect;       /* Rectangle definition */
    EventHandlerUPP    handler;    /* Event handler */
    EventLoopTimerUPP  thandler;   /* Timer handler */
    EventLoopTimerRef  timer;     /* Timer for animating the window */
    ProcessSerialNumber psn;       /* Process serial number */
    short               font;       /* Font number */
    Str255              fontname;   /* Font name */
    static EventTypeSpec events[] = /* Events we are interested in... */
    {
        { kEventClassMouse, kEventMouseDown },
        { kEventClassMouse, kEventMouseUp },
        { kEventClassMouse, kEventMouseMoved },
        { kEventClassMouse, kEventMouseDragged },
        { kEventClassWindow, kEventWindowDrawContent },
        { kEventClassWindow, kEventWindowShown },
        { kEventClassWindow, kEventWindowHidden },
        { kEventClassWindow, kEventWindowActivated },
        { kEventClassWindow, kEventWindowDeactivated },
        { kEventClassWindow, kEventWindowClose },
        { kEventClassWindow, kEventWindowBoundsChanged }
    };
    static GLint          attributes[] = /* OpenGL attributes */
    {
        AGL_RGBA,
        AGL_GREEN_SIZE, 1,
        AGL_DOUBLEBUFFER,
        AGL_DEPTH_SIZE, 16,
        AGL_NONE
    };
    //Set initial values for window
    const int            origWinHeight = 628;
    const int            origWinWidth  = 850;
    const int            origWinXOffset = 50;
    const int            origWinYOffset = 50;
    /*
     * Create the window...
     */
    CubeContext = 0;
    CubeVisible = 0;
    SetRect(&rect, origWinXOffset, origWinYOffset, origWinWidth, origWinHeight);
    winattrs = kWindowStandardHandlerAttribute | kWindowCloseBoxAttribute |
               kWindowCollapseBoxAttribute | kWindowFullZoomAttribute |
               kWindowResizableAttribute | kWindowLiveResizeAttribute;
    winattrs &= GetAvailableWindowAttributes(kDocumentWindowClass);
    strcpy(title + 1, "Carbon OpenGL Example");
}

```

```

title[0] = strlen(title + 1);
CreateNewWindow(kDocumentWindowClass, winattrs, &rect, &window);
SetWTitle(window, title);
handler = NewEventHandlerUPP(EventHandler);
InstallWindowEventHandler(window, handler,
                           sizeof(events) / sizeof(events[0]),
                           events, NULL, 0L);
thandler =
    NewEventLoopTimerUPP((void (*)(EventLoopTimerRef, void *))IdleFunc);
InstallEventLoopTimer(GetMainEventLoop(), 0, 0, thandler,
                      0, &timer);
GetCurrentProcess(&psn);
SetFrontProcess(&psn);
DrawGrowIcon(window);
ShowWindow(window);

/*
 * Create the OpenGL context and bind it to the window...
 */
format      = aglChoosePixelFormat(NULL, 0, attributes);
CubeContext = aglCreateContext(format, NULL);
if (!CubeContext)
{
    puts("Unable to get OpenGL context!");
    return (1);
}
aglDestroyPixelFormat(format);
aglSetDrawable(CubeContext, GetWindowPort(window));
/*
 * Setup remaining globals...
 */
CubeX        = 50;
CubeY        = 50;
CubeWidth    = 400;
CubeHeight   = 400;
CubeRotation[0] = 45.0f;
CubeRotation[1] = 45.0f;
CubeRotation[2] = 45.0f;
CubeRate[0]   = 1.0f;
CubeRate[1]   = 1.0f;
CubeRate[2]   = 1.0f;
/*
 * Setup font...
 */
strcpy(fontname + 1, "Courier New");
fontname[0] = strlen(fontname + 1);
GetFNum(fontname, &font);
CubeFont = glGenLists(96);
aglUseFont(CubeContext, font, bold, 14, ' ', 96, CubeFont);
//Set the initial size of the cube
ReshapeFunc(origWinWidth - origWinXOffset, origWinHeight - origWinYOffset);
/*
 * Loop forever...
 */
for (;;)
{
    if (CubeVisible)
        SetEventLoopTimerNextFireTime(timer, 0.05);
    RunApplicationEventLoop();
    if (CubeVisible)
    {
        /*

```

```

        * Animate the cube...
        */
        DisplayFunc();
    }
}

/*
 * 'DisplayFunc()' - Draw a cube.
 */
void
DisplayFunc(void)
{
    int                  i, j;          /* Looping vars */
float            aspectRatio,windowWidth, windowHeight;
    static const GLfloat  corners[8][3] = /* Corner vertices */
    {
        { 1.0f, 1.0f, 1.0f }, /* Front top right */
        { 1.0f, -1.0f, 1.0f }, /* Front bottom right */
        { -1.0f, -1.0f, 1.0f }, /* Front bottom left */
        { -1.0f, 1.0f, 1.0f }, /* Front top left */
        { 1.0f, 1.0f, -1.0f }, /* Back top right */
        { 1.0f, -1.0f, -1.0f }, /* Back bottom right */
        { -1.0f, -1.0f, -1.0f }, /* Back bottom left */
        { -1.0f, 1.0f, -1.0f } /* Back top left */
    };
    static const int      sides[6][4] = /* Sides */
    {
        { 0, 1, 2, 3 },          /* Front */
        { 4, 5, 6, 7 },          /* Back */
        { 0, 1, 5, 4 },          /* Right */
        { 2, 3, 7, 6 },          /* Left */
        { 0, 3, 7, 4 },          /* Top */
        { 1, 2, 6, 5 }           /* Bottom */
    };
    static const GLfloat  colors[6][3] = /* Colors */
    {
        { 1.0f, 0.0f, 0.0f },    /* Red */
        { 0.0f, 1.0f, 0.0f },    /* Green */
        { 1.0f, 1.0f, 0.0f },    /* Yellow */
        { 0.0f, 0.0f, 1.0f },    /* Blue */
        { 1.0f, 0.0f, 1.0f },    /* Magenta */
        { 0.0f, 1.0f, 1.0f }     /* Cyan */
    };
/*
 * Set the current OpenGL context...
 */
aglSetCurrentContext(CubeContext);
/*
 * Clear the window...
 */
glViewport(0, 0, CubeWidth, CubeHeight);
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
/*
 * Setup the matrices...
 */
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
aspectRatio = (GLfloat)CubeWidth / (GLfloat)CubeHeight;
if(CubeWidth <= CubeHeight)
{

```

```

windowWidth = 2.0f;
windowHeight = 2.0f / aspectRatio;
glOrtho(-2.0f, 2.0f, -windowHeight, windowHeight, 2.0f, -2.0f);
}
else
{
    windowHeight = 2.0f * aspectRatio;
    windowHeight = 2.0f;
    glOrtho(-windowHeight, windowHeight, -2.0f, 2.0f, 2.0f, -2.0f);
}

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glRotatef(CubeRotation[0], 1.0f, 0.0f, 0.0f);
glRotatef(CubeRotation[1], 0.0f, 1.0f, 0.0f);
glRotatef(CubeRotation[2], 0.0f, 0.0f, 1.0f);
/*
 * Draw the cube...
 */
glEnable(GL_DEPTH_TEST);
glBegin(GL_QUADS);
for (i = 0; i < 6; i++)
{
    glColor3fv(colors[i]);
    for (j = 0; j < 4; j++)
        glVertex3fv(corners[sides[i][j]]);
}
glEnd();
/*
 * Draw lines coming out of the cube...
 */
glColor3f(1.0f, 1.0f, 1.0f);
glBegin(GL_LINES);
    glVertex3f(0.0f, 0.0f, -1.5f);
    glVertex3f(0.0f, 0.0f, 1.5f);
    glVertex3f(-1.5f, 0.0f, 0.0f);
    glVertex3f(1.5f, 0.0f, 0.0f);
    glVertex3f(0.0f, 1.5f, 0.0f);
    glVertex3f(0.0f, -1.5f, 0.0f);
glEnd();
/*
 * Draw text for each side...
 */

glPushAttrib(GL_LIST_BIT);
    glListBase(CubeFont - ' ');
    glRasterPos3f(0.0f, 0.0f, -1.5f);
    glCallLists(4, GL_BYTE, "Back");
    glRasterPos3f(0.0f, 0.0f, 1.5f);
    glCallLists(5, GL_BYTE, "Front");
    glRasterPos3f(-1.5f, 0.0f, 0.0f);
    glCallLists(4, GL_BYTE, "Left");
    glRasterPos3f(1.5f, 0.0f, 0.0f);
    glCallLists(5, GL_BYTE, "Right");
    glRasterPos3f(0.0f, 1.5f, 0.0f);
    glCallLists(3, GL_BYTE, "Top");
    glRasterPos3f(0.0f, -1.5f, 0.0f);
    glCallLists(6, GL_BYTE, "Bottom");
glPopAttrib();
/*
 * Swap the front and back buffers...
 */

```

```

    aglSwapBuffers(CubeContext);
}

/*
 * 'EventHandler()' - Handle window and mouse events from the window manager.
 */

static pascal OSStatus           /* O - noErr on success or error code */
EventHandler(EventHandlerCallRef nextHandler,      /* I - Next handler to call */
             EventRef                 /* I - Event reference */
             void                     /* I - User data (not used) */
{
    UInt32                 kind;           /* Kind of event */
    Rect                  rect;           /* New window size */
    EventMouseButton      button;        /* Mouse button */
    Point                 point;         /* Mouse position */
    kind = GetEventKind(event);
    if (GetEventClass(event) == kEventClassWindow)
    {
        switch (kind)
        {
            case kEventWindowDrawContent :
                if (CubeVisible && CubeContext)
                    DisplayFunc();
                break;
            case kEventWindowBoundsChanged :
                GetEventParameter(event, kEventParamCurrentBounds, typeQDRectangle,
                                  NULL, sizeof(Rect), NULL, &rect);
                CubeX = rect.left;
                CubeY = rect.top;
                if (CubeContext)
                    aglUpdateContext(CubeContext);
                ReshapeFunc(rect.right - rect.left, rect.bottom - rect.top);
                if (CubeVisible && CubeContext)
                    DisplayFunc();
                break;

            case kEventWindowShown :
                CubeVisible = 1;
                if (CubeContext)
                    DisplayFunc();
                break;
            case kEventWindowHidden :
                CubeVisible = 0;
                break;
            case kEventWindowClose :
                ExitToShell();
                break;
        }
    }
    else
    {
        switch (kind)
        {
            case kEventMouseDown :
                GetEventParameter(event, kEventParamMouseButton, typeMouseButton,
                                  NULL, sizeof(EventMouseButton), NULL, &button);
                GetEventParameter(event, kEventParamMouseLocation, typeQDPoint,
                                  NULL, sizeof(Point), NULL, &point);
                if (point.v < CubeY ||
                    (point.v > (CubeY + CubeHeight - 8) &&
                     point.h > (CubeX + CubeWidth - 8)))
                    return (CallNextEventHandler(nextHandler, event));
        }
    }
}

```

```

        MouseFunc(button, 0, point.h, point.v);
        break;
    case kEventMouseUp :
        GetEventParameter(event, kEventParamMouseButton, typeMouseButton,
                          NULL, sizeof(EventMouseButton), NULL, &button);
        GetEventParameter(event, kEventParamMouseLocation, typeQDPoint,
                          NULL, sizeof(Point), NULL, &point);

        if (point.v < CubeY ||
            (point.v > (CubeY + CubeHeight - 8) &&
             point.h > (CubeX + CubeWidth - 8)))
            return (CallNextEventHandler(nextHandler, event));
        MouseFunc(button, 1, point.h, point.v);
        break;
    case kEventMouseDragged :
        GetEventParameter(event, kEventParamMouseLocation, typeQDPoint,
                          NULL, sizeof(Point), NULL, &point);
        if (point.v < CubeY ||
            (point.v > (CubeY + CubeHeight - 8) &&
             point.h > (CubeX + CubeWidth - 8)))
            return (CallNextEventHandler(nextHandler, event));
        MotionFunc(point.h, point.v);
        break;
    default :
        return (CallNextEventHandler(nextHandler, event));
    }
}
/*
 * Return whether we handled the event...
 */
return (noErr);
}
/*
 * 'IdleFunc()' - Rotate and redraw the cube.
 */
void
IdleFunc(void)
{
    CubeRotation[0] += CubeRate[0];
    CubeRotation[1] += CubeRate[1];
    CubeRotation[2] += CubeRate[2];
    QuitApplicationEventLoop();
}
/*
 * 'MotionFunc()' - Handle mouse pointer motion.
 */
void
MotionFunc(int x,                      /* I - X position */
           int y)                      /* I - Y position */
{
/*
 * Get the mouse movement...
 */
x -= CubeMouseX;
y -= CubeMouseY;
/*
 * Update the cube rotation rate based upon the mouse movement and
 * button...
 */
switch (CubeMouseButton)
{

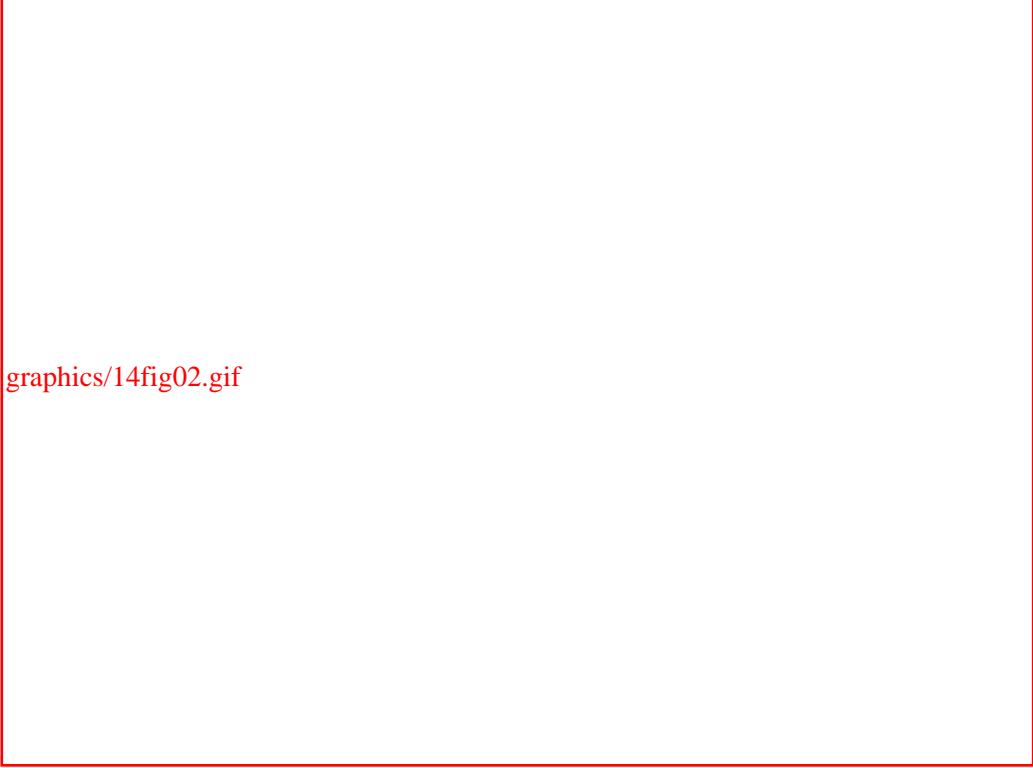
```

```

case 0 :                                /* Button 1 */
    CubeRate[0] = 0.01f * y;
    CubeRate[1] = 0.01f * x;
    CubeRate[2] = 0.0f;
    break;
case 1 :                                /* Button 2 */
    CubeRate[0] = 0.0f;
    CubeRate[1] = 0.01f * y;
    CubeRate[2] = 0.01f * x;
    break;
default :                                /* Button 3 */
    CubeRate[0] = 0.01f * y;
    CubeRate[1] = 0.0f;
    CubeRate[2] = 0.01f * x;
    break;
}
}
*/
/* 'MouseFunc()' - Handle mouse button press/release events.
 */
void
MouseFunc(int button,                  /* I - Button that was pressed */
          int state,                  /* I - Button state (0 = down) */
          int x,                      /* I - X position */
          int y)                      /* I - Y position */
{
/*
 * Only respond to button presses...
 */
if (state)
    return;
/*
 * Save the mouse state...
 */
CubeMouseButton = button;
CubeMouseX      = x;
CubeMouseY      = y;
/*
 * Zero-out the rotation rates...
 */
CubeRate[0] = 0.0f;
CubeRate[1] = 0.0f;
CubeRate[2] = 0.0f;
}
/*
 * 'ReshapeFunc()' - Resize the window.
 */
void
ReshapeFunc(int width,                /* I - Width of window */
            int height)               /* I - Height of window */
{
    CubeWidth  = width;
    CubeHeight = height;
}

```

Figure 14.2. The Carbon spinning cube with fonts.



graphics/14fig02.gif

Using the Cocoa API

The Cocoa API is suitable for applications developed using Objective C and the Cocoa user interface classes. Cocoa provides a single OpenGL rendering class that you must subclass to do OpenGL rendering.

Cocoa programs of this type require the ApplicationServices, Cocoa, and OpenGL frameworks; you can start with a Cocoa-based application in the project builder application and add the OpenGL framework or use the following linker options:

```
-framework Cocoa -framework OpenGL -framework ApplicationServices
```

The `NSOpenGL` Class

The `NSOpenGL` class provides the basis for all OpenGL-based Cocoa components. You must subclass this class to implement a Cocoa-based OpenGL display, implementing three required methods: `basicPixelFormat`, `drawRect`, and `initWithFrame`. The `basicPixelFormat` method is used to create a copy of an attribute array similar to that used by the `aglChoosePixelFormat` function. The following code shows a typical implementation that requests a double-buffered pixel format with at least 16 bits of depth buffer:

```
+ (NSOpenGLPixelFormat*) basicPixelFormat
{
    static NSOpenGLPixelFormatAttribute attributes[ ] =
    {
        NSOpenGLPFAWindow,
        NSOpenGLPFADoubleBuffer,
        NSOpenGLPFADepthSize, (NSOpenGLPixelFormatAttribute)16,
        (NSOpenGLPixelFormatAttribute)nil
    };
    return ([[NSOpenGLPixelFormat alloc] initWithAttributes] autorelease);
}
```

Each attribute value is of type `NSOpenGLPixelFormatAttribute` and the attribute list is terminated by a `nil` (zero) value. Table 14.3 lists the valid attribute values.

`NSOpenGLPFAAccelerated`

This constant specifies that a hardware-accelerated pixel format is required.

`NSOpenGLPFAAccumAlphaSize`

The number that follows specifies the minimum number of alpha bits that are required in the accumulation buffer.

`NSOpenGLPFAAccumBlueSize`

The number that follows specifies the minimum number of blue bits that are required in the accumulation buffer.

`NSOpenGLPFAAccumGreenSize`

The number that follows specifies the minimum number of green bits that are required in the accumulation buffer.

`NSOpenGLPFAAccumRedSize`

The number that follows specifies the minimum number of red bits that are required in the accumulation buffer.

`NSOpenGLPFAAlphaSize`

The number that follows specifies the minimum number of alpha bits that are required.

`NSOpenGLPFAAuxBuffers`

The number that follows specifies the number of auxiliary buffers that are required.

`NSOpenGLFABackingStore`

This constant specifies that only formats that support a full backing store capability should be used.

`NSOpenGLPFABlueSize`

The number that follows specifies the minimum number of blue bits that are required.

`NSOpenGLPFABufferSize`

The number that follows specifies the number of color index bits that are desired.

`NSOpenGLPFAClosestPolicy`

This constant specifies that the sizes specified should be used as the ideal requirements for the pixel format, and the pixel format should use the closest number of bits possible.

NSOpenGLPFADepthSize

The number that follows specifies the minimum number of depth bits that are required.

NSOpenGLPFADoubleBuffer

This constant specifies that a double-buffered visual is desired.

NSOpenGLPFAGreenSize

The number that follows specifies the minimum number of green bits that are required.

NSOpenGLPFALevel

This constant specifies the buffer level in the number that follows; 0 is the main buffer, 1 is the first overlay buffer, -1 is the first underlay buffer, and so forth.

NSOpenGLPFAMinimumPolicy

This constant specifies that the sizes specified should be used as the minimum requirements for the pixel format, and the pixel format should use the minimum number of bits possible.

NSOpenGLPFAMaximumPolicy

This constant specifies that the sizes specified should be used as the minimum requirements for the pixel format, and the pixel format should use the maximum number of bits possible.

NSOpenGLPFAOffScreen

This constant specifies that the pixel format is for an offscreen buffer.

NSOpenGLPFAPixelSize

The number that follows specifies the total number of bits that are used for a pixel.

NSOpenGLPFARedSize

The number that follows specifies the minimum number of red bits that are required.

NSOpenGLPFAStencilSize

The number that follows specifies the minimum number of stencil bits that are required.

NSOpenGLFAStereo

This constant specifies that a stereo visual is desired; stereo visuals provide separate left and right eye images.

Table 14.3.**NSOpenGLPixelFormatAttribute****Constants**

Constant	Description

The `drawRect` method does any OpenGL drawing calls to redraw the widget; the `rect` argument can be used to set the viewport and viewing transformation as necessary:

```
- (void)drawRect:(NSRect)rect
{
    int width, height;
    // Get the current bounding rectangle...
    width = rect.size.width;
    height = rect.size.height;
    // Set the viewport...
    glViewport(0, 0, width, height);
    // Setup the projection matrix...
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-2.0f, 2.0f,
            -2.0f * height / width, 2.0f * height / width,
            -2.0f, 2.0f);
    // Call any OpenGL functions to draw the scene...
    ...
}
```

Finally, the `initWithFrame` method initializes the OpenGL drawing area:

```
- (id)initWithFrame:(NSRect)frameRect
{
    NSOpenGLPixelFormat *pf;
    // Get the pixel format and return a new window from it...
    pf = [MyClass basicPixelFormat];
    self = [super initWithFrame: frameRect pixelFormat: pf];
    return (self);
}
```

The `NSOpenGL` class handles the creation of an OpenGL context for use with the pixel format that you have supplied.

Your First Cocoa Program

Now that we have covered the basics of using the `NSOpenGL` class for Cocoa applications, we will convert the Carbon Cube example to Cocoa. All the Carbon code is fully contained in a class called `Cube` that is based on `NSOpenGL` and implements all the required methods described in the previous section. It also implements the mouse methods so that you can click and drag the mouse to rotate the cube. Listing 14.3 shows the completed Cocoa application that displays a spinning cube. The results are shown in Figure 14.3.

Listing 14.3. The `cocoa.m` Sample Program

```
#import <Cocoa/Cocoa.h>
#import <Carbon/Carbon.h>
#import <OpenGL/gl.h>
#import <OpenGL/glext.h>
#import <OpenGL/glu.h>
// Interface definition for our NIB CustomView
```

```

// In the NIB builder, derive as subclass named
// Cube from NSOpenGLView. Then assign this to a
// customview that fills the window.
@interface Cube : NSOpenGLView
{
    bool         initialized;
    NSTimer      *timer;
    float        rotation[3],
                 rate[3];
    int          mouse_x,
                 mouse_y;
    GLuint       font;
}
@end
//
// 'main()' - Main entry for program.
//

int
main(int argc,                                     // O - Exit status
     const char *argv[])                         // I - Number of command-line args
{
    return (NSApplicationMain(argc, argv));      // I - Command-line arguments
}
//
// Cube class based upon NSOpenGLView
//
@implementation Cube : NSOpenGLView
{
    bool         initialized;                  // Are we initialized?
    NSTimer      *timer;                     // Timer for animation
    float        rotation[3],                // Rotation of cube
                 rate[3];                   // Rotation rate of cube
    int          mouse_x,                   // Start X position of mouse
                 mouse_y;                   // Start Y position of mouse
}
//
// 'basicPixelFormat()' - Set the pixel format for the window.
//
+ (NSOpenGLPixelFormat*) basicPixelFormat
{
    static NSOpenGLPixelFormatAttribute attributes[] = // OpenGL attributes
    {
        NSOpenGLPFAWindow,
        NSOpenGLPFADoubleBuffer,
        NSOpenGLPFADepthSize, (NSOpenGLPixelFormatAttribute)16,
                                         (NSOpenGLPixelFormatAttribute)nil
    };
    return ([[NSOpenGLPixelFormat alloc] initWithAttributes:attributes]autorelease);
}
//
// 'resizeGL()' - Resize the window.
//
- (void) resizeGL
{
}
//
// 'idle()' - Update the display using the current rotation rates...
//
- (void)idle:(NSTimer *)timer
{
}

```

```

// Rotate...
rotation[0] += rate[0];
rotation[1] += rate[1];
rotation[2] += rate[2];
// Redraw the window...
[self drawRect:[self bounds]];
}

// 
// 'mouseDown()' - Handle left mouse button presses...
//
- (void)mouseDown:(NSEvent *)theEvent
{
    NSPoint      point;           // Mouse position
    // Get and save the mouse position
    point    = [self convertPoint:[theEvent locationInWindow] fromView:nil];
    mouse_x = point.x;
    mouse_y = point.y;
    // Null the rotation rates...
    rate[0] = 0.0f;
    rate[1] = 0.0f;
    rate[2] = 0.0f;
}
// 
// 'rightMouseDown()' - Handle right mouse button presses...
//
- (void)rightMouseDown:(NSEvent *)theEvent
{
    NSPoint      point;           // Mouse position
    // Get and save the mouse position
    point    = [self convertPoint:[theEvent locationInWindow] fromView:nil];
    mouse_x = point.x;
    mouse_y = point.y;

    // Null the rotation rates...
    rate[0] = 0.0f;
    rate[1] = 0.0f;
    rate[2] = 0.0f;
}
// 
// 'otherMouseDown()' - Handle middle mouse button presses...
//
- (void)otherMouseDown:(NSEvent *)theEvent
{
    NSPoint      point;           // Mouse position
    // Get and save the mouse position
    point    = [self convertPoint:[theEvent locationInWindow] fromView:nil];
    mouse_x = point.x;
    mouse_y = point.y;
    // Null the rotation rates...
    rate[0] = 0.0f;
    rate[1] = 0.0f;
    rate[2] = 0.0f;
}
// 
// 'mouseDragged()' - Handle drags using the left mouse button.
//
- (void)mouseDragged:(NSEvent *)theEvent
{
    NSPoint      point;           // Mouse position
    // Get the mouse position and update the rotation rates...
    point    = [self convertPoint:[theEvent locationInWindow] fromView:nil];
    rate[0] = 0.01f * (mouse_y - point.y);
}

```

```

    rate[1] = 0.01f * (mouse_x - point.x);
}

// 
// 'rightMouseDragged()' - Handle drags using the right mouse button.
// 
- (void)rightMouseDragged:(NSEvent *)theEvent
{
    NSPoint      point;           // Mouse position
    // Get the mouse position and update the rotation rates...
    point      = [self convertPoint:[theEvent locationInWindow] fromView:nil];
    rate[0] = 0.01f * (mouse_y - point.y);
    rate[2] = 0.01f * (mouse_x - point.x);
}
// 
// 'otherMouseDragged()' - Handle drags using the middle mouse button.
// 
- (void)otherMouseDragged:(NSEvent *)theEvent
{
    NSPoint      point;           // Mouse position
    // Get the mouse position and update the rotation rates...
    point      = [self convertPoint:[theEvent locationInWindow] fromView:nil];
    rate[1] = 0.01f * (mouse_y - point.y);
    rate[2] = 0.01f * (mouse_x - point.x);
}
- (void)drawRect:(NSRect)rect
{
    int          width,           // Width of window
                height;          // Height of window
    int          i, j;           // Looping vars
    float        aspectRatio,    windowWidth, windowHeight;
    static const GLfloat  corners[8][3] = // Corner vertices
    {
        { 1.0f,  1.0f,  1.0f },      // Front top right
        { 1.0f, -1.0f,  1.0f },      // Front bottom right
        { -1.0f, -1.0f,  1.0f },     // Front bottom left
        { -1.0f,  1.0f,  1.0f },      // Front top left
        { 1.0f,  1.0f, -1.0f },      // Back top right
        { 1.0f, -1.0f, -1.0f },      // Back bottom right
        { -1.0f, -1.0f, -1.0f },     // Back bottom left
        { -1.0f,  1.0f, -1.0f }       // Back top left
    };
    static const int    sides[6][4] = // Sides
    {
        { 0, 1, 2, 3 },           // Front
        { 4, 5, 6, 7 },           // Back
        { 0, 1, 5, 4 },           // Right
        { 2, 3, 7, 6 },           // Left
        { 0, 3, 7, 4 },           // Top
        { 1, 2, 6, 5 }            // Bottom
    };
    static const GLfloat colors[6][3] = // Colors
    {
        { 1.0f, 0.0f, 0.0f },      // Red
        { 0.0f, 1.0f, 0.0f },      // Green
        { 1.0f, 1.0f, 0.0f },      // Yellow
        { 0.0f, 0.0f, 1.0f },      // Blue
        { 1.0f, 0.0f, 1.0f },      // Magenta
        { 0.0f, 1.0f, 1.0f }       // Cyan
    };
    // Set the current OpenGL context...
}

```

```

if (!initialized)
{
    rotation[0] = 45.0f;
    rotation[1] = 45.0f;
    rotation[2] = 45.0f;
    rate[0]     = 1.0f;
    rate[1]     = 1.0f;
    rate[2]     = 1.0f;
    initialized = true;
}
// Use the current bounding rectangle for the cube window...
width = rect.size.width;
height = rect.size.height;
// Clear the window...
glViewport(0, 0, width, height);
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// Setup the matrices...
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
aspectRatio = (GLfloat)width / (GLfloat)height;
if(width <= height)
{
    windowHeight = 2.0f;
    windowHeight = 2.0f / aspectRatio;
    glOrtho(-2.0f, 2.0f, -windowHeight, windowHeight, 2.0f, -2.0f);
}
else
{
    windowHeight = 2.0f * aspectRatio;
    windowHeight = 2.0f;
    glOrtho(-windowHeight, windowHeight, -2.0f, 2.0f, 2.0f, -2.0f);
}
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glRotatef(rotation[0], 1.0f, 0.0f, 0.0f);
glRotatef(rotation[1], 0.0f, 1.0f, 0.0f);
glRotatef(rotation[2], 0.0f, 0.0f, 1.0f);
// Draw the cube...
glEnable(GL_DEPTH_TEST);
glBegin(GL_QUADS);
for (i = 0; i < 6; i++)
{
    glColor3fv(colors[i]);
    for (j = 0; j < 4; j++)
        glVertex3fv(corners[sides[i][j]]);
}
glEnd();
// Draw lines coming out of the cube...
glColor3f(1.0f, 1.0f, 1.0f);
glBegin(GL_LINES);
    glVertex3f(0.0f, 0.0f, -1.5f);
    glVertex3f(0.0f, 0.0f, 1.5f);
    glVertex3f(-1.5f, 0.0f, 0.0f);
    glVertex3f(1.5f, 0.0f, 0.0f);
    glVertex3f(0.0f, 1.5f, 0.0f);
    glVertex3f(0.0f, -1.5f, 0.0f);
glEnd();
// Swap the front and back buffers...
[[self openGLContext]flushBuffer];
}

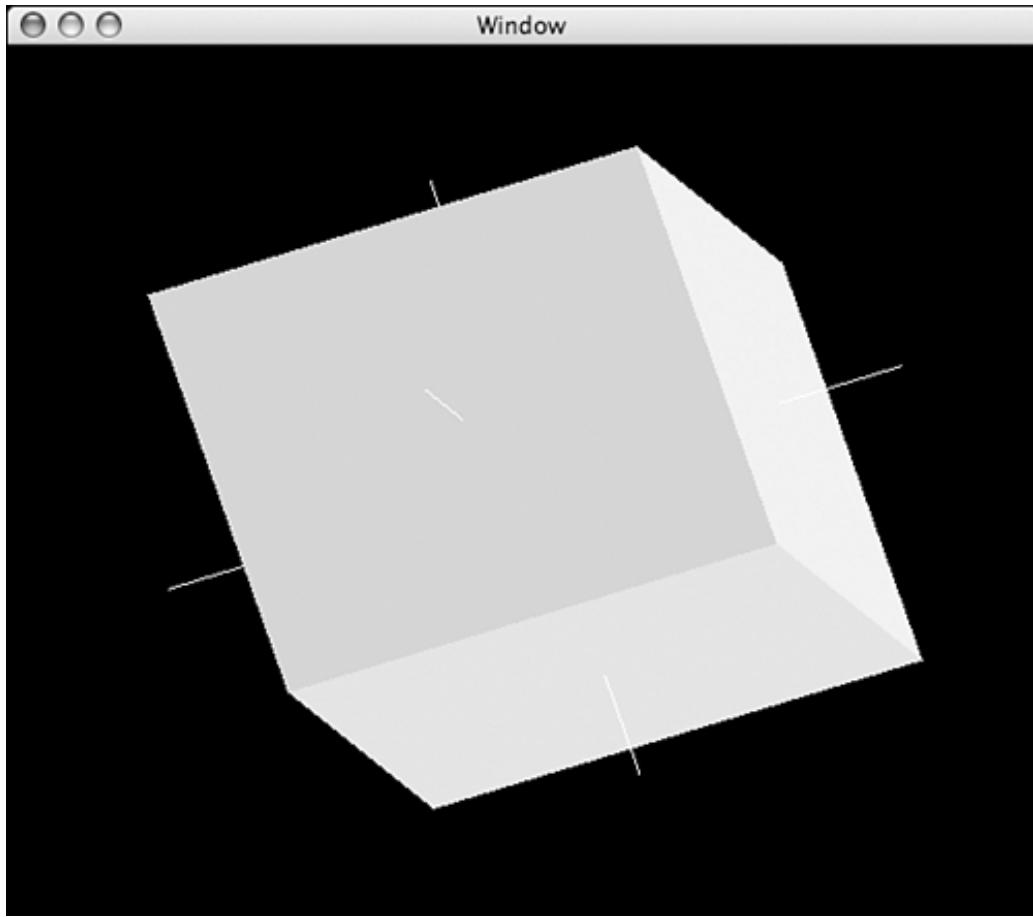
```

```

// 
// 'acceptsFirstResponder()' ...
// 
- (BOOL)acceptsFirstResponder
{
    return (YES);
}
- (BOOL) becomeFirstResponder
{
    return (YES);
}
- (BOOL) resignFirstResponder
{
    return (YES);
}
// 
// 'initWithFrame()' - Initialize the cube.
// 
- (id)initWithFrame:(NSRect)frameRect
{
    NSOpenGLPixelFormat      *pf;
    // Get the pixel format and return a new cube window from it...
    pf    = [Cube basicPixelFormat];
    self = [super initWithFrame: frameRect pixelFormat: pf];
    return (self);
}
// 
// 'awakeFromNib()' - Do stuff once the UI is loaded from the NIB file...
// 
- (void)awakeFromNib
{
    // Set initial values...
    initialized = false;
    //start cube rotating
    [self drawRect:[self bounds]];
    // Start the timer running...
    timer = [NSTimer timerWithTimeInterval:(0.05f) target:self
                                         selector:@selector(idle:) userInfo:nil repeats:YES];
    [[NSRunLoop currentRunLoop]addTimer:timer forMode:NSDefaultRunLoopMode];
    [[NSRunLoop currentRunLoop]addTimer:timer forMode:NSEventTrackingRunLoopMode];
}
@end

```

Figure 14.3. The COCOACUBE program.



Summary

OpenGL is well supported on Mac OS X and can be accessed using several different APIs provided by Apple: AGL, CGL, Cocoa, and GLUT. The AGL API provides OpenGL support for Carbon-based applications and is supported on Mac OS 8 through X, the CGL API provides full-screen OpenGL support and is meant for use with games and other full-screen applications, the Cocoa API is an Objective C interface for complex GUIs that include OpenGL-based components, and the GLUT API provides a simple, cross-platform interface for the sample programs in this book as well as for simple applications.

Reference

aglChoosePixelFormat

Purpose: Enables you to choose a pixel format for OpenGL rendering.

Include File: <AGL/agl.h>

Syntax:

```
AGLPixelFormat aglChoosePixelFormat(const
→ AGLDevice *gdevs, GLint ndev, const GLint *attribs);
```

Description: This function locates a compatible pixel format for all the listed devices. If *ndev* is 0, a pixel format is chosen that is compatible with all display devices.

Parameters:

***gdevs** `const AGLDevice`: An array of graphics devices.

ndev `GLint`: The number of graphics devices.

***attribs** `const GLint`: An array of integer attributes terminated by the `AGL_NONE` constant.

Returns: A compatible pixel format or `NULL` if no pixel format satisfies the specified attributes.

See Also: [aglCreateContext](#)

aglCreateContext

Purpose: Creates an OpenGL context for rendering.

Include File: `<AGL/agl.h>`

Syntax:

```
AGLContext aglCreateContext(AGLPixelFormat pix,
                           AGLContext share);
```

Description: This function creates a new OpenGL rendering context. If the `share` argument is not `NULL`, the new context will share display lists, texture objects, and vertex arrays with the specified context.

Parameters:

`pix` `AGLPixelFormat`: The pixel format to use, as returned by `aglChoosePixelFormat`.

`share` `AGLContext`: The OpenGL context to share with or `NULL`.

Returns: A new OpenGL context or `NULL` if it could not be created.

See Also: [aglChoosePixelFormat](#), [aglDestroyContext](#), [aglSetCurrentContext](#), [aglSetDrawable](#)

aglDestroyContext

Purpose: To destroy an OpenGL rendering context.

Include File: `<AGL/agl.h>`

Syntax:

```
GLboolean aglDestroyContext(AGLContext ctx);
```

Description: This function destroys the specified OpenGL context.

Parameters:

`ctx` `AGLContext`: The OpenGL context to destroy.

Returns: `GL_FALSE` if the context could not be destroyed; `GL_TRUE` otherwise.

See Also: [aglCreateContext](#)

aglSetCurrentContext

Purpose: Sets the current context for OpenGL rendering.

Include File: [<AGL/agl.h>](#)

Syntax:

```
GLboolean aglSetCurrentContext(AGLContext ctx);
```

Description: This function sets the current OpenGL context for rendering.

Parameters:

ctx [AGLContext](#): The OpenGL context to use.

Returns: [GL_FALSE](#) if the context could not be used; [GL_TRUE](#) otherwise.

See Also: [aglCreateContext](#), [aglSetDrawable](#)

aglSetDrawable

Purpose: Sets the window or offscreen buffer associated with a context.

Include File: [<AGL/agl.h>](#)

Syntax:

```
GLboolean aglSetDrawable(AGLContext ctx,
    ↪ AGLDrawable draw);
```

Description: This function binds a window or offscreen buffer to an OpenGL context. You must call this function after creating the context and before using the context to do any rendering.

Parameters:

ctx [AGLContext](#): The OpenGL context to bind.

draw [AGLDrawable](#): The window or offscreen buffer to use.

Returns: [GL_FALSE](#) if the drawable could not be bound; [GL_TRUE](#) otherwise.

See Also: [aglCreateContext](#), [aglSetCurrentContext](#)

AgISwapBuffers

Purpose: Swaps the front and back buffers in an OpenGL window.

Include File: [<AGL/agl.h>](#)

Syntax:

```
void aglSwapBuffers(AGLContext ctx);
```

Description: This function swaps the front and back buffers of the double-buffered OpenGL window bound to the specified context. You typically call this function after drawing a scene or frame using OpenGL functions.

Parameters:

ctx **AGLContext**: The OpenGL context to swap.

Returns: Nothing

See Also: [AglCreateContext](#)

AglUseFont

Purpose: Creates a collection of bitmap display lists.

Include File: `<AGL/agl.h>`

Syntax:

```
GLboolean aglUseFont(AGLContext ctx, GLint fontID,
➥ Style face, GLint size, int first, int count, int
➥ base);
```

Description: This function creates *count* display lists containing bitmaps of characters in the specified font. You allocate the display lists for the bitmaps using the [glGenLists](#) function.

Parameters:

ctx **AGLContext**: Specifies the current rendering context.

fontID **GLint**: Specifies the font to use.

face **Style**: Specifies the font style to use.

size **GLint**: Specifies the size of font to use.

first **int**: Specifies the first character in the font to use.

count **int**: Specifies the number of characters to use from the font.

base **int**: Specifies the first display list to use as returned by [glGenLists](#)

Returns: **GL_TRUE** if successful; **GL_FALSE** otherwise

See Also: [AglCreateContext](#)

Chapter 15. GLX: OpenGL on Linux

by Nick Haemel

WHAT YOU'LL LEARN IN THIS CHAPTER:

How To	Functions You'll Use
Choose appropriate visuals for	<code>glXChooseVisual, glXChooseFBConfig</code>
Manage OpenGL drawing contexts	<code>glXCreateContext, glXDestroyContext, glXMakeCurrent</code>
Create OpenGL windows	<code>glXCreateWindow</code>
Do double-buffered drawing	<code>glXSwapBuffers</code>
Create bitmap text fonts	<code>glXUseXFont</code>
Do offscreen drawing	<code>glXCreateGLXPixmap, glXCreatePbuffer</code>

This chapter discusses GLX, the OpenGL extension that is used to support OpenGL applications through the X Window System on UNIX and Linux. You learn how to create and manage OpenGL contexts as well as how to create OpenGL drawing areas with several of the common GUI toolkits. You also learn how to use GLUT with all the examples in this book.

The Basics

OpenGL on UNIX and Linux uses the OpenGL extension to the X Window System called GLX. All GLX-specific functions start with the prefix `glX` and are generally included in the GL library. The GLX functions are the "glue" binding OpenGL, X11, and the graphics hardware that provides accelerated rendering.

Using the OpenGL and X11 Libraries

The location of the OpenGL and X11 libraries and header files varies from system to system. Some are in the standard include and library locations, such as `/usr/include` and `/usr/lib`, that require only the libraries with the link command:

```
gcc -o myprogram myprogram.o -lGLU -lGL -lXext -lX11 -lm
```

Others use a version-specific location, such as `/usr/X11R6/include` or `/usr/X11R6/lib`, that requires both compiler and linker options:

```
gcc -I/usr/X11R6/include -c myprogram.c
gcc -o myprogram myprogram.o -L/usr/X11R6/lib -lGLU -lGL -lXext -lX11 -lm
```

You can hard-code the compiler and linker options if you are writing an application for a single platform; however, most OpenGL applications see life on many different platforms, so you might want to detect the appropriate options prior to doing a build. One common method is to use the GNU `autoconf` software to create a `configure` script that initializes your options for you. [Listing 15.1](#) provides a sample `configure.in` file for use with `autoconf`.

Listing 15.1. Sample `configure.in` File for GNU `autoconf`

```
dnl Required file in package...
```

```

AC_INIT(myprogram.c)
dnl Find the C compiler...
AC_PROG_CC
dnl Find the X Window System...
AC_PATH_XTRA
LIBS="$LIBS -lXext -lX11 $X_EXTRA_LIBS"
CFLAGS="$CFLAGS $X_CFLAGS"
LDFLAGS="$X_LIBS $LDFLAGS"
if test "x$x_includes" != x; then
    ac_cpp="$ac_cpp -I$x_includes"
fi
dnl OpenGL uses the math functions...
AC_SEARCH_LIBS(pow, m)
dnl Some OpenGL implementations use dlopen()...
AC_SEARCH_LIBS(dlopen, dl)
dnl Look for the OpenGL or Mesa libraries...
AC_CHECK_HEADER(GL/gl.h)
AC_CHECK_HEADER(GL/glu.h)
AC_CHECK_HEADER(GL/glx.h)
AC_CHECK_LIB(GL, glXMakeCurrent,
    LIBS="-lGLU -lGL $LIBS",
    AC_CHECK_LIB(MesaGL, glXMakeCurrent,
        LIBS="-lMesaGLU -lMesaGL $LIBS"))
dnl Generate a Makefile for the program
AC_OUTPUT(Makefile)

```

You build the actual `configure` script using the following command:

```
autoconf
```

The `AC_PATH_XTRA` and `AC_CHECK_LIB` macros handle searching for the X11 and OpenGL libraries, and the `AC_OUTPUT` macro generates any files using what the script found. The `Makefile.in` file used by the `configure` script is shown in [Listing 15.2](#).

Listing 15.2. Makefile Template File `Makefile.in`

```

#
# Sample makefile template for configure script.
#
#
# Compiler and options...
#
CC      =      @CC@
CFLAGS   =      @CFLAGS@
LDFLAGS  =      @LDFLAGS@
LIBS    =      @LIBS@
#
# Program
#
myprogram:    myprogram.o
    $(CC) $(LDFLAGS) -o myprogram myprogram.o $(LIBS)

```

The `configure` script substitutes the variables specified in `Makefile.in` to produce the final makefile to build your application. You specify variables using an at sign (@) before and after the variable name, such as `@CC@` for the C compiler, `@CFLAGS@` for the compiler options, and so forth.

You run the `configure` script to generate the makefile and then use the `make` command to build

your program, as follows:

```
./configure
make
```

You can use this same technique to create include files for makefiles to include common compiler and linker options for larger, multiple-directory projects, or you can list multiple makefile templates in the `AC_OUTPUT` macro.

A variation of the `configure.in` and `Makefile.in` files is used to build all the examples in this book on UNIX and Linux.

Using the GLUT Library

The GLUT library is not generally provided as a standard part of UNIX or Linux distributions; however, it is available as source code and can be compiled and installed relatively easily from the source that comes on the CD-ROM. Start by copying the `glut-3.7` directory to your own hard drive. Then run the following commands:

```
cd glut-3.7
./mkmkfiles.imeake
make
make install
```

After you install GLUT, simply add the GLUT library to your link command:

```
gcc -o myprogram myprogram.o -lglut -lGLU -lGL -lXext -lX11 -lm
```

Or add it to your makefile template:

```
LIBS      =      -lglut @LIBS@
```

OpenGL on Linux

Although most commercial versions of UNIX provide OpenGL support out of the box, OpenGL support on Linux depends greatly on the graphics card and Linux distribution you use. Also, there are both free and commercial versions of OpenGL that you can use.

The Xfree86 project provides the most popular free implementation of the X Window System and is provided with every Linux distribution. Xfree86 4.x uses the Direct Rendering Infrastructure (DRI) to provide accelerated OpenGL rendering. The following URLs provide information on each of these projects:

<http://www.xfree86.org/>

<http://dri.sf.net/>

OpenGL support is enabled through the `XF86Config` or `XF86Config-4` file, usually installed in `/etc/X11`. The key area of this file is the Module section, which follows:

```
Section "Module"
Load  "GLcore"      # OpenGL support
Load  "glx"         # OpenGL X protocol interface
Load  "dri"         # Direct rendering infrastructure
```

The `Load` lines load the named modules—`GLcore`, `glx`, and `dri`—and those modules provide OpenGL support. If the underlying driver supports OpenGL, your output will be hardware

accelerated. Otherwise, software emulation is used.

Commercial implementations of the X Window System are also available. One popular package that provides strong OpenGL support is Summit from Xi Graphics, available at the following URL:

<http://www.xi-graphics.com/>

OpenGL Emulation: Mesa

The Mesa library can be used on systems that don't support OpenGL natively, when you want to experiment with new OpenGL features or extensions that are not supported by your graphics card, or when you want to do offline rendering using OpenGL. You can find the Mesa library at the following URL:

<http://mesa3d.sf.net/>

Aside from its use as a standalone library, Mesa also serves as the core of the Xfree86 implementation of OpenGL.

The OpenGL Extension for the X Window System

The OpenGL extension for the X Window System, GLX, provides the interface between your application, the X Window System, and the graphics driver to provide accelerated rendering. If the underlying graphics hardware does not support a particular feature, it is automatically emulated in software. You can check whether your X server supports this extension by using the `xdpyinfo` program on the command line:

```
xdpyinfo | grep GLX
```

There have been five versions of the GLX extension to date (1.0, 1.1, 1.2, 1.3, and 1.4); by far the most common version in use is 1.2, but this chapter also covers the Pbuffer functionality available in GLX 1.3.

GLX provides several functions that manage visuals, contexts, and drawing surfaces.

X Window System Basics

The X Window System provides a network-transparent interface for creating windows, drawing things on the screen, getting input from the user, and so forth. The program that manages one or more screens, keyboard, mouse, and other assorted input devices is usually called an *X server*. The connection to the server is managed by a `Display` pointer. You connect to the server using the `XOpenDisplay()` function and the display name:

```
Display *display;
display = XOpenDisplay(getenv("DISPLAY"));
```

The default display name is, by convention, provided in the `DISPLAY` environment variable. When you have a display connection, you can create windows, draw graphics, and collect input from the user.

Choosing a Visual

Each window on the screen has a visual associated with it. Each visual has an associated class—`DirectGray`, `DirectColor`, `PseudoColor`, or `TrueColor`—that defines its properties. In general, most OpenGL rendering uses `TrueColor` visuals, which support arbitrary red, green, and blue color values. The other visual types map color numbers or indices to specific RGB values, making them useful only for specialized applications.

The `glXChooseVisual()` function determines the correct visual to use for a specific screen and combination of OpenGL features. The function accepts a `Display` pointer, screen number (usually 0), and attribute list. The attribute list is an array of integers. Each attribute consists of a token name (for example, `GLX_RED_SIZE`) and, for some attributes, a value. [Table 15.1](#) lists the attribute list tokens that are defined.

Table 15.1. GLX Visual Attribute List Constants

Constant	Description
<code>GLX_ACCUM_ALPHA_SIZE</code>	The number that follows specifies the minimum number of alpha bits that are required in the accumulation buffer.
<code>GLX_ACCUM_BLUE_SIZE</code>	The number that follows specifies the minimum number of blue bits that are required in the accumulation buffer.
<code>GLX_ACCUM_GREEN_SIZE</code>	The number that follows specifies the minimum number of green bits that are required in the accumulation buffer.
<code>GLX_ACCUM_RED_SIZE</code>	The number that follows specifies the minimum number of red bits that are required in the accumulation buffer.
<code>GLX_ALPHA_SIZE</code>	The number that follows specifies the minimum number of alpha bits that are required.
<code>GLX_AUX_BUFFERS</code>	The number that follows specifies the number of auxiliary buffers that are required.
<code>GLX_BLUE_SIZE</code>	The number that follows specifies the minimum number of blue bits that are required.
<code>GLX_BUFFER_SIZE</code>	The number that follows specifies the number of color index bits that are desired.
<code>GLX_DEPTH_SIZE</code>	The number that follows specifies the minimum number of depth bits that are required.
<code>GLX_DOUBLEBUFFER</code>	This token specifies that a double-buffered visual is desired.
<code>GLX_GREEN_SIZE</code>	The number that follows specifies the minimum number of green bits that are required.
<code>GLX_LEVEL</code>	This token specifies the buffer level in the number that follows; 0 is the main buffer, 1 is the first overlay buffer, -1 is the first underlay buffer, and so forth.
<code>GLX_RED_SIZE</code>	The number that follows specifies the minimum number of red bits that are required.
<code>GLX_RGBA</code>	This token specifies that an RGBA visual is desired.
<code>GLX_STENCIL_SIZE</code>	The number that follows specifies the minimum number of stencil bits that are required.
<code>GLX_STEREO</code>	This token specifies that a stereo visual is desired; stereo visuals provide separate left and right eye images.
<code>GLX_USE_GL</code>	This token specifies that OpenGL visuals are desired. This attribute is ignored because <code>glXChooseVisual()</code> returns only OpenGL visuals.

You might use the following code to find a double-buffered RGB visual:

```

int attributes[] = {
    GLX_RGBA,
    GLX_DOUBLEBUFFER,
    GLX_RED_SIZE, 4,
    GLX_GREEN_SIZE, 4,
    GLX_BLUE_SIZE, 4,
    GLX_DEPTH_SIZE, 16,
    0
};
Display *display;
XVisualInfo *vinfo;
vinfo = glXChooseVisual(display, DefaultScreen(display), attributes);

```

An `XVisualInfo` pointer is returned; it provides information on the correct visual to use. If no matching visual can be found, a `NULL` pointer is returned, and you should retry with different attributes or inform the users that the window cannot be displayed on their system.

When you have found the correct visual to use, you can create an OpenGL context to use with a window or pixmap.

Managing OpenGL Contexts

OpenGL contexts are used to manage drawing in a single window or pixmap. Contexts are identified using the `GLXContext` data type. The `glXCreateContext()` function creates a new context and accepts the `Display` pointer, an `XVisualInfo` pointer, a `GLXContext` to share with, and a boolean value indicating whether to create a direct or indirect context:

```

GLXContext context;
context = glXCreateContext(display, vinfo, 0, True);

```

The display and visual information pointers are as initialized by the previous code examples. The third argument specifies a `GLXContext` with which to share display lists, texture objects, and other stored data. You'll likely specify this argument when creating multiple OpenGL windows for your application. If you have no other contexts defined, pass a value of 0 for the context.

The fourth argument specifies whether a direct (`True`) or indirect (`False`) context is created. Direct contexts allow the OpenGL library to talk directly to the graphics hardware, providing the highest speed rendering. Indirect contexts send OpenGL drawing commands through the X server allowing for remote display over a network. Direct contexts cannot share with indirect contexts, and vice versa. This has the greatest impact on some types of offscreen rendering. Normally, you create a direct context.

When you are finished with an OpenGL context, call the `glXDestroyContext()` function to free the resources it uses:

```
glXDestroyContext(display, context);
```

Creating an OpenGL Window

When you have a `Display` pointer and `GLXContext`, you can use the `XCreateWindow()` function to create a window:

```

Window window;
XSetWindowAttributes winattrs;
winattrs.event_mask = ExposureMask | VisibilityChangeMask |
    StructureNotifyMask | ButtonPressMask |
    ButtonReleaseMask | PointerMotionMask;

```

```

winattrs.border_pixel = 0;
winattrs.bit_gravity = StaticGravity;
winmask              = CWBorderPixel | CWBitGravity | CWEEventMask;
window               = XCreateWindow(display, DefaultRootWindow(display),
                                     0, 0, 400, 400, 0, vinfo->depth, InputOutput,
                                     vinfo->visual, winmask, &winattrs);

```

After you create the window, you can show it using the `XMapWindow()` function and bind the OpenGL context to the window using the `glXMakeCurrent()` function:

```

XMapWindow(display, window);
glXMakeCurrent(display, window, context);

```

If you have multiple OpenGL windows, you need to call the `glXMakeCurrent()` function before drawing in each window. Otherwise, you can call it once for the life of the program.

Before you can actually draw in the window, you must wait for a `MapNotify` event, which tells your application that the X server has displayed your window on the screen. You can use the `XNextEvent()` function to wait for a `MapNotify` event or track whether you have seen the event in the normal event loop in your application.

Double-Buffered Windows

You create double-buffered windows by using a double-buffered visual. You get this type of visual by using the `glXChooseVisual()` function with the `GLX_DOUBLEBUFFER` attribute.

After the window has been created, the `glXSwapBuffers()` function will swap the front and back buffer for the window:

```
glXSwapBuffers(display, window);
```

The `glXSwapBuffers()` function flushes any pending OpenGL drawing commands and may block until the next vertical retrace.

Putting It All Together

Figure 15.1 shows an OpenGL application written entirely using the Xlib and GLX functions covered in this section; this application displays a spinning cube. The user can click any mouse button to stop the rotation or drag the mouse to rotate the cube in any axis. The program responds to `ConfigureNotify` events to track when the window is resized and `MapNotify` and `UnmapNotify` events when the window is shown or iconified. Listing 15.3 shows the source for this application.

Listing 15.3. The `xlib.c` Sample Program

```

/*
 * Include necessary headers...
 */
#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>
#include <X11/Xatom.h>
#include <GL/glx.h>
#include <GL/gl.h>
#include <sys/select.h>
#include <sys/types.h>
#include <sys/time.h>
/*

```

```

/* Globals...
*/
float      CubeRotation[3],      /* Rotation of cube */
           CubeRate[3];      /* Rotation rate of cube */
int       CubeMouseButton,      /* Button that was pressed */
          CubeMouseX,      /* Start X position of mouse */
          CubeMouseY;      /* Start Y position of mouse */
int       CubeWidth,          /* Width of window */
          CubeHeight;      /* Height of window */

/*
 * Functions...
 */
void      DisplayFunc(void);
void      IdleFunc(void);
void      MotionFunc(int x, int y);
void      MouseFunc(int button, int state, int x, int y);
void      ReshapeFunc(int width, int height);
/*
 * 'main()' - Main entry for example program.
 */
int      /* O - Exit status */
main(int  argc,           /* I - Number of command-line args */
      char *argv[]);      /* I - Command-line args */
{
    Bool      mapped;      /* Is the window mapped? */
    GLXContext context;    /* OpenGL context */
    Display   *display;    /* X display connection */
    Window    window;      /* X window */
    XVisualInfo *vinfo;    /* X visual information */
    XSetWindowAttributes winattrs; /* Window attributes */
    int       winmask;     /* Mask for attributes */
    XEvent    event;       /* Event data */
    XWindowAttributes windata; /* Window data */
    struct timeval timeout; /* Timeout interval for select() */
    fd_set    input;       /* Input set for select() */
    int       ready;       /* Event ready? */
    static int attributes[] = /* OpenGL attributes */
    {
        GLX_RGBA,
        GLX_DOUBLEBUFFER,
        GLX_RED_SIZE, 8,
        GLX_GREEN_SIZE, 8,
        GLX_BLUE_SIZE, 8,
        GLX_DEPTH_SIZE, 16,
        0
    };
}

/*
 * Open a connection to the X server...
 */
display = XOpenDisplay(getenv("DISPLAY"));
/*
 * Find the proper visual for an OpenGL window...
 */
vinfo = glXChooseVisual(display, DefaultScreen(display), attributes);
/*
 * Create the window...
 */
winattrs.event_mask = ExposureMask | VisibilityChangeMask |
                      StructureNotifyMask | ButtonPressMask |
                      ButtonReleaseMask | PointerMotionMask;

```

```

winattrs.border_pixel = 0;
winattrs.bit_gravity = StaticGravity;
winmask              = CWBorderPixel | CWBitGravity | CWEEventMask;
window   = XCreateWindow(display, DefaultRootWindow(display),
                        0, 0, 400, 400, 0, vinfo->depth, InputOutput,
                        vinfo->visual, winmask, &winattrs);
XChangeProperty(display, window, XA_WM_NAME, XA_STRING, 8, 0,
                 (unsigned char *)argv[0], strlen(argv[0]));
XChangeProperty(display, window, XA_WM_ICON_NAME, XA_STRING, 8, 0,
                 (unsigned char *)argv[0], strlen(argv[0]));
XMapWindow(display, window);
/*
 * Create the OpenGL context...
 */

context = glXCreateContext(display, vinfo, 0, True);
glXMakeCurrent(display, window, context);
/*
 * Setup remaining globals...
 */
CubeWidth      = 400;
CubeHeight     = 400;
CubeRotation[0] = 45.0f;
CubeRotation[1] = 45.0f;
CubeRotation[2] = 45.0f;
CubeRate[0]     = 1.0f;
CubeRate[1]     = 1.0f;
CubeRate[2]     = 1.0f;
/*
 * Loop forever...
 */
mapped = False;
for (++)
{
/*
 * Use select() to respond asynchronously to events; when the window is
 * not mapped, we wait indefinitely; otherwise we'll timeout after 20ms
 * to rotate the cube...
 */
if (mapped)
{
    FD_ZERO(&input);
    FD_SET(ConnectionNumber(display), &input);
    timeout.tv_sec  = 0;
    timeout.tv_usec = 20000;

    ready = select(ConnectionNumber(display) + 1, &input, NULL, NULL,
                   &timeout);
}
else
    ready = 1;
if (ready)
{
/*
 * An event is ready, handle it...
 */
XNextEvent(display, &event);
switch (event.type)
{
    case MapNotify :
        mapped = True;
    case ConfigureNotify :

```

```

XGetWindowAttributes(display, window, &windata);
ReshapeFunc(windata.width, windata.height);
break;
case UnmapNotify :
    mapped = False;
    break;
case ButtonPress :
    MouseFunc(event.xbutton.button, 0, event.xbutton.x,
              event.xbutton.y);
    break;

case ButtonRelease :
    MouseFunc(event.xbutton.button, 1, event.xbutton.x,
              event.xbutton.y);
    break;
case MotionNotify :
    if (event.xmotion.state & (Button1Mask | Button2Mask | Button3Mask))
        MotionFunc(event.xmotion.x, event.xmotion.y);
    break;
}
}
/*
 * Redraw if the window is mapped...
 */
if (mapped)
{
/*
 * Update the cube rotation...
 */
IdleFunc();
/*
 * Draw the cube...
 */
DisplayFunc();
/*
 * Swap the front and back buffers...
 */
glXSwapBuffers(display, window);
}
}

/*
 * 'DisplayFunc()' - Draw a cube.
 */
void
DisplayFunc(void)
{
    int
        i, j; /* Looping vars */
    static const GLfloat corners[8][3] = /* Corner vertices */
    {
        { 1.0f, 1.0f, 1.0f }, /* Front top right */
        { 1.0f, -1.0f, 1.0f }, /* Front bottom right */
        { -1.0f, -1.0f, 1.0f }, /* Front bottom left */
        { -1.0f, 1.0f, 1.0f }, /* Front top left */
        { 1.0f, 1.0f, -1.0f }, /* Back top right */
        { 1.0f, -1.0f, -1.0f }, /* Back bottom right */
        { -1.0f, -1.0f, -1.0f }, /* Back bottom left */
        { -1.0f, 1.0f, -1.0f } /* Back top left */
    };
    static const int
        sides[6][4] = /* Sides */
    {

```

```

        { 0, 1, 2, 3 },           /* Front */
        { 4, 5, 6, 7 },           /* Back */
        { 0, 1, 5, 4 },           /* Right */
        { 2, 3, 7, 6 },           /* Left */
        { 0, 3, 7, 4 },           /* Top */
        { 1, 2, 6, 5 }            /* Bottom */
    };

static const GLfloat colors[6][3] = /* Colors */
{
    { 1.0f, 0.0f, 0.0f },      /* Red */
    { 0.0f, 1.0f, 0.0f },      /* Green */
    { 1.0f, 1.0f, 0.0f },      /* Yellow */
    { 0.0f, 0.0f, 1.0f },      /* Blue */
    { 1.0f, 0.0f, 1.0f },      /* Magenta */
    { 0.0f, 1.0f, 1.0f }       /* Cyan */
};

/*
 * Clear the window...
 */
glViewport(0, 0, CubeWidth, CubeHeight);
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
/*
 * Setup the matrices...
 */
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-2.0f, 2.0f,
        -2.0f * CubeHeight / CubeWidth, 2.0f * CubeHeight / CubeWidth,
        -2.0f, 2.0f);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glRotatef(CubeRotation[0], 1.0f, 0.0f, 0.0f);
glRotatef(CubeRotation[1], 0.0f, 1.0f, 0.0f);
glRotatef(CubeRotation[2], 0.0f, 0.0f, 1.0f);
/*
 * Draw the cube...
 */
glEnable(GL_DEPTH_TEST);
glBegin(GL_QUADS);

for (i = 0; i < 6; i++)
{
    glColor3fv(colors[i]);
    for (j = 0; j < 4; j++)
        glVertex3fv(corners[sides[i][j]]);
}
glEnd();
}
/*
 * 'IdleFunc()' - Rotate and redraw the cube.
 */
void
IdleFunc(void)
{
    CubeRotation[0] += CubeRate[0];
    CubeRotation[1] += CubeRate[1];
    CubeRotation[2] += CubeRate[2];
}
/*
 * 'MotionFunc()' - Handle mouse pointer motion.

```

```

*/
void
MotionFunc(int x,                                /* I - X position */
           int y)                                /* I - Y position */
{
/*
 * Get the mouse movement...
 */
x -= CubeMouseX;
y -= CubeMouseY;

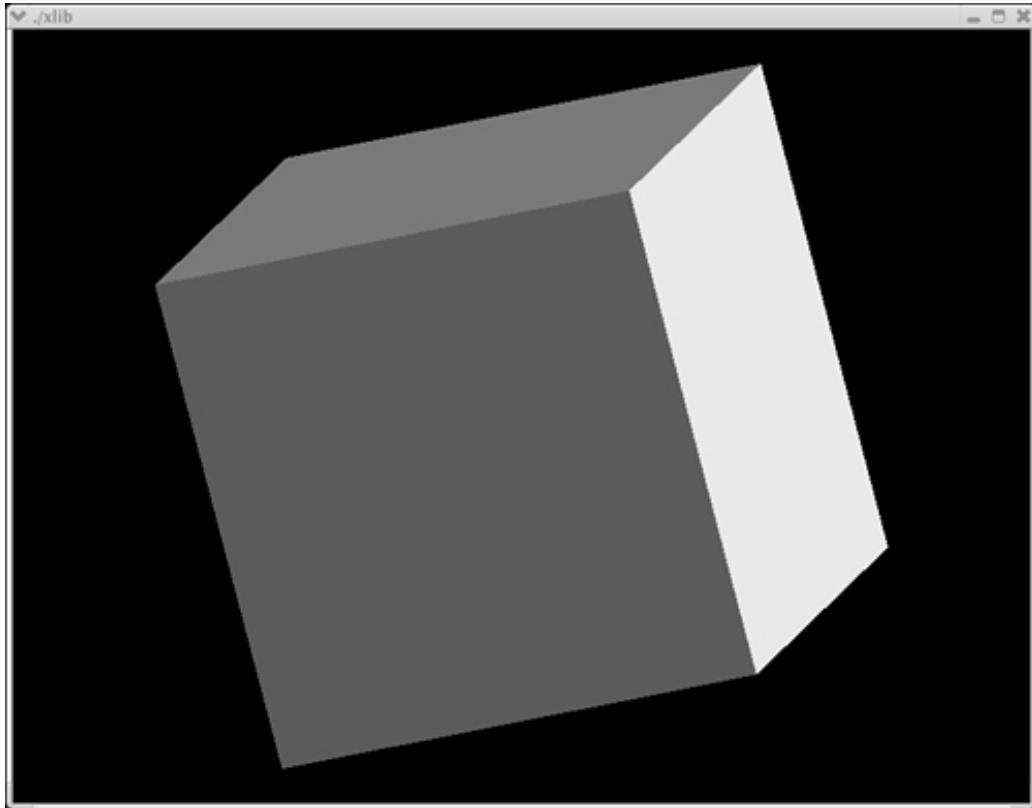
/*
 * Update the cube rotation rate based upon the mouse movement and
 * button...
 */
switch (CubeMouseButton)
{
    case 0 :                                /* Button 1 */
        CubeRate[0] = 0.01f * y;
        CubeRate[1] = 0.01f * x;
        CubeRate[2] = 0.0f;
        break;
    case 1 :                                /* Button 2 */
        CubeRate[0] = 0.0f;
        CubeRate[1] = 0.01f * y;
        CubeRate[2] = 0.01f * x;
        break;
    default :                               /* Button 3 */
        CubeRate[0] = 0.01f * y;
        CubeRate[1] = 0.0f;
        CubeRate[2] = 0.01f * x;
        break;
}
}

/*
 * 'MouseFunc()' - Handle mouse button press/release events.
*/
void
MouseFunc(int button,                            /* I - Button that was pressed */
          int state,                            /* I - Button state (1 = down) */
          int x,                                /* I - X position */
          int y)                                /* I - Y position */
{
/*
 * Only respond to button presses...
 */
if (state)
    return;
/*
 * Save the mouse state...
 */
CubeMouseButton = button;
CubeMouseX      = x;
CubeMouseY      = y;
/*
 * Zero-out the rotation rates...
 */
CubeRate[0] = 0.0f;
CubeRate[1] = 0.0f;
CubeRate[2] = 0.0f;
}

```

```
/*
 * 'ReshapeFunc( )' - Resize the window.
 */
void
ReshapeFunc( int width,                      /* I - Width of window */
             int height)                     /* I - Height of window */
{
    CubeWidth  = width;
    CubeHeight = height;
}
```

Figure 15.1. The Xlib spinning cube example.



Creating Bitmap Fonts for OpenGL

The GLX extension provides a single function called `glXUseXFont()` for converting an X font to OpenGL bitmaps. Each bitmap is placed in a display list, allowing you to display a string of text using the `glCallLists()` function. You start by looking up an X font using the `XLoadQueryFont()` function:

```
XFontStruct *font;
font = XLoadQueryFont(display, "-* -courier -bold -r -normal -14 -* -* -* -* -* -* -* -*");
```

The sample code loads a 14-pixel Courier Bold font; however, any X font can be used. After you load the X font, you call `glGenLists()` to create display lists for the number of characters you want to use and `glXUseXFont()` to load the display list bitmaps. In the following sample code, characters from the space (32) to delete (127) are loaded into 96 display lists:

```
GLuint listbase;  
listbase = glGenLists(96);  
qlxUseXFont(font->fid, ' ', 96, listbase);
```

Then you can draw text using a combination of the `glRasterPos()`, `glPushAttrib()`, `glListBase()`, `glCallLists()`, and `glPopAttrib()` functions, as follows:

```
char *s = "Hello, World!";
glPushAttrib(GL_LIST_BIT);
glListBase(CubeFont - ' ');
glRasterPos3f(0.0f, 0.0f, 0.0f);
glCallLists(strlen(s), GL_BYTE, s);
glPopAttrib();
```

The example in [Listing 15.4](#) uses this code to draw the names of each side of the cube. [Figure 15.2](#) shows the result.

Listing 15.4. The `xlibfonts.c` Sample Program

```
/*
 * Include necessary headers...
 */
#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>
#include <X11/Xatom.h>
#include <GL/glx.h>
#include <GL/gl.h>
#include <sys/select.h>
#include <sys/types.h>
#include <sys/time.h>
/*
 * Globals...
 */
float      CubeRotation[3],      /* Rotation of cube */
           CubeRate[3];      /* Rotation rate of cube */
int       CubeMouseButton,      /* Button that was pressed */
          CubeMouseX,      /* Start X position of mouse */
          CubeMouseY;      /* Start Y position of mouse */
int       CubeWidth,           /* Width of window */
          CubeHeight;      /* Height of window */
GLuint    CubeFont;           /* Display list base for font */
/*
 * Functions...
 */
void      DisplayFunc(void);
void      IdleFunc(void);
void      MotionFunc(int x, int y);
void      MouseFunc(int button, int state, int x, int y);
void      ReshapeFunc(int width, int height);

/*
 * 'main()' - Main entry for example program.
 */
int
main(int   argc,               /* O - Exit status */
     char *argv[])            /* I - Number of command-line args */
{
    Bool      mapped;        /* I - Command-line args */
    GLXContext context;      /* Is the window mapped? */
    Display   *display;       /* OpenGL context */
    Window    window;        /* X display connection */
    XVisualInfo *vinfo;      /* X window */
    XFontStruct *font;       /* X visual information */
    /* X font information */
```

```

XSetWindowAttributes  winattrs;      /* Window attributes */
int                  winmask;       /* Mask for attributes */
XEvent               event;         /* Event data */
XWindowAttributes    windata;       /* Window data */
struct timeval       timeout;       /* Timeout interval for select() */
fd_set               input;         /* Input set for select() */
int                  ready;         /* Event ready? */
static int            attributes[] = /* OpenGL attributes */
{
    GLX_RGBA,
    GLX_DOUBLEBUFFER,
    GLX_RED_SIZE, 8,
    GLX_GREEN_SIZE, 8,
    GLX_BLUE_SIZE, 8,
    GLX_DEPTH_SIZE, 16,
    0
};

/*
 * Open a connection to the X server...
 */
display = XOpenDisplay(getenv("DISPLAY"));
/*
 * Find the proper visual for an OpenGL window...
 */
vinfo = glXChooseVisual(display, DefaultScreen(display), attributes);
/*
 * Create the window...
 */
winattrs.event_mask = ExposureMask | VisibilityChangeMask |
    StructureNotifyMask | ButtonPressMask |
    ButtonReleaseMask | PointerMotionMask;
winattrs.border_pixel = 0;
winattrs.bit_gravity = StaticGravity;
winmask = CWBorderPixel | CWBitGravity | CWEEventMask;
window = XCreateWindow(display, DefaultRootWindow(display),
    0, 0, 400, 400, 0, vinfo->depth, InputOutput,
    vinfo->visual, winmask, &winattrs);
XChangeProperty(display, window, XA_WM_NAME, XA_STRING, 8, 0,
    (unsigned char *)argv[0], strlen(argv[0]));
XChangeProperty(display, window, XA_WM_ICON_NAME, XA_STRING, 8, 0,
    (unsigned char *)argv[0], strlen(argv[0]));
XMapWindow(display, window);

/*
 * Create the OpenGL context...
 */
context = glXCreateContext(display, vinfo, 0, True);
glXMakeCurrent(display, window, context);
/*
 * Setup remaining globals...
 */
CubeWidth      = 400;
CubeHeight     = 400;
CubeRotation[0] = 45.0f;
CubeRotation[1] = 45.0f;
CubeRotation[2] = 45.0f;
CubeRate[0]     = 1.0f;
CubeRate[1]     = 1.0f;
CubeRate[2]     = 1.0f;
/*
 * Setup font...
 */

```



```

{
/*
 * Update the cube rotation...
 */
IdleFunc();
/*
 * Draw the cube...
 */
DisplayFunc();
/*
 * Swap the front and back buffers...
 */
glXSwapBuffers(display, window);
}
}

/*
 * 'DisplayFunc()' - Draw a cube.
 */
void
DisplayFunc(void)
{
    int             i, j;          /* Looping vars */
    static const GLfloat corners[8][3] = /* Corner vertices */
    {
        { 1.0f,  1.0f,  1.0f },      /* Front top right */
        { 1.0f, -1.0f,  1.0f },      /* Front bottom right */
        { -1.0f, -1.0f,  1.0f },     /* Front bottom left */
        { -1.0f,  1.0f,  1.0f },      /* Front top left */
        { 1.0f,  1.0f, -1.0f },      /* Back top right */
        { 1.0f, -1.0f, -1.0f },      /* Back bottom right */
        { -1.0f, -1.0f, -1.0f },     /* Back bottom left */
        { -1.0f,  1.0f, -1.0f }       /* Back top left */
    };
    static const int sides[6][4] = /* Sides */
    {
        { 0, 1, 2, 3 },             /* Front */
        { 4, 5, 6, 7 },             /* Back */
        { 0, 1, 5, 4 },             /* Right */
        { 2, 3, 7, 6 },             /* Left */
        { 0, 3, 7, 4 },             /* Top */
        { 1, 2, 6, 5 }              /* Bottom */
    };
    static const GLfloat colors[6][3] = /* Colors */
    {
        { 1.0f, 0.0f, 0.0f },       /* Red */
        { 0.0f, 1.0f, 0.0f },       /* Green */
        { 1.0f, 1.0f, 0.0f },       /* Yellow */
        { 0.0f, 0.0f, 1.0f },       /* Blue */
        { 1.0f, 0.0f, 1.0f },       /* Magenta */
        { 0.0f, 1.0f, 1.0f }        /* Cyan */
    };
/*
 * Clear the window...
 */
glViewport(0, 0, CubeWidth, CubeHeight);
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
/*
 * Setup the matrices...
 */
glMatrixMode(GL_PROJECTION);

```

```

glLoadIdentity();
glOrtho(-2.0f, 2.0f,
        -2.0f * CubeHeight / CubeWidth, 2.0f * CubeHeight / CubeWidth,
        -2.0f, 2.0f);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glRotatef(CubeRotation[0], 1.0f, 0.0f, 0.0f);
glRotatef(CubeRotation[1], 0.0f, 1.0f, 0.0f);
glRotatef(CubeRotation[2], 0.0f, 0.0f, 1.0f);
/*
 * Draw the cube...
 */
glEnable(GL_DEPTH_TEST);
glBegin(GL_QUADS);
for (i = 0; i < 6; i++)
{
    glColor3fv(colors[i]);
    for (j = 0; j < 4; j++)
        glVertex3fv(corners[sides[i][j]]);
}
glEnd();

/*
 * Draw lines coming out of the cube...
 */
glColor3f(1.0f, 1.0f, 1.0f);
glBegin(GL_LINES);
    glVertex3f(0.0f, 0.0f, -1.5f);
    glVertex3f(0.0f, 0.0f, 1.5f);
    glVertex3f(-1.5f, 0.0f, 0.0f);
    glVertex3f(1.5f, 0.0f, 0.0f);
    glVertex3f(0.0f, 1.5f, 0.0f);
    glVertex3f(0.0f, -1.5f, 0.0f);
glEnd();
/*
 * Draw text for each side...
 */
glPushAttrib(GL_LIST_BIT);
    glListBase(CubeFont - ' ');
    glRasterPos3f(0.0f, 0.0f, -1.5f);
    glCallLists(4, GL_BYTE, "Back");
    glRasterPos3f(0.0f, 0.0f, 1.5f);
    glCallLists(5, GL_BYTE, "Front");
    glRasterPos3f(-1.5f, 0.0f, 0.0f);
    glCallLists(4, GL_BYTE, "Left");
    glRasterPos3f(1.5f, 0.0f, 0.0f);
    glCallLists(5, GL_BYTE, "Right");
    glRasterPos3f(0.0f, 1.5f, 0.0f);
    glCallLists(3, GL_BYTE, "Top");
    glRasterPos3f(0.0f, -1.5f, 0.0f);
    glCallLists(6, GL_BYTE, "Bottom");
glPopAttrib();
}
/*
 * 'IdleFunc()' - Rotate and redraw the cube.
 */
void
IdleFunc(void)
{
    CubeRotation[0] += CubeRate[0];
    CubeRotation[1] += CubeRate[1];
    CubeRotation[2] += CubeRate[2];
}

```

```

}

/*
 * 'MotionFunc()' - Handle mouse pointer motion.
 */
void
MotionFunc(int x,                                /* I - X position */
           int y)                                /* I - Y position */
{
/*
 * Get the mouse movement...
 */
x -= CubeMouseX;
y -= CubeMouseY;

/*
 * Update the cube rotation rate based upon the mouse movement and
 * button...
 */
switch (CubeMouseButton)
{
    case 0 :                                /* Button 1 */
        CubeRate[0] = 0.01f * y;
        CubeRate[1] = 0.01f * x;
        CubeRate[2] = 0.0f;
        break;
    case 1 :                                /* Button 2 */
        CubeRate[0] = 0.0f;
        CubeRate[1] = 0.01f * y;
        CubeRate[2] = 0.01f * x;
        break;
    default :                               /* Button 3 */
        CubeRate[0] = 0.01f * y;
        CubeRate[1] = 0.0f;
        CubeRate[2] = 0.01f * x;
        break;
}
}

/*
 * 'MouseFunc()' - Handle mouse button press/release events.
 */
void
MouseFunc(int button,                            /* I - Button that was pressed */
          int state,                            /* I - Button state (1 = down) */
          int x,                                /* I - X position */
          int y)                                /* I - Y position */
{
/*
 * Only respond to button presses...
 */
if (state)
    return;
/*
 * Save the mouse state...
 */
CubeMouseButton = button;
CubeMouseX      = x;
CubeMouseY      = y;
/*
 * Zero-out the rotation rates...
 */
CubeRate[0] = 0.0f;
CubeRate[1] = 0.0f;
}

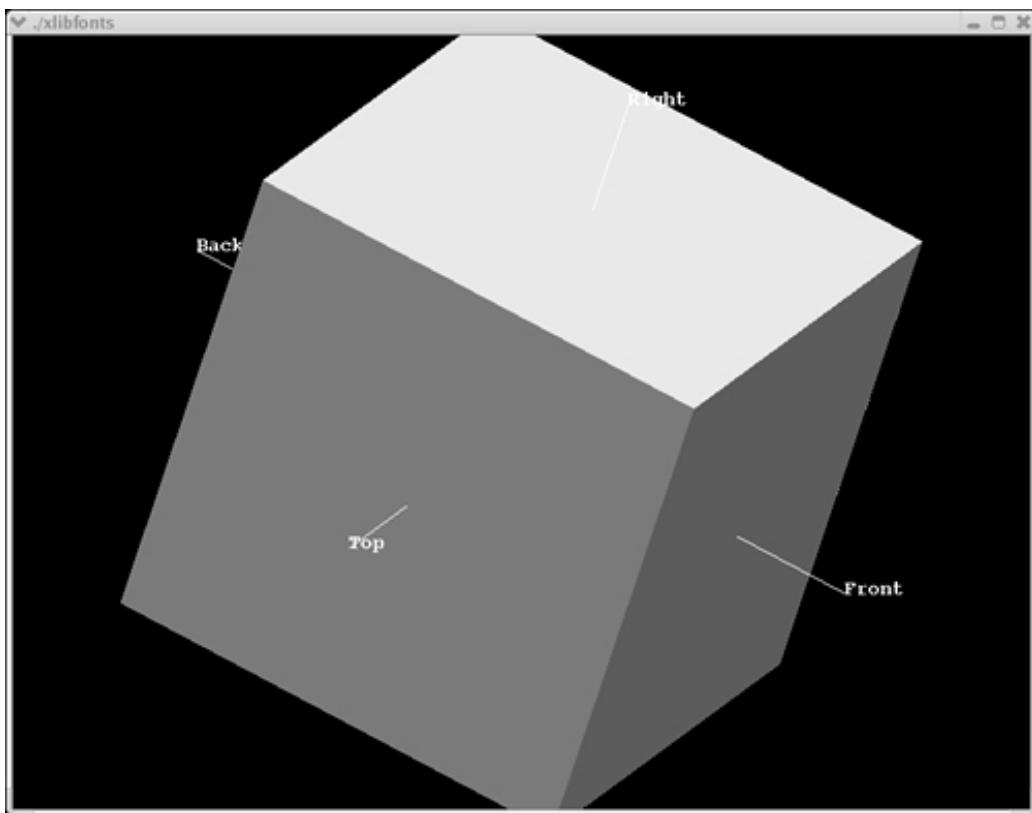
```

```

    CubeRate[2] = 0.0f;
}
/*
 * 'ReshapeFunc()' - Resize the window.
 */
void
ReshapeFunc(int width,           /* I - Width of window */
            int height)        /* I - Height of window */
{
    CubeWidth = width;
    CubeHeight = height;
}

```

Figure 15.2. The Xlib spinning cube with text example.



Offscreen Rendering

GLX supports two types of offscreen rendering, each with its own advantage: GLX pixmaps and Pbuffers.

Using GLX Pixmaps

GLX pixmaps are the original type of offscreen rendering and generally support `TrueColor` and `PseudoColor` visuals. They are generally used when portability is desired over performance; although GLX pixmaps are available on all platforms, GLX pixmaps are not hardware accelerated. GLX pixmaps also often support larger bit depths and dimensions than the graphics hardware, making them ideal for offline rendering of images when graphics card memory is limited.

As with OpenGL windows, GLX pixmaps start with a call to the `glXChooseVisual()` function to find an appropriate visual. Because some systems and graphics cards provide only double-buffered OpenGL visuals, you have to check for both single- and double-buffered visuals:

```

Display *display;
XVisualInfo *vinfo;
static int attributes[] =
{
    GLX_RGBA,
    GLX_RED_SIZE, 8,
    GLX_GREEN_SIZE, 8,
    GLX_BLUE_SIZE, 8,
    GLX_DEPTH_SIZE, 16,
    0,           /* Save space for GLX_DOUBLEBUFFER */
    0
};
display = XOpenDisplay(getenv("DISPLAY"));
vinfo = glXChooseVisual(display, DefaultScreen(display), attributes);
if (!vinfo)
{
/*
 * If no single-buffered visual is available, try a double-buffered one...
 */
attributes[9] = GLX_DOUBLEBUFFER;
vinfo = glXChooseVisual(display, DefaultScreen(display),
                        attributes);
}

```

When you have an appropriate visual, you can create an X `Pixmap` using the `XCreatePixmap()` function; this pixmap will hold the actual pixels for your offscreen buffer. It is then bound to GLX for OpenGL rendering using the `glXCreateGLXPixmap()` function:

```

Pixmap pixmap;
GLXPixmap glx pixmap;
pixmap = XCreatePixmap(display, DefaultRootWindow(display),
                      1024, 1024, vinfo->depth);
glx pixmap = glXCreateGLXPixmap(display, vinfo, pixmap);

```

Finally, you call the `glXCreateContext()` function to create a context for the GLX pixmap, specifying a value of `False` for the fourth parameter for an indirect rendering context:

```

GLXContext context;
context = glXCreateContext(display, vinfo, 0, False);
glXMakeCurrent(display, glx pixmap, context);

```

You can then draw into the pixmap using OpenGL functions and read the results back using the `glReadPixels()` function. Listing 15.5 shows a variation of the previous sample program that creates a GLX pixmap, draws a cube, reads the image using `glReadPixels()`, and writes the result to a PPM image file called `glx pixmap.ppm`.

Listing 15.5. GLX Pixmap Sample Program

```

/*
 * Include necessary headers...
 */
#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>
#include <X11/Xatom.h>
#include <GL/glx.h>
#include <GL/gl.h>
/*
 * Globals...

```

```

/*
float          CubeRotation[3],      /* Rotation of cube */
              CubeRate[3];      /* Rotation rate of cube */
int           CubeWidth,          /* Width of window */
              CubeHeight;        /* Height of window */

/*
 * Functions...
 */
void          DisplayFunc(void);

/*
 * 'main()' - Main entry for example program.
 */

int           /* O - Exit status */
main(int  argc,           /* I - Number of command-line args */
      char *argv[]);      /* I - Command-line args */

{
    GLXContext      context;      /* OpenGL context */
    Display          *display;     /* X display connection */
    Pixmap           pixmap;      /* X pixmap */
    GLXPixmap        glx pixmap;  /* GLX pixmap */
    XVisualInfo     *vinfo;       /* X visual information */
    FILE             *fp;          /* PPM file pointer */
    int              y;           /* Current row */
    unsigned char    pixels[3072]; /* One line of RGB pixels */
    static int       attributes[] = /* OpenGL attributes */
    {
        GLX_RGBA,
        GLX_RED_SIZE, 8,
        GLX_GREEN_SIZE, 8,
        GLX_BLUE_SIZE, 8,
        GLX_DEPTH_SIZE, 16,
        0,           /* Save space for GLX_DOUBLEBUFFER */
        0
    };

/*
 * Open a connection to the X server...
 */
display = XOpenDisplay(getenv("DISPLAY"));
/*
 * Find the proper visual for a GLX pixmap...
 */
vinfo = glXChooseVisual(display, DefaultScreen(display), attributes);
if (!vinfo)
{
/*
 * If no single-buffered visual is available, try a double-buffered one...
 */
attributes[9] = GLX_DOUBLEBUFFER;
vinfo = glXChooseVisual(display, DefaultScreen(display),
                       attributes);
}
if (!vinfo)
{
    puts("No OpenGL visual available!");
    return (1);
}
/*
 * Create the pixmap...
 */
pixmap = XCreatePixmap(display, DefaultRootWindow(display),

```

```

        1024, 1024, vinfo->depth);
glxpixmap = glXCreateGLXPixmap(display, vinfo, pixmap);

/*
 * Create the OpenGL context...
 */
context = glXCreateContext(display, vinfo, 0, False);
glXMakeCurrent(display, glxpixmap, context);
/*
 * Setup remaining globals...
 */
CubeWidth      = 1024;
CubeHeight     = 1024;
CubeRotation[0] = 45.0f;
CubeRotation[1] = 45.0f;
CubeRotation[2] = 45.0f;
CubeRate[0]     = 1.0f;
CubeRate[1]     = 1.0f;
CubeRate[2]     = 1.0f;
/*
 * Draw a cube...
 */
DisplayFunc();
/*
 * Read back the RGB pixels and write the result as a PPM file.
 */
if ((fp = fopen("glxpixmap.ppm", "wb")) == NULL)
    perror("Unable to create glxpixmap.ppm");
else
{
/*
 * Write a PPM image from top to bottom...
 */
fputs("P6\n1024 1024 255\n", fp);
for (y = 1023; y >= 0; y --)
{
    glReadPixels(0, y, 1024, 1, GL_RGB, GL_UNSIGNED_BYTE, pixels);
    fwrite(pixels, 1024, 3, fp);
}
fclose(fp);
}
/*
 * Destroy all resources we used and close the display...
 */
glXDestroyContext(display, context);
glXDestroyGLXPixmap(display, glxpixmap);
XFreePixmap(display, pixmap);
XCloseDisplay(display);
return (0);
}
/*
 * 'DisplayFunc()' - Draw a cube.
 */
void
DisplayFunc(void)
{
    int             i, j;          /* Looping vars */
    static const GLfloat corners[8][3] = /* Corner vertices */
    {
        { 1.0f,  1.0f,  1.0f },      /* Front top right */

```

```

        { 1.0f, -1.0f,  1.0f },      /* Front bottom right */
        { -1.0f, -1.0f,  1.0f },    /* Front bottom left */
        { -1.0f,  1.0f,  1.0f },    /* Front top left */
        { 1.0f,  1.0f, -1.0f },    /* Back top right */
        { 1.0f, -1.0f, -1.0f },    /* Back bottom right */
        { -1.0f, -1.0f, -1.0f },   /* Back bottom left */
        { -1.0f,  1.0f, -1.0f }    /* Back top left */
    };
static const int sides[6][4] = /* Sides */
{
    { 0, 1, 2, 3 },             /* Front */
    { 4, 5, 6, 7 },             /* Back */
    { 0, 1, 5, 4 },             /* Right */
    { 2, 3, 7, 6 },             /* Left */
    { 0, 3, 7, 4 },             /* Top */
    { 1, 2, 6, 5 }              /* Bottom */
};

static const GLfloat colors[6][3] = /* Colors */
{
    { 1.0f, 0.0f, 0.0f },       /* Red */
    { 0.0f, 1.0f, 0.0f },       /* Green */
    { 1.0f, 1.0f, 0.0f },       /* Yellow */
    { 0.0f, 0.0f, 1.0f },       /* Blue */
    { 1.0f, 0.0f, 1.0f },       /* Magenta */
    { 0.0f, 1.0f, 1.0f }        /* Cyan */
};

/*
 * Clear the window...
 */
glViewport(0, 0, CubeWidth, CubeHeight);
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
/*
 * Setup the matrices...
 */
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-2.0f, 2.0f,
        -2.0f * CubeHeight / CubeWidth, 2.0f * CubeHeight / CubeWidth,
        -2.0f, 2.0f);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glRotatef(CubeRotation[0], 1.0f, 0.0f, 0.0f);
glRotatef(CubeRotation[1], 0.0f, 1.0f, 0.0f);
glRotatef(CubeRotation[2], 0.0f, 0.0f, 1.0f);
/*
 * Draw the cube...
 */
glEnable(GL_DEPTH_TEST);
glBegin(GL_QUADS);
for (i = 0; i < 6; i++)
{
    glColor3fv(colors[i]);
    for (j = 0; j < 4; j++)
        glVertex3fv(corners[sides[i][j]]);
}
glEnd();
}

```

Using Pbuffers

Pbuffers, the second type of offscreen buffer, are supported by GLX 1.3 implementations. Pbuffers use graphics memory instead of X pixmaps and are hardware accelerated, providing faster offscreen rendering. However, the use of graphics memory often limits the maximum size of a Pbuffer, and Pbuffers are not universally supported.

Using Pbuffers, unlike OpenGL windows and GLX pixmaps, you start by choosing a framebuffer configuration using the `glXChooseFBConfig()` function instead of `glXChooseVisual()`:

```
Display *display;
int nconfigs;
GLXFBConfig *configs;
static int attributes[] =
{
    GLX_RGBA,
    GLX_RED_SIZE, 8,
    GLX_GREEN_SIZE, 8,
    GLX_BLUE_SIZE, 8,
    GLX_DEPTH_SIZE, 16,
    0,           /* Save space for GLX_DOUBLEBUFFER */
    0
};
display = XOpenDisplay(getenv("DISPLAY"));
configs = glXChooseFBConfig(display, DefaultScreen(display), attributes,
                            &nconfigs);
if (!configs)
{
    attributes[3] = GLX_DOUBLEBUFFER;
    configs = glXChooseFBConfig(display, DefaultScreen(display),
                                attributes, &nconfigs);
}
```

When you have a list of the matching framebuffer configurations, you can create the Pbuffer using the `glXCreatePbuffer()` function. The function takes a display, framebuffer configuration, and list of Pbuffer attributes:

```
GLXPbuffer pbuffer;
static int pbattrs[] =
{
    GLX_PBUFFER_WIDTH, 1024,
    GLX_PBUFFER_HEIGHT, 1024,
    0
};
pbuffer = glXCreatePbuffer(display, *configs, pbattrs);
```

The Pbuffer attribute list consists of the `GLX_PBUFFER_WIDTH` and `GLX_PBUFFER_HEIGHT` values specifying the width and height of the Pbuffer.

After you create the Pbuffer, you call the `glXCreateNewContext()` function to create a context based on the framebuffer configuration for the Pbuffer, specifying a value of `True` for the fifth parameter for a direct rendering context:

```
GLXContext context;
context = glXCreateNewContext(display, *configs, GLX_RGBA_BIT, 0, True);
glXMakeCurrent(display, pbuffer, context);
```

You can then draw into the pixmap using OpenGL functions and read the results back using the `glReadPixels()` function. [Listing 15.6](#) shows a variation of the previous sample program that

creates a Pbuffer, draws a cube, reads the image using `glReadPixels()`, and writes the result to a PPM image file called `pbuffer.ppm`.

Listing 15.6. Pbuffer Sample Program

```
/*
 * Include necessary headers...
 */
#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>
#include <X11/Xatom.h>
#include <GL/glx.h>
#include <GL/gl.h>
/*
 * Globals...
 */
float          CubeRotation[3],           /* Rotation of cube */
               CubeRate[3];           /* Rotation rate of cube */
int            CubeWidth,                /* Width of window */
               CubeHeight;              /* Height of window */

/*
 * Functions...
 */
void          DisplayFunc(void);

/*
 * 'main()' - Main entry for example program.
 */
int
main(int  argc,
     char *argv[])
{
    GLXContext      context;           /* OpenGL context */
    Display         *display;          /* X display connection */
    GLXPbuffer      pbuffer;          /* Pbuffer */
    int             nconfigs;         /* Number of configurations */
    GLXFBConfig    *configs;          /* GLX framebuffer configuration */
    FILE            *fp;              /* PPM file pointer */
    int             y;                /* Current row */
    unsigned char   pixels[3072];      /* One line of RGB pixels */
    static int      attributes[] =   /* OpenGL attributes */
    {
        GLX_RGBA,
        GLX_RED_SIZE, 8,
        GLX_GREEN_SIZE, 8,
        GLX_BLUE_SIZE, 8,
        GLX_DEPTH_SIZE, 16,
        0,           /* Save space for GLX_DOUBLEBUFFER */
        0
    };
    static int      pbattrs[] =      /* Pbuffer attributes */
    {
        GLX_PBUFFER_WIDTH, 1024,
        GLX_PBUFFER_HEIGHT, 1024,
        0
    };

/*
 * Open a connection to the X server...
 */
display = XOpenDisplay(getenv("DISPLAY"));

```

```

/*
 * Get a matching framebuffer configuration...
 */
configs = glXChooseFBConfig(display, DefaultScreen(display), attributes,
                             &nconfigs);
if (!configs)
{
    attributes[3] = GLX_DOUBLEBUFFER;
    configs      = glXChooseFBConfig(display, DefaultScreen(display),
                                     attributes, &nconfigs);
}
if (!configs)
{
    puts("No OpenGL framebuffer configurations available!");
    return (1);
}
/*
 * Create the Pbuffer...
 */
pbuffer = glXCreatePbuffer(display, *configs, pbattrs);
/*
 * Create the OpenGL context...
 */

context = glXCreateNewContext(display, *configs, GLX_RGBA_BIT, 0, True);
glXMakeCurrent(display, pbuffer, context);
/*
 * Setup remaining globals...
 */
CubeWidth      = 1024;
CubeHeight     = 1024;
CubeRotation[0] = 45.0f;
CubeRotation[1] = 45.0f;
CubeRotation[2] = 45.0f;
CubeRate[0]    = 1.0f;
CubeRate[1]    = 1.0f;
CubeRate[2]    = 1.0f;
/*
 * Draw a cube...
 */
DisplayFunc();
/*
 * Read back the RGB pixels and write the result as a PPM file.
 */
if ((fp = fopen("pbuffer.ppm", "wb")) == NULL)
    perror("Unable to create pbuffer.ppm");
else
{
    /*
     * Write a PPM image from top to bottom...
     */
    fputs("P6\n1024 1024 255\n", fp);
    for (y = 1023; y >= 0; y --)
    {
        glReadPixels(0, y, 1024, 1, GL_RGB, GL_UNSIGNED_BYTE, pixels);
        fwrite(pixels, 1024, 3, fp);
    }
    fclose(fp);
}
/*
 * Destroy all resources we used and close the display...

```

```

*/
glXDestroyContext(display, context);
glXDestroyPbuffer(display, pbuffer);
XCloseDisplay(display);
return (0);
}
/*
* 'DisplayFunc()' - Draw a cube.
*/
void
DisplayFunc(void)
{
    int             i, j;           /* Looping vars */
    static const GLfloat corners[8][3] = /* Corner vertices */
    {
        { 1.0f, 1.0f, 1.0f },      /* Front top right */
        { 1.0f, -1.0f, 1.0f },     /* Front bottom right */
        { -1.0f, -1.0f, 1.0f },    /* Front bottom left */
        { -1.0f, 1.0f, 1.0f },     /* Front top left */
        { 1.0f, 1.0f, -1.0f },    /* Back top right */
        { 1.0f, -1.0f, -1.0f },   /* Back bottom right */
        { -1.0f, -1.0f, -1.0f },  /* Back bottom left */
        { -1.0f, 1.0f, -1.0f }    /* Back top left */
    };
    static const int sides[6][4] = /* Sides */
    {
        { 0, 1, 2, 3 },           /* Front */
        { 4, 5, 6, 7 },           /* Back */
        { 0, 1, 5, 4 },           /* Right */
        { 2, 3, 7, 6 },           /* Left */
        { 0, 3, 7, 4 },           /* Top */
        { 1, 2, 6, 5 }            /* Bottom */
    };
    static const GLfloat colors[6][3] = /* Colors */
    {
        { 1.0f, 0.0f, 0.0f },     /* Red */
        { 0.0f, 1.0f, 0.0f },     /* Green */
        { 1.0f, 1.0f, 0.0f },     /* Yellow */
        { 0.0f, 0.0f, 1.0f },     /* Blue */
        { 1.0f, 0.0f, 1.0f },     /* Magenta */
        { 0.0f, 1.0f, 1.0f }      /* Cyan */
    };
/*
* Clear the window...
*/
    glViewport(0, 0, CubeWidth, CubeHeight);
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
/*
* Setup the matrices...
*/
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-2.0f, 2.0f,
            -2.0f * CubeHeight / CubeWidth, 2.0f * CubeHeight / CubeWidth,
            -2.0f, 2.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glRotatef(CubeRotation[0], 1.0f, 0.0f, 0.0f);
    glRotatef(CubeRotation[1], 0.0f, 1.0f, 0.0f);
    glRotatef(CubeRotation[2], 0.0f, 0.0f, 1.0f);
}

```

```

/*
 * Draw the cube...
 */
glEnable(GL_DEPTH_TEST);
glBegin(GL_QUADS);
for (i = 0; i < 6; i++)
{
    glColor3fv(colors[i]);
    for (j = 0; j < 4; j++)
        glVertex3fv(corners[sides[i][j]]);
}
glEnd();
}

```

Using the Motif Library

The Motif library is one of the older toolkits used on UNIX/Linux and is still the standard used for applications developed solely for commercial versions of UNIX. Motif is based on the X Intrinsics (Xt) library, which provides the core support for several other toolkits such as the Athena toolkit (Xaw), the 3D Athena toolkit (Xaw3d), and the neXtaW toolkit, which provides a NextStep look and feel.

Two OpenGL widgets are available: one for generic Xt-based toolkits called `GLwDrawingArea` and one specifically integrated with Motif called `GLwMDrawingArea`. Both are functionally equivalent, so we will create an example based on the Motif toolkit and `GLwMDrawingArea` widget.

The OpenGL widgets are provided in a separate library called `GLw`. You include them in your application by using the `-lGLw` linker option. A typical link command for a Motif-based OpenGL application looks like the following:

```
gcc -o myprogram myprogram.o -lGLw -lGL -lXm -lXt -lXext -lX11
```

GLwDrawingArea and GLwMDrawingArea: The OpenGL Widgets

Like most Motif and Xt widgets, the OpenGL widgets use abbreviated header filenames. The include file for the generic `GlwDrawingArea` widget is `<GL/GLwDrawA.h>`, and the Motif widget is `<GL/GLwMDrawA.h>`. After you include the appropriate header file, you use the `XtVaCreateManagedWidget` function to create the OpenGL widget, as follows:

```

widget = XtVaCreateManagedWidget(
    "name",
    glwMDrawingAreaWidgetClass,
    parent,
    GLwNrgba, True,
    GLwNdoublebuffer, True,
    GLwNdepthSize, 16,
    ... other resource arguments as needed ...
    NULL);

```

The first argument is the resource name of the widget; it can be used to associate additional X resources with the widget, including the OpenGL resources.

The second argument specifies the widget class; `glwMDrawingAreaWidgetClass` specifies the Motif OpenGL widget. You use `glwDrawingAreaWidgetClass` for the generic Xt OpenGL widget.

The third argument specifies the parent widget, which is normally a manager widget like `XmForm`.

The remaining arguments specify widget resources in name/value pairs along with a trailing `NULL` pointer to specify the end of the resource argument list. Normally, Motif and Xt applications use hard-coded resources for widget configuration and resource files and fallback resources for labels and basic look-and-feel preferences. In this case, we specify the OpenGL visual attributes so that the correct visual will be used for double-buffered RGB (TrueColor) drawing. [Table 15.2](#) lists the available OpenGL widget resources. Most directly correspond to the GLX attributes and are used to construct a GLX attribute list. The `GLwNattribList` resource specifies the GLX attribute list directly.

Table 15.2. OpenGL Widget Resources

Resource Name	Resource Type	Corresponding GLX Attribute from Table 15.1
<code>GLwNaccumAlphaSize</code>	<code>int</code>	<code>GLX_ACCUM_ALPHA_SIZE</code>
<code>GLwNaccumBlueSize</code>	<code>int</code>	<code>GLX_ACCUM_BLUE_SIZE</code>
<code>GLwNaccumGreenSize</code>	<code>int</code>	<code>GLX_ACCUM_GREEN_SIZE</code>
<code>GLwNaccumRedSize</code>	<code>int</code>	<code>GLX_ACCUM_RED_SIZE</code>
<code>GLwNalphaSize</code>	<code>int</code>	<code>GLX_ALPHA_SIZE</code>
<code>GLwNattribList</code>	<code>int *</code>	None; specifies the GLX attribute list directly
<code>GLwNauxBuffers</code>	<code>boolean</code>	<code>GLX_AUX_BUFFERS</code>
<code>GLwNblueSize</code>	<code>int</code>	<code>GLX_BLUE_SIZE</code>
<code>GLwNbufferSize</code>	<code>int</code>	<code>GLX_BUFFER_SIZE</code>
<code>GLwNdepthSize</code>	<code>int</code>	<code>GLX_DEPTH_SIZE</code>
<code>GLwNdoublebuffer</code>	<code>boolean</code>	<code>GLX_DOUBLEBUFFER</code>
<code>GLwNgreenSize</code>	<code>int</code>	<code>GLX_GREEN_SIZE</code>
<code>GLwNlevel</code>	<code>int</code>	<code>GLX_LEVEL</code>
<code>GLwNredSize</code>	<code>int</code>	<code>GLX_RED_SIZE</code>
<code>GLwNrgba</code>	<code>boolean</code>	<code>GLX_RGBA</code>
<code>GLwNstencilSize</code>	<code>int</code>	<code>GLX_STENCIL_SIZE</code>
<code>GLwNstereo</code>	<code>boolean</code>	<code>GLX_STEREO</code>

Callbacks

After you create the widget, you must associate several callbacks with one or more callback functions in your application. [Table 15.3](#) lists the callbacks that the OpenGL widgets define.

Table 15.3. OpenGL Widget Callback Resources

Resource Name	Description
<code>GLwNexposeCallback</code>	The redraw callback
<code>GLwNginitCallback</code>	The initialization callback

`GLwNinputCallback` The callback for mouse and keyboard input

`GLwNresizeCallback` The callback for widget resizes

OpenGL widget callbacks take three arguments: the widget pointer, the user data pointer, and a pointer to the `GLwDrawingAreaCallbackStruct` data structure. [Table 15.4](#) shows the members of the structure. You can use the `reason` member of this structure to handle all types of callbacks using a single function.

Table 15.4. `GLwDrawingAreaCallbackStruct` Members

Name	Type	Description
<code>event</code>	<code>XEvent *</code>	The X event associated with the input or expose callback
<code>height</code>	<code>Dimension</code>	The new height of the widget for expose and resize callbacks
<code>reason</code>	<code>int</code>	The reason for the callback: <code>GLwCR_EXPOSE</code> , <code>GLwCR_GINIT</code> , <code>GLwCR_INPUT</code> , or <code>GLwCR_RESIZE</code>
<code>width</code>	<code>Dimension</code>	The new width of the widget for expose and resize callbacks

The `GLwNexposeCallback` Callback

The `GLwNexposeCallback` callback function handles redrawing the widget when the window manager reports that all or part of the widget is exposed and needs to be drawn.

Typically, this function sets the current OpenGL context and draws in the widget. The `width` and `height` members of the callback structure contain the current width and height of the widget, and the `event` member contains any X expose event data that can be used to see whether additional expose events follow or to limit the redraw to the area that needs it.

The `GLwNginitCallback` Callback

The `GLwNginitCallback` callback function handles any initialization of the OpenGL widget.

Typically, this function creates the OpenGL context, loads textures and fonts, and initializes display lists for common display elements.

The `GLwNvisualInfo` resource can be queried by the callback to create the OpenGL context. The following code creates an OpenGL context using the resource value:

```
Widget application_shell;
Widget drawing_area;
XVisualInfo *info;
GLXcontext context;
XtVaGetValues(drawing_area, GLwNvisualInfo, &info, NULL);
context = glXCreateContext(XtDisplay(application_shell), info,
                           NULL, GL_TRUE);
```

This resource and the OpenGL window for the widget are not created until your callback function is called.

The `GLwNinputCallback` Callback

The `GLwNinputCallback` callback function handles user input in the form of button clicks, mouse motion, and keyboard interaction. The event member of the callback data points to the `ButtonPress`, `ButtonRelease`, `MotionNotify`, `KeyPress`, or `KeyRelease` event that triggered the callback.

The `GLwNresizeCallback` Callback

The `GLwNresizeCallback` callback function is called whenever the application or user resizes the OpenGL widget. The `width` and `height` members of the callback data contain the new width and height of the widget and can be used to track changes to the size of the widget. An expose callback with the same information will follow a resize callback, so many applications do not need to use the resize callback.

Functions

The `GLw` library provides two helper functions that work with both of the OpenGL widgets. The `GLwDrawingAreaMakeCurrent()` function sets the current OpenGL context for the widget and must be called before drawing to the widget:

```
GLwDrawingAreaMakeCurrent(drawing_area, context);
```

When you are done drawing in a double-buffered widget, call the `GLwDrawingAreaSwapBuffers()` function to swap the front and back buffers:

```
GLwDrawingAreaSwapBuffers(drawing_area);
```

Putting It All Together

[Listing 15.7](#) shows a Motif version of the `xlibfonts` example presented in [Listing 15.4](#), providing identical output. The program starts by creating an Xt "application shell," which includes the main window. It then adds a Motif `XmForm` widget to manage the OpenGL widget and the OpenGL widget itself:

```
Widget      CubeShell;           /* Application shell */
XtApplicationContext CubeContext; /* Application context */
Widget      CubeGLArea;         /* OpenGL drawing area */
...
Widget      form;               /* Form management widget */
XtApplicationContext context;   /* Application context */
static char *fallback[] =      /* Fallback resources */
{
    "Motif.geometry: 400x400",
    NULL
};

/*
 * Initialize the application window and manager widgets...
 */
CubeShell = XtVaAppInitialize(
    &CubeContext, "Motif", NULL, 0, &argc, argv,
    fallback,
    XmNtitle,    "Motif Example",
    XmNiconName, "Motif",
    NULL);

form = XtVaCreateManagedWidget(
    "form", xmFormWidgetClass, CubeShell,
    NULL);

/*
```

```

* Create the OpenGL drawing area...
*/
CubeGLArea = XtVaCreateManagedWidget(
    "drawingArea", glwMDrawingAreaWidgetClass, form,
    GLwNrgba,           True,
    GLwNdoublebuffer,  True,
    GLwNdepthSize,     16,
    XmNtopAttachment,  XmATTACH_FORM,
    XmNbottomAttachment, XmATTACH_FORM,
    XmNleftAttachment, XmATTACH_FORM,
    XmNrightAttachment, XmATTACH_FORM,
    NULL);

```

The OpenGL widget is attached to the sides of the form, causing it to occupy the entire window. In a typical application with a menu bar, you would probably attach the top of the OpenGL widget to the menu bar instead.

After you create the widgets, you set the callback functions to use for the OpenGL widget:

```

XtAddCallback(CubeGLArea, GLwNexposeCallback,
               (XtCallbackProc)DisplayCB, NULL);
XtAddCallback(CubeGLArea, GLwNinitCallback,
               (XtCallbackProc)InitCB, NULL);
XtAddCallback(CubeGLArea, GLwNresizeCallback,
               (XtCallbackProc)ReshapeCB, NULL);
XtAddCallback(CubeGLArea, GLwNinputCallback,
               (XtCallbackProc)InputCB, NULL);

```

Finally, you "realize" the application shell to show the window and call the `XtAppMainLoop()` function to start the application event loop:

```

XtRealizeWidget(CubeShell);
XtAppMainLoop(CubeContext);

```

The callback functions use the same functions as the `xlibfonts` example to initialize the OpenGL context and font, draw the cube and text, and handle mouse input. A new `TimeOutCB()` function is used to rotate the cube once every 20 milliseconds and is registered via `XtAppAddTimeOut()`:

```

void
TimeOutCB(void)
{
    CubeRotation[0] += CubeRate[0];
    CubeRotation[1] += CubeRate[1];
    CubeRotation[2] += CubeRate[2];
    if (CubeRate[0] || CubeRate[1] || CubeRate[2])
        XmRedisplayWidget(CubeGLArea);
    XtAppAddTimeOut(CubeContext, 20,
                    (XtTimerCallbackProc)TimeOutCB, NULL);
}

```

The `XmRedisplayWidget()` function tells the OpenGL widget to redraw itself and can be used by applications to update their display based on new, possibly asynchronous data or user input.

Listing 15.7. The Motif Sample Source Code

```

/*
 * Include necessary headers...
 */
#include <stdio.h>

```

```

#include <stdlib.h>
#include <Xm/Xm.h>
#include <Xm/Form.h>
#include <GL/GLwMDrawA.h>
/*
 * Globals...
 */
float      CubeRotation[3],      /* Rotation of cube */
           CubeRate[3];      /* Rotation rate of cube */
int        CubeMouseButton,      /* Button that was pressed */
           CubeMouseX,        /* Start X position of mouse */
           CubeMouseY;        /* Start Y position of mouse */
int        CubeWidth,           /* Width of window */
           CubeHeight;        /* Height of window */
GLuint     CubeFont;           /* Display list base for font */
Widget     CubeShell;           /* Application shell */
XtApplicationContext CubeContext; /* Application context */
Widget     CubeGLArea;          /* OpenGL drawing area */
GLXContext CubeGLContext;      /* OpenGL drawing context */
/*
 * Functions...
 */
void      DisplayCB(void);
void      InitCB(Widget w, void *ud, GLwDrawingAreaCallbackStruct *cd);
void      InputCB(Widget w, void *ud, GLwDrawingAreaCallbackStruct *cd);
void      ReshapeCB(Widget w, void *ud, GLwDrawingAreaCallbackStruct *cd);
void      TimeOutCB(void);
/*
 * 'main()' - Main entry for example program.
 */
int      main(int argc,          /* O - Exit status */
            char *argv[]);      /* I - Number of command-line args */
{
    Widget      form;           /* Form management widget */
    XtApplicationContext context; /* Application context */
    static char  *fallback[] =  /* Fallback resources */
    {
        "Motif.geometry: 400x400",
        NULL
    };

/*
 * Initialize the application window and manager widgets...
 */
CubeShell = XtVaAppInitialize(
    &CubeContext, "Motif", NULL, 0, &argc, argv,
    fallback,
    XmNtitle,     "Motif Example",
    XmNiconName,  "Motif",
    NULL);

form = XtVaCreateManagedWidget(
    "form", xmFormWidgetClass, CubeShell,
    NULL);

/*
 * Create the OpenGL drawing area...
 */
CubeGLArea = XtVaCreateManagedWidget(
    "drawingArea", glwMDrawingAreaWidgetClass, form,
    GLwNrgba,           True,
    GLwNdoublebuffer,   True,
    GLwNdepthSize,      16,

```

```

XmNtopAttachment, XmATTACH_FORM,
XmNbottomAttachment, XmATTACH_FORM,
XmNleftAttachment, XmATTACH_FORM,
XmNrightAttachment, XmATTACH_FORM,
NULL);

/*
 * Set callbacks and timeout processing...
 */
XtAddCallback(CubeGLArea, GLwNexposeCallback,
               (XtCallbackProc)DisplayCB, NULL);
XtAddCallback(CubeGLArea, GLwNinitCallback,
               (XtCallbackProc)InitCB, NULL);
XtAddCallback(CubeGLArea, GLwNresizeCallback,
               (XtCallbackProc)ReshapeCB, NULL);
XtAddCallback(CubeGLArea, GLwNinputCallback,
               (XtCallbackProc)InputCB, NULL);

/*
 * Setup remaining globals...
 */
CubeWidth      = 400;
CubeHeight     = 400;
CubeRotation[0] = 45.0f;
CubeRotation[1] = 45.0f;
CubeRotation[2] = 45.0f;
CubeRate[0]     = 1.0f;
CubeRate[1]     = 1.0f;
CubeRate[2]     = 1.0f;

/*
 * Loop forever...
 */
XtRealizeWidget(CubeShell);
XtAppMainLoop(CubeContext);
return (0);
}
/*
 * 'DisplayCB()' - Display callback.
 */
void
DisplayCB(void)
{
    int                  i, j;          /* Looping vars */
    XVisualInfo         *info;        /* Drawing area visual */
    XFontStruct         *font;        /* X font information */
    static const GLfloat corners[8][3] = /* Corner vertices */
    {
        { 1.0f,  1.0f,  1.0f },      /* Front top right */
        { 1.0f, -1.0f,  1.0f },      /* Front bottom right */
        { -1.0f, -1.0f,  1.0f },     /* Front bottom left */
        { -1.0f,  1.0f,  1.0f },     /* Front top left */
        { 1.0f,  1.0f, -1.0f },     /* Back top right */
        { 1.0f, -1.0f, -1.0f },     /* Back bottom right */
        { -1.0f, -1.0f, -1.0f },    /* Back bottom left */
        { -1.0f,  1.0f, -1.0f }      /* Back top left */
    };
    static const int      sides[6][4] = /* Sides */
    {
        { 0, 1, 2, 3 },            /* Front */
        { 4, 5, 6, 7 },            /* Back */
        { 0, 1, 5, 4 },            /* Right */
        { 2, 3, 7, 6 },            /* Left */
        { 0, 3, 7, 4 },            /* Top */
        { 1, 2, 6, 5 }             /* Bottom */
    };
}

```

```

        };
static const GLfloat colors[6][3] = /* Colors */
{
    { 1.0f, 0.0f, 0.0f }, /* Red */
    { 0.0f, 1.0f, 0.0f }, /* Green */
    { 1.0f, 1.0f, 0.0f }, /* Yellow */
    { 0.0f, 0.0f, 1.0f }, /* Blue */
    { 1.0f, 0.0f, 1.0f }, /* Magenta */
    { 0.0f, 1.0f, 1.0f } /* Cyan */
};

/*
 * Clear the window...
 */
glViewport(0, 0, CubeWidth, CubeHeight);
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
/*
 * Setup the matrices...
 */
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-2.0f, 2.0f,
        -2.0f * CubeHeight / CubeWidth, 2.0f * CubeHeight / CubeWidth,
        -2.0f, 2.0f);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glRotatef(CubeRotation[0], 1.0f, 0.0f, 0.0f);
glRotatef(CubeRotation[1], 0.0f, 1.0f, 0.0f);
glRotatef(CubeRotation[2], 0.0f, 0.0f, 1.0f);
/*
 * Draw the cube...
 */
glEnable(GL_DEPTH_TEST);
glBegin(GL_QUADS);
for (i = 0; i < 6; i++)
{
    glColor3fv(colors[i]);
    for (j = 0; j < 4; j++)
        glVertex3fv(corners[sides[i][j]]);
}
glEnd();

/*
 * Draw lines coming out of the cube...
 */
glColor3f(1.0f, 1.0f, 1.0f);
glBegin(GL_LINES);
    glVertex3f(0.0f, 0.0f, -1.5f);
    glVertex3f(0.0f, 0.0f, 1.5f);
    glVertex3f(-1.5f, 0.0f, 0.0f);
    glVertex3f(1.5f, 0.0f, 0.0f);
    glVertex3f(0.0f, 1.5f, 0.0f);
    glVertex3f(0.0f, -1.5f, 0.0f);
glEnd();
/*
 * Draw text for each side...
 */
glPushAttrib(GL_LIST_BIT);
    glListBase(CubeFont - ' ');
    glRasterPos3f(0.0f, 0.0f, -1.5f);
    glCallLists(4, GL_BYTE, "Back");
    glRasterPos3f(0.0f, 0.0f, 1.5f);

```



```

switch (cd->event->type)
{
    case ButtonPress :
        /*
         * Save the initial mouse button + position...
         */
        CubeMouseButton = cd->event->xbutton.button;
        CubeMouseX      = cd->event->xbutton.x;
        CubeMouseY      = cd->event->xbutton.y;
        /*
         * Zero-out the rotation rates...
         */
        CubeRate[0] = 0.0f;
        CubeRate[1] = 0.0f;
        CubeRate[2] = 0.0f;
        break;
    case MotionNotify :
        /*
         * Get the mouse movement...
         */
        x = cd->event->xmotion.x - CubeMouseX;
        y = cd->event->xmotion.y - CubeMouseY;
        /*
         * Update the cube rotation rate based upon the mouse
         * movement and button...
         */
        switch (CubeMouseButton)
        {
            case 0 : /* Button 1 */
                CubeRate[0] = 0.01f * y;
                CubeRate[1] = 0.01f * x;
                CubeRate[2] = 0.0f;
                break;

            case 1 : /* Button 2 */
                CubeRate[0] = 0.0f;
                CubeRate[1] = 0.01f * y;
                CubeRate[2] = 0.01f * x;
                break;

            default : /* Button 3 */
                CubeRate[0] = 0.01f * y;
                CubeRate[1] = 0.0f;
                CubeRate[2] = 0.01f * x;
                break;
        }
        break;
    }
}

/*
 * 'ReshapeCB()' - Resize callback.
 */
void
ReshapeCB(Widget w,
          /* I - Widget */
          void *ud,
          /* I - User data */
          GLwDrawingAreaCallbackStruct *cd)
          /* I - Callback data */
{
    /*
     * Save the current width and height...
     */

```

```

    CubeWidth  = cd->width;
    CubeHeight = cd->height;
}
/*
 * 'TimeOutCB()' - Rotate and redraw the cube.
 */
void
TimeOutCB(void)
{
    CubeRotation[0] += CubeRate[0];
    CubeRotation[1] += CubeRate[1];
    CubeRotation[2] += CubeRate[2];
    if (CubeRate[0] || CubeRate[1] || CubeRate[2])
        XmRedisplayWidget(CubeGLArea);
    XtAppAddTimeOut(CubeContext, 20,
                    (XtTimerCallbackProc)TimeOutCB, NULL);
}

```

Summary

In this chapter, we have taken the basic OpenGL principles and extended them for use in a Linux environment. You have learned how to set up an appropriate visual for an application and how rendering contexts are handled in the Linux environment. You now also know how to deal with double buffered contexts. You learned how to generate and use bitmap fonts. Finally, we introduced using pBuffers for offscreen rendering on Linux.

Reference

glXChooseFBConfig

Purpose: Gets a list of matching framebuffer configurations.

Include File: <GL/glx.h>

Syntax:

```
GLXFBConfig *glXChooseFBConfig(Display *dpy, int
→ screen, const int *attribList, int *nelements);
```

Description: This function finds a list of framebuffer configurations that match the specified GLX attributes.

Parameters:

**dpy* The X display connection

screen The screen to query

**attribList* The NULL-terminated list of GLX attributes

**nelements* Pointer to an integer that will hold the number of framebuffer configurations that are pointed to

Returns: A pointer to an array of matching framebuffer configurations or **NULL** if the X display does not support the framebuffer query. Use the **XFree()** function to free the memory used for the array.

See Also: [glXGetFBConfigs](#), [glXGetVisualFromFBConfig](#), [glXCreatePbuffer](#)

glXChooseVisual**Purpose:** Selects an X visual to use for OpenGL rendering.**Include File:** <GL/glx.h>**Syntax:**

```
XVisualInfo *glXChooseVisual(Display *dpy, int
  ↪ screen, int *attribList);
```

Description: This function finds a visual matching the specified GLX rendering attributes that can be used to create a window or pixmap for OpenGL rendering.**Parameters:******dpy*** The X display connection***screen*** The screen number****attribList*** The zero-terminated attribute list**Returns:** A pointer to a matching X visual information structure or **NULL**.**See Also:** [glXCreateContext](#), [glXCreateGLXPixmap](#)**glXCreateContext****Purpose:** Creates an OpenGL drawing context.**Include File:** <GL/glx.h>**Syntax:**

```
GLXContext glXCreateContext(Display *dpy,
  ↪ XVisualInfo *vis, GLXContext shareList, Bool direct);
```

Description: This function creates a context for OpenGL rendering. The context can be direct-to-hardware or indirect and can share the display lists, textures, and so forth with other OpenGL contexts of the same type.**Parameters:******dpy*** The X display connection****vis*** The X visual to use***shareList*** An OpenGL context for sharing display lists, textures, and so on***direct*** **True** if a direct-to-hardware context is desired; **False** otherwise**Returns:** A new OpenGL context or **NULL** if the context cannot be created.**See Also:** [glXChooseVisual](#), [glXCreateNewContext](#)

glXCreateGLXPixmap**Purpose:** Creates an offscreen pixmap for OpenGL rendering.**Include File:** `<GL/glx.h>`**Syntax:**

```
GLXPixmap glXCreateGLXPixmap(Display *dpy,
➥ XvisualInfo *vis, Pixmap pixmap);
```

Description: This function creates a GLX pixmap that can be used to do offscreen rendering of OpenGL scenes. Only indirect rendering contexts may be used with GLX pixmaps.**Parameters:**`*dpy` The X display connection`*vis` The X visual to use`pixmap` The X pixmap to use**Returns:** The new GLX pixmap.**See Also:** `glXChooseVisual`, `glXCreateContext`**glXCreateNewContext****Purpose:** Creates a new OpenGL context.**Include File:** `<GL/glx.h>`**Syntax:**

```
GLXContext glXCreateNewContext(Display *dpy,
➥ GLXFBConfig config, int renderType, GLXContext
➥ shareList, Bool direct);
```

Description: This function creates a context for OpenGL rendering and is functionally equivalent to `glXCreateContext()`. The context can be direct-to-hardware or indirect and can share the display lists, textures, and so forth with other OpenGL contexts of the same type.**Parameters:**`Display *dpy` The X display connection`config` The framebuffer configuration to use`renderType` The color type of the context: `GLX_RGBA_TYPE` for RGBA rendering or `GLX_COLOR_INDEX_TYPE` for color indexed rendering`shareList` An OpenGL context for sharing display lists, textures, and so on`direct` `True` if a direct-to-hardware context is desired; `False` otherwise**Returns:** A new OpenGL context or `NULL` if the context cannot be created.

See Also: [glXChooseFBConfig](#), [glXCreateContext](#), [glXDestroyContext](#), [glXGetFBConfigs](#)

glXCreatePbuffer

Purpose: Creates an offscreen pixel buffer for OpenGL rendering.

Include File: `<GL/glx.h>`

Syntax:

```
GLXPbuffer glXCreatePbuffer(Display *dpy,
→ GLXFBCConfig config, const int *attribList);
```

Description: This function creates an offscreen pixel buffer for OpenGL rendering. The dimensions of the Pbuffer are specified using the `GLX_WIDTH` and `GLX_HEIGHT` attributes.

Parameters:

`*dpy` The X display connection

`config` The framebuffer configuration

`*attribList` The zero-terminated list of GLX attributes

Returns: A new Pbuffer or `NULL` if it could not be created.

See Also: [glXGetFBConfigs](#), [glXChooseFBConfig](#), [glXDestroyPbuffer](#)

glXDestroyContext

Purpose: Destroys an OpenGL context.

Include File: `<GL/glx.h>`

Syntax:

```
void glXDestroyContext(Display *dpy, GLXContext ctx);
```

Description: This function destroys an OpenGL rendering context, freeing any system resources associated with it.

Parameters:

`*dpy` The X display connection

`ctx` The OpenGL context

Returns: Nothing.

See Also: [glXCreateContext](#)

glXDestroyGLXPixmap**Purpose:** Destroys a GLX pixmap.**Include File:** <GL/glx.h>**Syntax:**

```
void glXDestroyGLXPixmap(Display *dpy, GLXPixmap pix);
```

Description: This function destroys a GLX pixmap resource. You must still destroy the X pixmap resource and OpenGL context separately.**Parameters:*****dpy** The X display connection**pix** The GLX pixmap**Returns:** Nothing.**See Also:** [glXCreateGLXPixmap](#)**glXDestroyPbuffer****Purpose:** Destroys an offscreen pixel buffer.**Include File:** <GL/glx.h>**Syntax:**

```
void glXDestroyPbuffer(Display *dpy, GLXPbuffer pbuf);
```

Description: This function releases all resources used for the specified Pbuffer.**Parameters:*****dpy** The X display connection**pbuf** The Pbuffer**Returns:** Nothing.**See Also:** [glXCreatePbuffer](#)**glXGetFBConfigs****Purpose:** Gets a list of supported framebuffer configurations.**Include File:** <GL/glx.h>**Syntax:**

```
GLXFBCConfig *glXGetFBConfigs(Display *dpy, int
➥ screen, int *nelements);
```

Description: This function gets a list of supported framebuffer configurations for the specified display and screen.

Parameters:

**dpy* The X display connection
screen The screen to query
**nelements* Pointer to an integer that will hold the number of framebuffer configurations that are pointed to

Returns: A pointer to an array of framebuffer configurations or **NULL** if the X display does not support the framebuffer query. Use the **XFree()** function to free the memory used for the array.

See Also: [glXChooseFBConfig](#), [glXGetVisualFromFBConfig](#), [glXCreatePbuffer](#)

glXGetVisualFromFBConfig

Purpose: Gets the X visual for a specific framebuffer configuration.

Include File: [<GL/glx.h>](#)

Syntax:

```
XVisualInfo *glXGetVisualFromFBConfig(Display *dpy
➥ , GLXFBCConfig config);
```

Description: This function finds the X visual that corresponds to the given framebuffer configuration.

Parameters:

**dpy* The X display connection
config The framebuffer configuration

Returns: A pointer to the **XVisualInfo** structure containing all the X visual information for the given framebuffer configuration.

See Also: [glXGetFBConfigs](#), [glXChooseFBConfigs](#), [glXCreatePbuffer](#)

glXMakeCurrent

Purpose: Sets the current OpenGL context for rendering.

Include File: [<GL/glx.h>](#)

Syntax:

```
Bool glXMakeCurrent(Display *dpy, GLXDrawable
    ↪ drawable, GLXContext ctx);
```

Description: This function sets the current OpenGL context and drawable (window or pixmap) to use when rendering. If there are any pending OpenGL drawing commands on the previous context, they are flushed prior to changing the current context. Pass `None` for the drawable and `NULL` for the context arguments to flush pending OpenGL commands and release the current context.

Parameters:

`*dpy` The X display connection

`ctx` The OpenGL context

`drawable` The window or pixmap

Returns: `True` if the context is set successfully; `False` otherwise.

See Also: `glXCreateContext`, `glXCreateNewContext`, `glXSwapBuffers`

glXSwapBuffers

Purpose: Swaps the front and back display buffers.

Include File: `<GL/glx.h>`

Syntax:

```
void glXSwapBuffers(Display *dpy, GLXDrawable
    ↪ drawable);
```

Description: This function swaps the back buffer with the front buffer, synchronizing with the vertical retrace of the screen as necessary.

Parameters:

`*dpy` The X display connection

`drawable` The window or pixmap

Returns: Nothing.

See Also: `glXCreateContext`, `glXCreateNewContext`, `glXMakeCurrent`

glXUseXFont

Purpose: Creates a collection of bitmap display lists.

Include File: `<GL/glx.h>`

Syntax:

```
void glXUseXFont(Font font, int first, int count,
    int listbase);
```

Description: This function creates *count* display lists containing bitmaps of characters in the specified font. You allocate the display lists for the bitmaps using the `glGenLists()` function.

Parameters:

font Specifies the font to use
first Specifies the first character in the font to use
count Specifies the number of characters to use from the font
listbase Specifies the first display list to use as returned by `glGenLists()`

Returns: Nothing.

See Also: `glXCreateContext`

Part III: OpenGL: The Next Generation

Now we come to what I feel is the most exciting part of the book—and what is perhaps the most exciting development in PC 3D graphics since hardware-accelerated Transform and Lighting. The 3D pipeline that OpenGL uses is pretty much *the* standard pipeline for creating real-time 3D graphics, regardless of which API you are using. However, graphics techniques have evolved to the point that the design of the pipeline has reached its full potential. True photo-realistic rendering requires a great deal more processing applied per vertex, or even to the fragments rendered between vertices.

To increase realism and extend graphics beyond what is possible with the standard pipeline, vendors have designed their hardware to be more flexible and allow not only the processing of a fixed set of commands and state variables, but also the actual execution of graphics code on their now renamed Graphics Processing Units (GPUs). The programmable pipeline is perhaps the single biggest development in commodity graphics that we will see in our careers. One vendor calls it "Cinematic Computing," and that description is not far off the mark. We can now approach in real-time the photo-realistic effects that used to take hours or days to create. OpenGL, too, has evolved to meet this new era.

The following chapters take you through some of the more recent innovations in 3D graphics: from finer grained control of graphics processing memory to extended buffer capabilities and finally to a full-featured graphics programming language executed by the graphics hardware.

Chapter 16. Buffer Objects: It's Your Video Memory; You Manage It!

by Benjamin Lipchak

WHAT YOU'LL LEARN IN THIS CHAPTER:

How To	Functions You'll Use
Create, bind to, and delete buffer objects	<code>glGenBuffers/glBindBuffer/glDeleteBuffers</code>
Send data into a buffer object indirectly	<code>glBufferData/glBufferSubData</code>
Write data into a buffer object directly	<code>glMapBuffer/glUnmapBuffer</code>

Graphics cards today have nearly as much memory as the rest of the system they're plugged into. The amount of graphics card memory tends to at least be within an order of magnitude, say 25%, of the amount of system memory. That's quite a resource to exploit—or to waste by not making the best use of it.

Video memory has traditionally been used for storing the following:

- Front buffers (what you see on the screen)
- Back buffers (what you don't see when double-buffering)
- Depth buffers (for hidden surface removal)
- Other per-pixel storage, such as stencil planes, overlay planes, and so on

Even at high resolutions and color depths, such as 1920x1280 and 32 bits per pixel, graphics consume only in the ballpark of 25 to 40MB. Depending on your available video memory, that can leave hundreds of megabytes at your disposal.

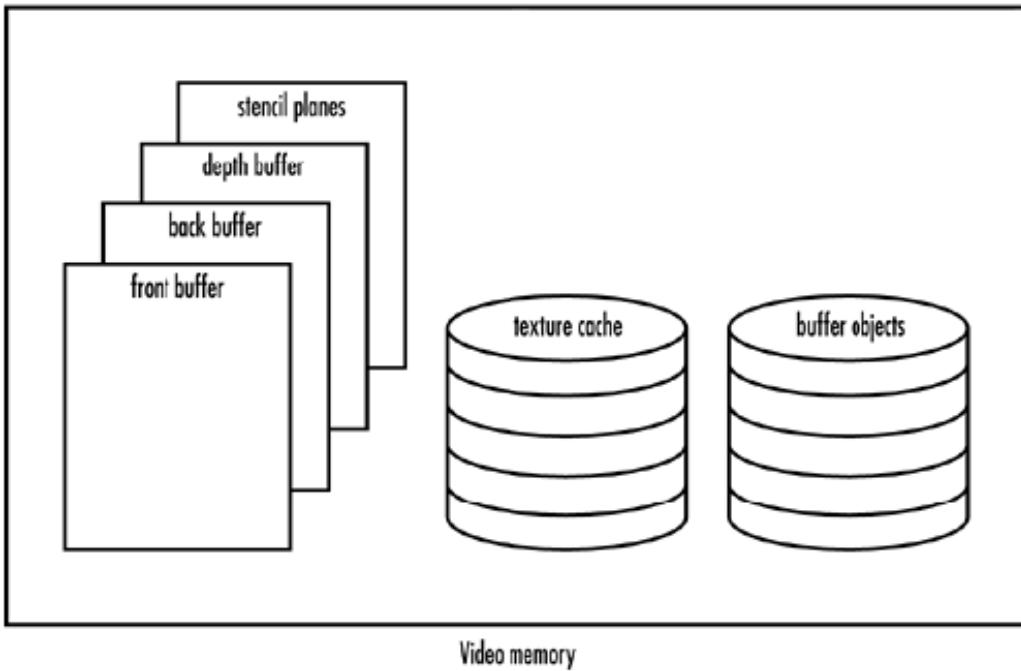
This extra space is most often used to cache texture maps so they don't have to be continually transmitted from system memory to the graphics card every time a new texture is used. Instead, they are kept locally in video memory so they're ready when needed. When the texture cache becomes full, old textures that haven't been used recently are evicted to make room for new textures.

Some OpenGL implementations also attempt to cache geometry data in video memory, such as that present in display lists or vertex arrays. Unfortunately, the driver doesn't know how often the geometry is going to change or how much total geometry there's going to be. For vertex arrays, it doesn't know when the data in the arrays has actually been changed by the application. Only the application knows all this information, so if the driver bothers to try at all, the best it can do is guess, and that's not enough to guarantee optimal performance.

Extensions such as `GL_EXT_compiled_vertex_array`, `GL_EXT_draw_range_elements`, `GL_NV_vertex_array_range`, and `GL_ATI_vertex_array_object` have been introduced over the years to attempt to supply the driver with some of this information. This progress has culminated in a single extension, `GL_ARB_vertex_buffer_object`, which hands over full control to the application when it comes to storing its geometry in local video memory for optimal rendering performance. This extension was promoted into OpenGL 1.5 as a core feature.

[Figure 16.1](#) illustrates the different types of data that share, and in fact compete for, local video memory on the graphics card.

Figure 16.1. A variety of data is stored in local video memory for quick access by the Graphics Processing Unit (GPU), saving a trip to system memory.



First, You Need Vertex Arrays

Buffer objects are repositories for storing data in local video memory. You can store anything you want in there and read it back later. If you want to store your grocery list in there, you're free to do that. But the only useful things to store in there are vertex arrays and array indices. You can clue OpenGL in on the fact that your vertex arrays live in a buffer object, at which point they become blazingly fast vertex arrays.

First, though, you need your vertex arrays. If your application uses *immediate mode* (`glBegin`/`glEnd` pairs), you can't take advantage of buffer objects without first switching over to the vertex array paradigm, discussed in [Chapter 11](#), "It's All About the Pipeline: Faster Geometry Throughput." When you have vertex arrays working, putting them into buffer objects is relatively easy. It also makes before and after performance comparisons straightforward and gratifying!

For our sample program, we'll construct vertex arrays with the geometry for some sphere-shaped particle clouds. The more of a geometry burden we can introduce, the more improvement we'll see when we get around to accelerating them. So let's lay on the vertices!

The number of particles per sphere is configurable. If your OpenGL implementation cannot handle the number of spheres defined here, or if it eats them for breakfast and wants more, just change this constant:

```
GLint numSphereVertices = 30000;
```

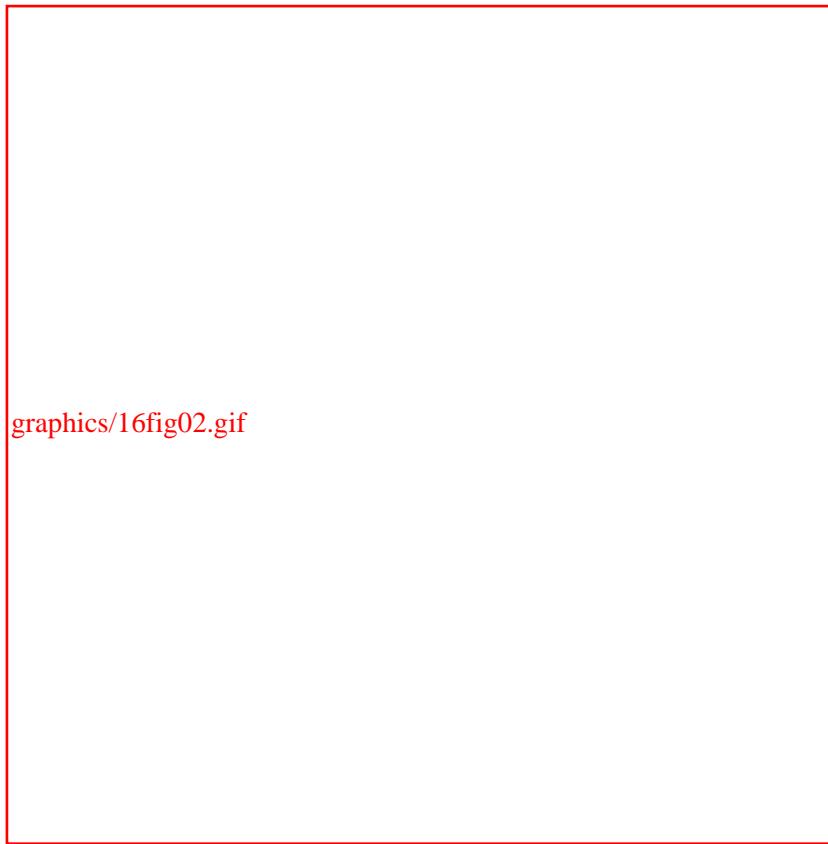
Generating the Spherical Particle Clouds

We need some geometry for this program, but we don't want to waste space in our code to load anything fancy, nor do we want to waste time explaining it. So we'll settle for something simple to generate, but moderately interesting: particle cloud spheres.

We've already decided how many vertices we want, set by the constant shown in the preceding section. We'll just scatter these points randomly across the surface of a sphere. Sounds complicated, right? Not really. All we have to do is generate a random point in space. We take the vector between this random point and the origin (0,0,0) and normalize it to a unit vector. It now represents a point 1 unit away from the origin in some random direction. [Figure 16.2](#) illustrates the normalization of the random vectors. Repeat 30,000 times, and we have a sphere-shaped

cloud of particles.

Figure 16.2. Random vectors are normalized onto the surface of a unit sphere.



Here's the code:

```
for (i = 0; i < numSphereVertices; i++)
{
    GLfloat r1, r2, r3, scaleFactor;
    // pick a random vector
    r1 = (GLfloat)(rand() - (RAND_MAX/2));
    r2 = (GLfloat)(rand() - (RAND_MAX/2));
    r3 = (GLfloat)(rand() - (RAND_MAX/2));
    // determine normalizing scale factor
    scaleFactor = 1.0f / sqrt(r1*r1 + r2*r2 + r3*r3);
    sphereVertexArray[(i*3)+0] = r1 * scaleFactor;
    sphereVertexArray[(i*3)+1] = r2 * scaleFactor;
    sphereVertexArray[(i*3)+2] = r3 * scaleFactor;
}
```

Enabling the Vertex Arrays

We have the data prepared. Now we must enable the arrays and set the array pointers so that OpenGL will know where to find the geometry when rendering:

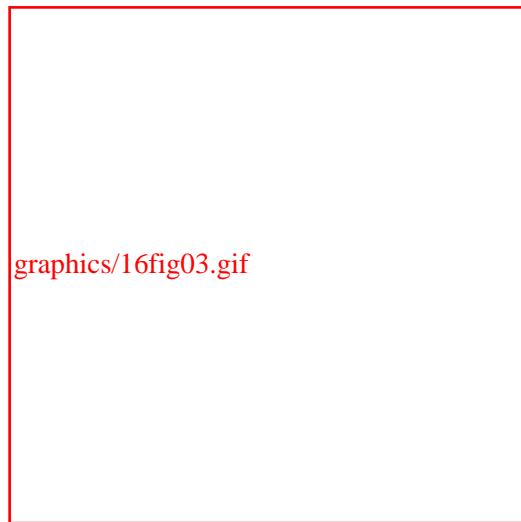
```
glNormalPointer(GL_FLOAT, 0, sphereVertexArray);
glVertexPointer(3, GL_FLOAT, 0, sphereVertexArray);
...
 glEnableClientState(GL_NORMAL_ARRAY);
 glEnableClientState(GL_VERTEX_ARRAY);
```

Notice that we're enabling two arrays: one for the vertex position, but also one for the vertex

normal. Normals make lighting possible, and it just so happens that for a unit sphere (where radius is 1) at the origin, the position is the same as the normal! So we can reuse the same data for both arrays.

Figure 16.3 visually depicts data in our vertex array.

Figure 16.3. Our vertex array data includes random positions that will also serve as surface normals.



More Spheres, Please!

Thirty thousand vertices might sound like a lot, but to bring our OpenGL implementation to its knees, we're going to have to throw it a bit more geometry still. So let's draw a 3x3x3 cube of spheres and set a different color for each cube. As demonstrated in [Listing 16.1](#), we can reuse the same vertex arrays, just changing the modelview matrix to individually resize and locate each sphere in between calls to `glDrawArrays`.

Listing 16.1. Sphere Vertex Array Drawn 27 Times

```
// Called to draw scene
void RenderScene(void)
{
    static GLTStopwatch stopWatch;
    static int frameCounter = 0;
    // Get initial time
    if (frameCounter == 0)
        gltStopwatchReset(&stopWatch);
    frameCounter++;
    if (frameCounter == 100)
    {
        frameCounter = 0;
        fprintf(stdout, "FPS: %f\n", 100.0f / gltStopwatchRead(&stopWatch));
        gltStopwatchReset(&stopWatch);
    }
    // Track camera angle
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0f, 1.0f, 10.0f, 10000.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(cameraPos[0], cameraPos[1], cameraPos[2],
              0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);
```

```

// Clear the window with current clearing color
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
if (animating)
{
    RegenerateSphere();
    SetRenderingMethod();
}
// Draw objects in the scene
DrawModels();
// Flush drawing commands
glutSwapBuffers();
glutPostRedisplay();
}

frameCounter++;
if (frameCounter == 100)
{
    long thisTime;
    frameCounter = 0;
    _ftime(&timeBuffer);
    thisTime = (timeBuffer.time * 1000) + timeBuffer.millitm;
    fprintf(stdout, "FPS: %f\n", 100.0f * 1000.0f / (thisTime - lastTime));
    lastTime = thisTime;
}
// Track camera angle
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(45.0f, 1.0f, 10.0f, 10000.0f);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(cameraPos[0], cameraPos[1], cameraPos[2],
           0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);
// Clear the window with current clearing color
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
if (animating)
{
    RegenerateSphere();
    SetRenderingMethod();
}
// Draw objects in the scene
DrawModels();
// Flush drawing commands
glutSwapBuffers();
glutPostRedisplay();
}

```

Two things to notice from this listing are the performance measurement code and the animation code. We need a way to test dynamically changing geometry, so when the animation toggle is on, we regenerate a new sphere vertex array for every frame. The random animated points sort of look like static.

Because this chapter is all about squeezing more performance out of geometry processing, we would be remiss not to measure that performance in some way. For every 100 frames we render, we look at the time that has elapsed since we started those 100 frames. Divide the 100 frames across the elapsed time, and we have a rough count of frames per second. This number lets us compare vertex array performance against buffer object performance. It is printed to `stdout`, so look for it in the console window, not in the sample program's graphics window.

Migration to Buffer Objects

Believe it or not, we've done the hard part. Generating or loading vertex array data still remains the same burden it always was. All we're going to do differently now is tell OpenGL to store the vertex array data inside a buffer object. Same stuff, different wrapper.

Before getting our hands dirty, we need to take care of one minor detail. Buffer objects are relatively new to OpenGL. The feature was first introduced as the `GL_ARB_vertex_buffer_object` extension and was promoted as a core feature quickly thereafter when OpenGL 1.5 was ratified. Some new features, such as depth textures and shadows, are easily integrated into applications because all they need are some new token definitions from a header file. Unfortunately, buffer objects need a bit more to make them start working. They introduce new API entrypoints that you need to latch onto before you can start using them.

As you do with any feature, you must make sure the appropriate extension or version of OpenGL is available before trying to use it. Here, we're checking for either OpenGL 1.5, which includes buffer objects, or for the extension that also provides the equivalent functionality:

```
// Make sure required functionality is available!
version = glGetString(GL_VERSION);
if ((version[0] == '1') && (version[1] == '.') &&
    (version[2] >= '5') && (version[2] <= '9'))
{
    glVersion15 = GL_TRUE;
}
if (!glVersion15 && !gltIsExtSupported("GL_ARB_vertex_buffer_object"))
{
    fprintf(stderr, "Neither OpenGL 1.5 nor GL_ARB_vertex_buffer_object"
                " extension is available!\n");
    Sleep(2000);
    exit(0);
}
```

Now that we know the feature is supported, we need the function pointers for its entrypoints. On Windows platforms, the function `wglGetProcAddress` queries for the function pointers based on a string containing the entrypoint name. Other platforms have other means of providing these pointers, so we've abstracted them into a tool library function, `gltGetExtensionPointer`. Note that if only the extension is available, the function names have the `ARB` suffix. If OpenGL 1.5 is available, we don't need the suffix:

```
// Load the function pointers
if (glVersion15)
{
    glBindBuffer = gltGetExtensionPointer("glBindBuffer");
    glBufferData = gltGetExtensionPointer("glBufferData");
    glBufferSubData = gltGetExtensionPointer("glBufferSubData");
    glDeleteBuffers = gltGetExtensionPointer("glDeleteBuffers");
    glGenBuffers = gltGetExtensionPointer("glGenBuffers");
    glMapBuffer = gltGetExtensionPointer("glMapBuffer");
    glUnmapBuffer = gltGetExtensionPointer("glUnmapBuffer");
}
else
{
    glBindBuffer = gltGetExtensionPointer("glBindBufferARB");
    glBufferData = gltGetExtensionPointer("glBufferDataARB");
    glBufferSubData = gltGetExtensionPointer("glBufferSubDataARB");
    glDeleteBuffers = gltGetExtensionPointer("glDeleteBuffersARB");
    glGenBuffers = gltGetExtensionPointer("glGenBuffersARB");
    glMapBuffer = gltGetExtensionPointer("glMapBufferARB");
    glUnmapBuffer = gltGetExtensionPointer("glUnmapBufferARB");
```

```

}

if (!glBindBuffer || !glBufferData || !glDeleteBuffers ||
    !glGenBuffers || !glMapBuffer || !glUnmapBuffer)
{
    fprintf(stderr, "Not all entrypoints were available!\n");
    Sleep(2000);
    exit(0);
}

```

Buffer Object Management

Buffer objects are treated similarly to other objects in OpenGL, such as texture objects. They are created and their state initialized when first bound with the `glBindBuffer` command.

`glGenBuffers` can be called first to get a list of available names, but this command doesn't actually create the buffer objects. You still need to bind the object name before it's created.

When you're finished with your buffer objects, you delete them with `glDeleteBuffers`. If a deleted buffer is currently bound, that binding is undone, and the null buffer object (name zero) is implicitly bound, telling OpenGL to go back to using traditional (nonbuffer object) vertex arrays:

```

// Generate a buffer object
glGenBuffers(1, &bufferID);
...
glBindBuffer(GL_ARRAY_BUFFER, bufferID);
...
glDeleteBuffers(1, &bufferID);

```

Rendering with Buffer Objects

When you have a buffer object bound, it tells OpenGL to source its vertex array data from the buffer object's data store instead of the vertex array pointers set via commands like `glVertexPointer`. But those pointers still are used. Instead of being pointers to data in client memory, when a buffer object is bound, these pointers are interpreted as offsets within the buffer object's data store.

In our buffer object program, the data used for both the vertex position array and the normal array begins right at the beginning of the buffer object's data store, so an offset of zero is used. Our data is tightly packed without any padding or interlacing, so the stride parameter is also zero:

```

glBindBuffer(GL_ARRAY_BUFFER, bufferID);
// No stride, no offset
glNormalPointer(GL_FLOAT, 0, 0);
glVertexPointer(3, GL_FLOAT, 0, 0);
...
glDrawArrays(GL_POINTS, 0, numSphereVertices);

```

Loading Data into Buffer Objects

We've described how to create buffer objects and how to tell OpenGL to use them as the source for rendering geometry. But we're still missing one important piece of the puzzle: how to load data into the buffer object. A buffer object starts out its life with an empty data store, so we need to take care of that before doing anything useful with them. The next two sections describe your two choices for loading up your buffer object's data store.

Copying Data into the Buffer Object

The first option for loading your buffer object's data store is analogous to the one you use for loading texture image data into a texture object. You give `glTexImage` a pointer to your texel

data, and OpenGL copies it into its internal texture storage. If you give it a null pointer, `glTexImage` still creates the texture with the size you want but leaves the texels uninitialized. If you want to respecify a portion of the texture, you can call `glTexSubImage` and tell it where and how much data to replace.

The procedure is the same with buffer objects. You can call `glBufferData` to establish the size of your data store and to supply a hint about how it will be accessed. Data is copied from the pointer you provide, unless the pointer is null, in which case the data remains uninitialized. `glBufferSubData` can be called to respecify a portion of the data store.

`glTexImage` and `glTexSubImage` entrypoints accept a target parameter to indicate which texture target is being specified, such as `GL_TEXTURE_2D` or `GL_TEXTURE_3D`. Similarly, `glBufferData`, `glBufferSubData`, and other buffer object entrypoints also accept a target parameter. This target reflects which type of buffer object is being operated on, and can either be `GL_ARRAY_BUFFER` or `GL_ELEMENT_ARRAY_BUFFER`. The former is used to store vertex array data, including colors, normals, texture coordinates, and positions. The latter is used to store array indices as used by `glDrawElements`.

In our sample program, if we're not animating, we simply create the data store by calling `glBufferData`, providing a usage hint that the data will be static. All buffer sizes are measured in terms of "basic machine units," or bytes:

```
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) *
              numSphereVertices * 3, sphereVertexArray,
              GL_STATIC_DRAW);
```

However, if we are animating, we don't want to incur the expense of re-creating the data store during every frame of animation, when in fact the size of the data remains the same. Instead, we'll create the data store once when entering animation mode, with a null pointer so no data is copied yet, and providing a usage hint that the data will be streaming (used only once). Then, for each frame, we update the data with `glBufferSubData`:

```
// Establish streaming buffer object
// Data will be loaded with subsequent calls to glBufferSubData
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) *
              numSphereVertices * 3, NULL, GL_STREAM_DRAW);
...
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(GLfloat) *
              numSphereVertices * 3, sphereVertexArray);
```

The usage hints that we've been passing into `glBufferData` are really just that—hints. One OpenGL implementation may ignore them completely, whereas another implementation may be able to base crucial decisions on a hint that will greatly impact the performance of your buffer objects. If your data never changes, the driver may decide to put the data in local video memory. If you hint that your data is dynamic, the driver may place your data somewhere it is cheaper to constantly update it, such as AGP memory. [Table 16.1](#) shows your three hint options.

Table 16.1. Buffer Object Usage Hints

Usage Hint	Description

<code>GL_DYNAMIC_DRAW</code>	The data stored in the buffer object is likely to change frequently but is likely to be used as a source for drawing several times in between changes. This hint tells the implementation to put the data somewhere it won't be too painful to update once in a while.
<code>GL_STATIC_DRAW</code>	The data stored in the buffer object is unlikely to change and will be used possibly many times as a source for drawing. This hint tells the implementation to put the data somewhere it's quick to draw from, but probably not quick to update.
<code>GL_STREAM_DRAW</code>	The data store in the buffer object is likely to change frequently and will be used only once (or at least very few times) in between changes. This hint tells the implementation that you have time-sensitive data such as animated geometry that will be used once and then replaced. It is crucial that the data be placed somewhere quick to update, even at the expense of faster rendering.

Note that there are actually `READ` and `COPY` variants of these hints in addition to the `DRAW` variants, but they are just in place for future use by extensions that build on buffer object functionality. Drawing from buffer object data is currently the only usage model available.

Mapping the Buffer Object Directly

This indirect copy method for loading buffer objects works well and follows similar paradigms already used by texture objects. However, after we set up our data in client memory, this method requires OpenGL to perform a copy into the buffer object memory. What if we could cut out the middleman and generate our geometry data right into the buffer object memory?

We can. This procedure is called *mapping* your buffer object. By calling `glMapBuffer`, you can get a pointer to the buffer object's data store mapped into the client's address space. This means you can write to it, read from it, whatever you want. The only string attached is that you can't use this memory as a source or destination parameter for other OpenGL entrypoints, such as `glTexImage`, `glLightfv`, `glDrawPixels`, and `glReadPixels`. Here we map the buffer object:

```
glBindBuffer(GL_ARRAY_BUFFER, bufferID);
// Avoid pipeline flush during glMapBuffer by
// marking buffer object's data store as empty
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) *
    numSphereVertices * 3, NULL,
    animating ? GL_STREAM_DRAW : GL_STATIC_DRAW);
sphereVertexArray = (GLfloat *)glMapBuffer(GL_ARRAY_BUFFER,
    GL_WRITE_ONLY);
```

Before calling `glMapBuffer`, we first call `glBufferData` with a null pointer. We do this even if we've already created the buffer object because it helps prevent a performance degradation during mapping. If there is any rendering still in the pipeline using old data in the buffer object, `glMapBuffer` has to wait for the pipeline to flush out before it can return the mapped buffer pointer to the application. (Otherwise, the application might alter data that still needs to be used by previous drawing commands, corrupting that rendering.) Emptying the buffer object's data store with the null pointer tells OpenGL that any data previously loaded into the buffer object is no longer needed. Mapping will then issue a new unused data store, avoiding the implicit pipeline synchronization that would otherwise occur.

When calling `glMapBuffer`, you must provide an access flag that tells OpenGL in which ways you'll be accessing the buffer object's data store while it's mapped. [Table 16.2](#) describes the three access modes.

Table 16.2. Mapped Buffer Object Access Modes

Access Mode	Description
<code>GL_READ_ONLY</code>	If you don't need to change data but simply need to look at or copy data from the data store, you should use read-only mode. An implementation will attempt to map the data store in a way that is most efficient to read. Attempts to write to the mapped data store will likely be slow or cause your application to crash.
<code>GL_READ_WRITE</code>	If you must both read and write from the data, this is the mode you want to use. This mapping may be the least efficient because the implementation may have to compromise between mapping the data store in a way that is efficient to read versus efficient to write.
<code>GL_WRITE_ONLY</code>	If you don't need to read back your data, just change it, you should use write-only mode. An implementation will attempt to map the data store in a way that is most efficient to write. Attempts to read from the data store will likely be slow or cause your application to crash.

You can't start drawing from the buffer object until it has been unmapped again with the `glUnmapBuffer` command. This command lets OpenGL know that you've made your changes to the data, and you're ready to hand control of the data store back to OpenGL. But things can go wrong during the time you have the buffer object mapped. Various system events, such as video mode changes or power-saving suspend/hibernation, can put your buffer object into a state where its integrity is unknown. The memory could have been temporarily reclaimed or powered off, leaving its contents corrupted or otherwise unknown. In this rare case, `glUnmapBuffer` returns `GL_FALSE`, meaning that the application is responsible for resupplying all the data. This is an unavoidable burden if you want your application to be robust. In our program, all we have to do is call back into the generation routine to try again:

```
if (!glUnmapBuffer(GL_ARRAY_BUFFER))
{
    // Some window system event has trashed our data...
    // Try, try again!
    RegenerateSphere();
}
```

Figure 16.4 shows the output from Listing 16.2. The output looks the same from both the traditional vertex array code and the buffer object code. But if you look at the frames per second output, that's where you'll see the difference. Roughly a 25% to 200% improvement or more may be observed in mapped buffer object mode, depending on your OpenGL implementation and the number of vertices you're throwing at it.

Listing 16.2. Code for Regenerating and Loading Sphere Vertex Array Data

```
// Called to regenerate points on the sphere
void RegenerateSphere(void)
{
    GLint i;
    if (mapBufferObject && useBufferObject)
    {
        // Delete old vertex array memory
        if (sphereVertexArray)
            free(sphereVertexArray);
        glBindBuffer(GL_ARRAY_BUFFER, bufferID);
        // Avoid pipeline flush during glMapBuffer by
        // marking buffer object's data store as empty
        glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) *
```

```

        numSphereVertices * 3, NULL,
        animating ? GL_STREAM_DRAW : GL_STATIC_DRAW);
sphereVertexArray = (GLfloat *)glMapBuffer(GL_ARRAY_BUFFER,
                                         GL_WRITE_ONLY);
}
else if (!sphereVertexArray)
{
    // We need our old vertex array memory back
    sphereVertexArray = (GLfloat *)malloc(sizeof(GLfloat) *
                                         numSphereVertices * 3);
    if (!sphereVertexArray)
    {
        fprintf(stderr, "Unable to allocate memory for vertex arrays!");
        Sleep(2000);
        exit(0);
    }
}
for (i = 0; i < numSphereVertices; i++)
{
    GLfloat r1, r2, r3, scaleFactor;
    // pick a random vector
    r1 = (GLfloat)(rand() - (RAND_MAX/2));
    r2 = (GLfloat)(rand() - (RAND_MAX/2));
    r3 = (GLfloat)(rand() - (RAND_MAX/2));
    // determine normalizing scale factor
    scaleFactor = 1.0f / sqrt(r1*r1 + r2*r2 + r3*r3);
    sphereVertexArray[(i*3)+0] = r1 * scaleFactor;
    sphereVertexArray[(i*3)+1] = r2 * scaleFactor;
    sphereVertexArray[(i*3)+2] = r3 * scaleFactor;
}
if (mapBufferObject && useBufferObject)
{
    if (!glUnmapBuffer(GL_ARRAY_BUFFER))
    {
        // Some window system event has trashed our data...
        // Try, try again!
        RegenerateSphere();
    }
    sphereVertexArray = NULL;
}
// Switch between buffer objects and plain old vertex arrays
void SetRenderingMethod(void)
{
    if (useBufferObject)
    {
        glBindBuffer(GL_ARRAY_BUFFER, bufferID);
        // No stride, no offset
        glNormalPointer(GL_FLOAT, 0, 0);
        glVertexPointer(3, GL_FLOAT, 0, 0);

        if (!mapBufferObject)
        {
            if (animating)
            {
                glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(GLfloat) *
                               numSphereVertices * 3, sphereVertexArray);
            }
            else
            {
                // If not animating, this gets called once
                // to establish new static buffer object

```

```

        glBindBuffer(GL_ARRAY_BUFFER, sizeof(GLfloat) *
            numSphereVertices * 3, sphereVertexArray,
            GL_STATIC_DRAW);
    }
}
else
{
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glNormalPointer(GL_FLOAT, 0, sphereVertexArray);
    glVertexPointer(3, GL_FLOAT, 0, sphereVertexArray);
}
}

```

Figure 16.4. The 3x3x3 array of particle cloud spheres is a mini color cube.



A Few Loose Ends

The sample program did not touch upon a couple of things; thus, they have not been discussed yet. The first is state queries. As with all OpenGL state, the new state related to buffer objects can be queried. See the reference section for details, but here's a brief breakdown of the state-querying entrypoints introduced for buffer objects:

- `glGetBufferParameteriv`— Query a buffer object's usage hint, mapped access flag, mapped status, and size.
- `glGetBufferPointerv`— Query a buffer object's mapped address.
- `glGetBufferSubData`— Query data from a buffer object's data store.
- `glIsBuffer`— Query if a buffer object name corresponds to an existing buffer object.

Our sample also did not use array indices. The benefit of using `glDrawElements` is that it removes redundancy from a pool of vertices. If a particular vertex is used more than once (for example, on a corner shared by several triangles), the vertex need only be present and processed once in the vertex array, but can be referenced many times in the array indices. Our spheres were composed of thousands of individual points, each used only once per sphere. So there was no benefit to be

gained from array indices. However, should you use them in your application, it's good to know that they can also be placed in buffer objects. You just need to use the `GL_ELEMENT_ARRAY_BUFFER` target instead of `GL_ARRAY_BUFFER`.

Summary

OpenGL implementations have traditionally been crippled in terms of their ability to efficiently take geometry from an application and render it. This has not been due to any technical obstacle. The application had no way of telling the driver how large a data set would be used or how often it would be updated—crucial information for the driver to know where it should store the data. The problem has just been a lack of communication, cleared up by the introduction of buffer objects.

Buffer objects are created and deleted just like texture objects. In fact, there's a way to copy data into buffer objects that is also similar to texture objects. But buffer objects also provide a powerful mechanism for mapping buffer object memory to the application's address space so the data store can be updated directly without an additional copy required.

Reference

glBindBuffer

Purpose: Binds a buffer object.

Include File: `<glext.h>`

Syntax:

```
void glBindBuffer(GLenum target, GLuint buffer);
```

Description: This function binds a buffer object to the vertex array or array index buffer target. If the buffer object has not been bound before, it's created. Subsequent changes to or queries of the target will affect or return state from the bound buffer object.

Parameters:

`target` `GLenum`: The type of data that will be sourced from the buffer object. It can be one of the following constants:

`GL_ARRAY_BUFFER`: The buffer object will store vertex array data.

`GL_ELEMENT_ARRAY_BUFFER`: The buffer object will store array indices.

`buffer` `GLuint`: The name of the buffer object to bind.

Returns: None.

See Also: `glGenBuffers`, `glDeleteBuffers`, `glIsBuffer`

glBufferData

Purpose: Creates and initializes a buffer object's data store.

Include File: `<glext.h>`

Syntax:

```
void glBufferData(GLenum target, GLsizeiptr size,
    const GLvoid *data, GLenum usage);
```

Description:

This function creates and initializes a buffer object's data store based on the specified size and performance hint. Any pre-existing data is deleted. If the data pointer is non-null, data is copied into the data store. If the buffer object's data store is mapped, it becomes unmapped.

Parameters:*target*

`GLenum`: The buffer object target whose data store is being created. It can be one of the following constants:

`GL_ARRAY_BUFFER`: The vertex array buffer object's data store is being created.

`GL_ELEMENT_ARRAY_BUFFER`: The array index buffer object's data store is being created.

size

`GLsizeiptr`: The size of the buffer object's new data store, measured in basic machine units.

data

`const GLvoid *:` The pointer to data that will be copied into the data store for initialization, or `NULL` if no data is to be copied.

usage

`GLenum`: A performance hint indicating how the data store will be accessed. It can be one of the following constants:

`GL_DYNAMIC_COPY`: The data will be changed often and will be used for both reading from and writing to OpenGL. (Currently, there is no mechanism for reading from OpenGL.)

`GL_DYNAMIC_DRAW`: The data will be changed often and will be used for writing to OpenGL.

`GL_DYNAMIC_READ`: The data will be changed often and will be used for reading from OpenGL. (Currently, there is no mechanism for reading from OpenGL.)

`GL_STATIC_COPY`: The data will be changed rarely and will be used for both reading from and writing to OpenGL. (Currently, there is no mechanism for reading from OpenGL.)

`GL_STATIC_DRAW`: The data will be changed rarely and will be used for writing to OpenGL.

`GL_STATIC_READ`: The data will be changed rarely and will be used for reading from OpenGL. (Currently, there is no mechanism for reading from OpenGL.)

`GL_STREAM_COPY`: The data will be accessed only once or a few times, and will be used for both reading from and writing to OpenGL. (Currently, there is no mechanism for reading from OpenGL.)

GL_STREAM_DRAW: The data will be accessed only once or a few times, and will be used for writing to OpenGL.

GL_STREAM_READ: The data will be accessed only once or a few times, and will be used for reading from OpenGL. (Currently, there is no mechanism for reading from OpenGL.)

Returns: None.

See Also: [glBindBuffer](#), [glBufferSubData](#), [glMapBuffer](#), [glUnmapBuffer](#)

glBufferSubData

Purpose: Updates a subset of a buffer object's data store.

Include File: `<glext.h>`

Syntax:

```
void glBuffer SubData(GLenum target, GLintptr
→ offset, GLsizeiptr size, const GLvoid *data);
```

Description: This function replaces a portion of a buffer object's data store based on the specified size and offset into the data store. Data store contents outside of this portion remain unchanged. An error is generated if the buffer object's data store is currently mapped, or if any part of the specified portion falls outside of the buffer object's data store.

Parameters:

target `GLenum`: The buffer object target whose data store is being updated. It can be one of the following constants:

`GL_ARRAY_BUFFER`: The vertex array buffer object's data store is being updated.

`GL_ELEMENT_ARRAY_BUFFER`: The array index buffer object's data store is being updated.

offset `GLintptr`: The offset into the buffer object's data store where data replacement begins, measured in basic machine units.

size `GLsizeiptr`: The size of the data store region being replaced, measured in basic machine units.

data `const GLvoid *`: The pointer to data that will be copied into the data store for initialization, or `NULL` if no data is to be copied.

Returns: None.

See Also: [glBindBuffer](#), [glBufferData](#), [glMapBuffer](#), [glUnmapBuffer](#)

glDeleteBuffers**Purpose:** Deletes one or more buffer objects.**Include File:** `<glext.h>`**Syntax:**

```
void glDeleteBuffers(GLsizei n, const GLuint
→ *buffers);
```

Description: This function deletes buffer objects. The contents are deleted, and the names are marked as unused. If such a buffer object is currently bound in the current context, all such bindings are reset to zero. If an unused buffer object name or name zero is specified for deletion, that name is silently ignored.**Parameters:*****n*** `GLsizei`: The number of buffer objects to delete.***buffers*** `const GLuint *`: Pointer to an array containing the names of the buffer objects to delete.**Returns:** None.**See Also:** `glGenBuffers`, `glBindBuffer`, `glIsBuffer`**glGenBuffers****Purpose:** Returns unused buffer object names.**Include File:** `<glext.h>`**Syntax:**

```
void glGenBuffers(GLsizei n, GLuint *buffers);
```

Description: This function returns unused buffer object names. The names can subsequently be bound and created with `glBindBuffer`.**Parameters:*****n*** `GLsizei`: The number of buffer object names to return.***buffers*** `GLuint *`: Pointer to an array to fill with unused buffer object names.**Returns:** None.**See Also:** `glBindBuffer`, `glDeleteBuffers`, `glIsBuffer`**glGetBufferParameteriv****Purpose:** Queries properties of a buffer object.

Include File: <glext.h>**Syntax:**

```
void glGetBufferParameteriv(GLenum target, GLenum
→ value, GLint *data);
```

Description: This function queries the state of a currently bound buffer object.**Parameters:**

target *GLenum*: The buffer object target whose state is being queried. It can be one of the following constants:

GL_ARRAY_BUFFER: The vertex array buffer object's state is being queried.

GL_ELEMENT_ARRAY_BUFFER: The array index buffer object's state is being queried.

value *GLenum*: The buffer object state being queried. It can be one of the following constants:

GL_BUFFER_ACCESS: The buffer object's access flag, which can be one of the following: *GL_READ_ONLY*, *GL_WRITE_ONLY*, or *GL_READ_WRITE*.

GL_BUFFER_MAPPED: The flag indicating whether the buffer object is currently mapped.

GL_BUFFER_SIZE: The size of the buffer object, measured in basic machine units.

GL_BUFFER_USAGE: The buffer object's usage pattern, which can be one of the following: *GL_DYNAMIC_COPY*, *GL_DYNAMIC_READ*, *GL_DYNAMIC_WRITE*, *GL_STATIC_COPY*, *GL_STATIC_READ*, *GL_STATIC_WRITE*, *GL_STREAM_COPY*, *GL_STREAM_READ*, or *GL_STREAM_WRITE*.

data *GLint* *: A pointer to the location where the query results are to be written.

Returns: None.**See Also:** *glBindBuffer*, *glBufferData*, *glMapBuffer*, *glUnmapBuffer*

glGetBufferPointerv

Purpose:

Queries the pointer to a mapped buffer object's data store.

Include File:

<glext.h>

Syntax:

```
void glGetBufferPointerv(GLenum target, GLenum
→ pname, GLvoid **params);
```

Description:

This function returns the pointer to which the buffer object's data store is mapped. If the buffer object is not currently mapped, `NULL` is returned. Depending on the implementation, `NULL` may also be returned if queried by a client that did not map the buffer object.

Parameters:*target*

`GLenum`: The buffer object target whose data store pointer is being queried. It can be one of the following constants:

`GL_ARRAY_BUFFER`: The vertex array buffer object's pointer is being queried.

`GL_ELEMENT_ARRAY_BUFFER`: The array index buffer object's pointer is being queried.

pname

`GLenum`: The pointer being queried, which must be the constant `BUFFER_MAP_POINTER`.

params

`GLvoid **`: A pointer to the location where the query result will be stored.

Returns:

None.

See Also:

`glBindBuffer`, `glMapBuffer`

glGetBufferSubData

Purpose: Queries a subset of a buffer object's data store.

Include File: `<glext.h>`

Syntax:

```
void glGetBuffer SubData(GLenum target, GLintptr
➥ offset, GLsizeiptr size, GLvoid *data);
```

Description: This function returns data from the specified portion of the current buffer object's data store. An error is generated if the buffer object is currently mapped.

Parameters:*target*

`GLenum`: The buffer object target whose data is being queried. It can be one of the following constants:

`GL_ARRAY_BUFFER`: The vertex array buffer object's data is being queried.

`GL_ELEMENT_ARRAY_BUFFER`: The array index buffer object's data is being queried.

offset

`GLintptr`: The offset into the buffer object's data store where data querying begins, measured in basic machine units.

size

`GLsizeiptr`: The size of the data store region being queried, measured in basic machine units.

data `GLvoid *`: A pointer to the location where queried data is returned.

Returns: None.

See Also: `glBindBuffer`, `glBufferData`, `glBufferSubData`, `glMapBuffer`, `glUnmapBuffer`

glIsBuffer

Purpose: Queries whether a name is a buffer object name.

Include File: `<glext.h>`

Syntax:

```
GLboolean glIsBuffer(GLuint buffer);
```

Description: This function queries whether the specified name is the name of a buffer object.

Parameters:

buffer `GLuint`: The buffer object name to be queried.

Returns: `GLboolean`: `GL_TRUE` is returned if a buffer object with this name has previously been bound and not yet deleted. Otherwise, `GL_FALSE` is returned.

See Also: `glBindBuffer`, `glDeleteBuffers`, `glGenBuffers`

glMapBuffer

Purpose: Maps a buffer object's data store.

Include File: `<glext.h>`

Syntax:

```
GLvoid *glMapBuffer(GLenum target, GLenum access);
```

Description:

This function maps a buffer object's data store to the client's address space. The data can then be directly read and/or written relative to the returned pointer, depending on the specified buffer access flag. An error is generated if the buffer object's data store is already mapped. The parameter values passed to OpenGL commands may not be sourced from the returned pointer.

Parameters:

target `GLenum`: The buffer object target whose data store is being mapped. It can be one of the following constants:

- `GL_ARRAY_BUFFER`: The vertex array buffer object's data store is being mapped.
- `GL_ELEMENT_ARRAY_BUFFER`: The array index buffer object's data store is being mapped.

access

GLenum: The access flag dictating whether it is possible to read from, write to, or both read from and write to the mapped buffer object's data store. It can be one of the following constants:

GL_READ_ONLY: Restricts access to the buffer object's data store such that writes to the data store may not be performed.

GL_READ_WRITE: Allows reads from and writes to the buffer object's data store without restriction.

GL_WRITE_ONLY: Restricts access to the buffer object's data store such that reads from the data store may not be performed.

Returns:

GLvoid *: A pointer to the buffer object's data store mapped to the client's address space.

See Also:

[glBindBuffer](#), [glBufferData](#), [glBufferSubData](#), [glUnmapBuffer](#)

glUnmapBuffer

Purpose: Unmaps a buffer object's data store.

Include File: `<glext.h>`

Syntax:

```
GLboolean glUnmapBuffer(GLenum target);
```

Description: This function unmaps a buffer object's data store. A data store must be unmapped before it can once again be accessed by OpenGL. An error is generated if the buffer object's data store is not currently mapped.

Parameters:

target **GLenum**: The buffer object target whose data store is being unmapped. It can be one of the following constants:

GL_ARRAY_BUFFER: The vertex array buffer object's data store is being unmapped.

GL_ELEMENT_ARRAY_BUFFER: The array index buffer object's data store is being unmapped.

Returns: **GLboolean**: **GL_TRUE** is returned unless some platform-dependent event has occurred, such as a video mode or power-saving mode change. Such events may leave the buffer object's data store in an undefined state, in which case **GL_FALSE** is returned, and the client must repopulate the data store.

See Also: [glBindBuffer](#), [glMapBuffer](#)

Chapter 17. Occlusion Queries: Why Do More Work Than You Need To?

by Benjamin Lipchak

WHAT YOU'LL LEARN IN THIS CHAPTER:

How To	Functions You'll Use
Create and delete query objects	<code>glGenQueries/glDeleteQueries</code>
Define bounding box occlusion queries	<code>glBeginQuery/glEndQuery</code>
Retrieve the results from an occlusion query	<code>glGetQueryObjectiv</code>

Complex scenes contain hundreds of objects and thousands upon thousands of polygons. Consider the room you're in now, reading this book. Look at all the furniture, objects, other people or pets and think of the rendering power needed to accurately represent their complexity. Several readers will find themselves happily sitting on a crate near a computer in an empty studio apartment, but the rest will envision a significant rendering workload around them.

Now think of all the things you can't see: objects hidden behind other objects, in drawers, or even in the next room. From most viewpoints, these objects are invisible to the viewer. If you rendered the scene, the objects would be drawn, but eventually something would draw over the top of them. Why bother doing all that work for nothing?

Enter occlusion queries. In this chapter, we describe a powerful new feature included in OpenGL 1.5 that can save a tremendous amount of vertex processing and texturing at the expense of a bit of extra nontextured fill rate. Often this trade-off is a very favorable one. We explore the use of occlusion detection and witness the dramatic increase in frame rates this technique affords.

The World Before Occlusion Queries

To show off the improved performance possible through the use of occlusion queries, we need an experimental control group. We'll draw a scene without any fancy occlusion detection. The scene is contrived so that there are plenty of objects both visible and hidden at any given time.

First, we'll draw the "main occluder." An *occluder* is a large object in a scene that tends to occlude, or hide, other objects in the scene. An occluder is often low detail, whereas the objects it occludes may be much higher in detail. Good examples are walls, floors, and ceilings. The main occluder in this scene is a grid made out of six walls, as illustrated in [Figure 17.1](#). [Listing 17.1](#) shows how the walls are actually just scaled cubes.

Listing 17.1. Main Occluder with Six Scaled and Translated Solid Cubes

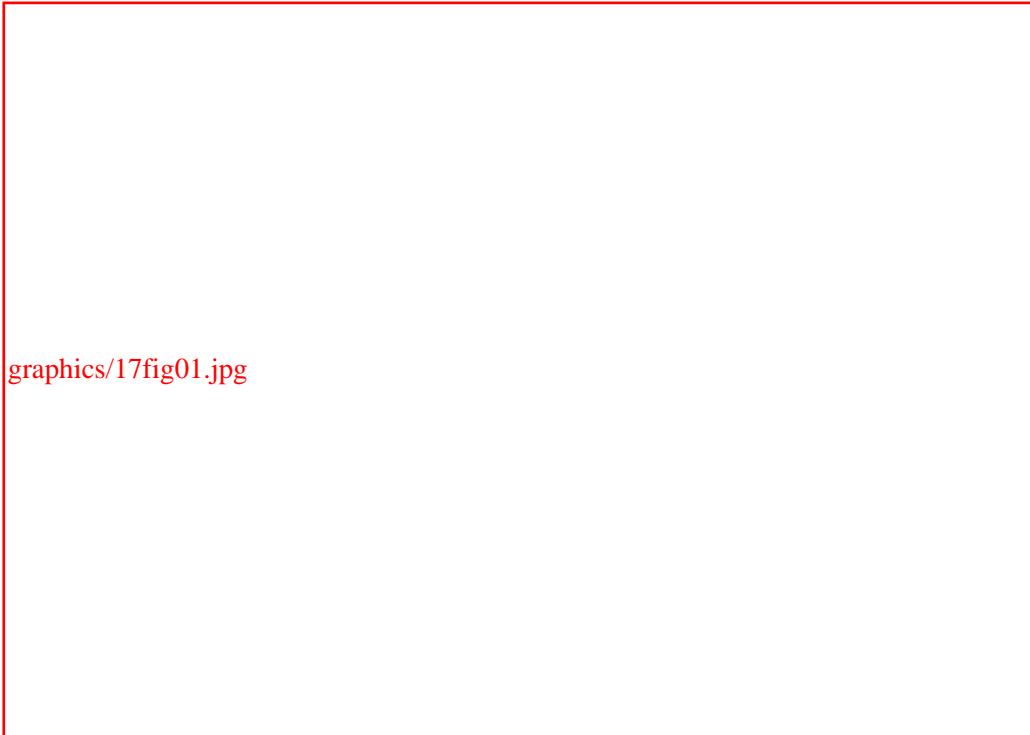
```
// Called to draw the occluding grid
void DrawOccluder(void)
{
    glColor3f(0.5f, 0.25f, 0.0f);
    glPushMatrix();
    glScalef(30.0f, 30.0f, 1.0f);
    glTranslatef(0.0f, 0.0f, 50.0f);
    glutSolidCube(10.0f);
    glTranslatef(0.0f, 0.0f, -100.0f);
    glutSolidCube(10.0f);
    glPopMatrix();
}
```

```

glPushMatrix();
glScalef(1.0f, 30.0f, 30.0f);
glTranslatef(50.0f, 0.0f, 0.0f);
glutSolidCube(10.0f);
glTranslatef(-100.0f, 0.0f, 0.0f);
glutSolidCube(10.0f);
glPopMatrix();
glPushMatrix();
glScalef(30.0f, 1.0f, 30.0f);
glTranslatef(0.0f, 50.0f, 0.0f);
glutSolidCube(10.0f);
glTranslatef(0.0f, -100.0f, 0.0f);
glutSolidCube(10.0f);
glPopMatrix();
}

```

Figure 17.1. Our main occluder is a grid constructed out of six walls.



In each grid compartment, we're going to put a highly tessellated textured sphere. These spheres are our "occludees," objects possibly hidden by the occluder. We need the high vertex count and texturing to accentuate the rendering burden so that we can subsequently relieve that burden courtesy of occlusion queries. Just as we did in the preceding chapter where we were showing off our buffer objects, we need to lay on the vertices!

Figure 17.2 shows the picture resulting from Listing 17.2. If you find this workload too heavy, feel free to reduce the tessellation in `glutSolidSphere` from the 100s to smaller numbers. Or if your OpenGL implementation is still hungry for more, go ahead and increase the tessellation.

Listing 17.2. Drawing 27 Highly Tessellated Spheres in a Color Cube

```

// Called to draw sphere
void DrawSphere(GLint sphereNum)
{
    ...
    glutSolidSphere(50.0f, 100, 100);
}

```

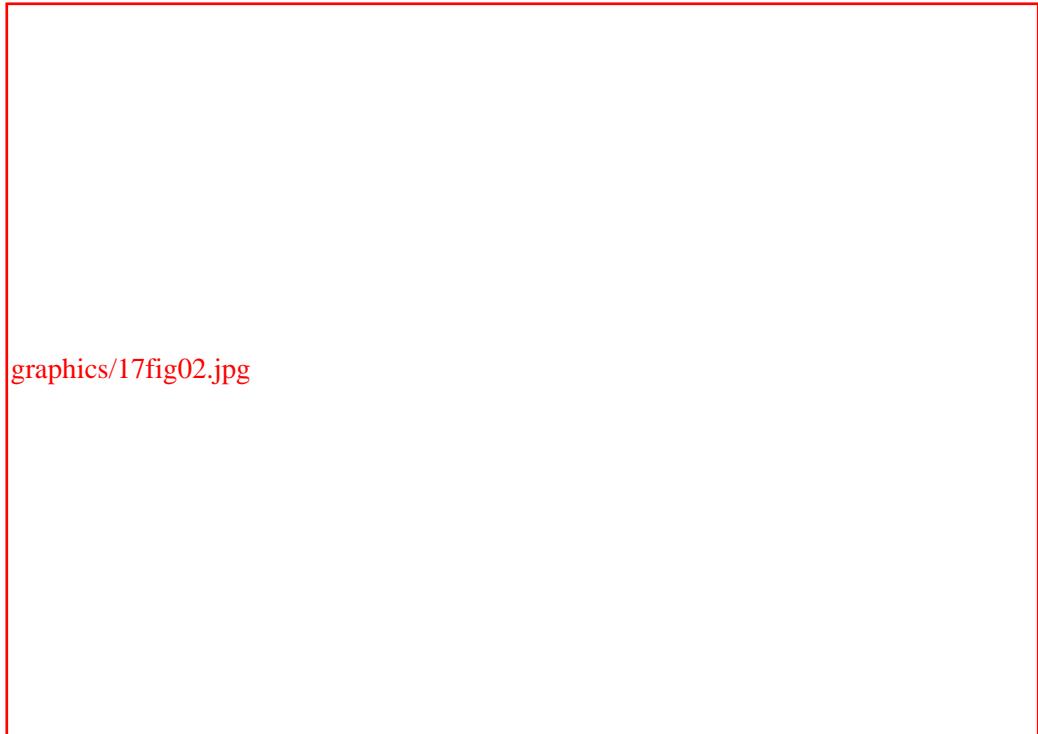
```

    ...
}

void DrawModels(void)
{
    ...
    // Turn on texturing just for spheres
    glEnable(GL_TEXTURE_2D);
    glEnable(GL_TEXTURE_GEN_S);
    glEnable(GL_TEXTURE_GEN_T);
    // Draw 27 spheres in a color cube
    for (r = 0; r < 3; r++)
    {
        for (g = 0; g < 3; g++)
        {
            for (b = 0; b < 3; b++)
            {
                glColor3f(r * 0.5f, g * 0.5f, b * 0.5f);
                glPushMatrix();
                glTranslatef(100.0f * r - 100.0f,
                            100.0f * g - 100.0f,
                            100.0f * b - 100.0f);
                DrawSphere((r*9)+(g*3)+b);
                glPopMatrix();
            }
        }
    }
    glDisable(GL_TEXTURE_2D);
    glDisable(GL_TEXTURE_GEN_S);
    glDisable(GL_TEXTURE_GEN_T);
}

```

Figure 17.2. Twenty-seven high-detail spheres will act as our occludees.



[Listing 17.2](#) marks the completion of our picture. If we were happy with the rendering performance, we could end the chapter right here. But if the sphere tessellation is cranked up high enough, or if you were to introduce complex multitexturing or fragment shading to the sphere

rendering, frame rates should be unacceptable. So read on!

Bounding Boxes

The theory behind occlusion detection is that if an object's *bounding volume* is not visible, neither is the object. A bounding volume is any volume that completely contains the object. The whole point of occlusion detection is to cheaply draw a simple bounding volume to find out whether you can avoid drawing the actual complex object. So the more complex our bounding volume is, the more it negates the purpose of the optimization we're trying to create.

The simplest bounding volume is a cube, also called a *bounding box*. Eight vertices, six faces. You can easily create a bounding box for any object just by scanning for its minimum and maximum coordinates on each of the x-, y-, and z-axes. For our spheres with a 50-unit radius, a bounding box with sides of length 100 units will fit perfectly.

Be aware of the trade-off when using such a simple and arbitrary bounding volume. The bounding volume may have very few vertices, but it will touch many more pixels than the original object would have. With a few additional strategically placed vertices, you can turn your bounding box into a more useful shape and significantly reduce the fill rate overhead. Fortunately, the bounding box is drawn without any fancy texturing or shading, so its overall fill rate cost will often be less than the original object anyway. [Figure 17.3](#) shows an example of how different bounding volume shapes affect pixel count and vertex count.

Figure 17.3. Various bounding volumes with their pros and cons.



When we draw our bounding volumes, we're going to enable an occlusion query that will count the number of fragments that pass the depth test (and the stencil test if enabled). Therefore, we don't care how the bounding volumes look. In fact, we don't even need to draw them to the screen at all. So we'll shut off all the bells and whistles before rendering the bounding volume, including writes to the color buffer:

```
glShadeModel(GL_FLAT);
glDisable(GL_LIGHTING);
glDisable(GL_COLOR_MATERIAL);
glDisable(GL_NORMALIZE);
// Texturing is already disabled
...
glColorMask(0, 0, 0, 0);
```

After all this talk about occlusion queries, we're finally going to create some. But first, we need to come up with names for them. Here, we request 27 names, one for each sphere's query, and we provide a pointer to the array where the new names should be stored:

```
// Generate occlusion query names
glGenQueries(27, queryIDs);
```

When we're done with them, we delete the query objects, indicating there are 27 names to be deleted in the provided array:

```
glDeleteQueries(27, queryIDs);
```

Occlusion query objects are not bound like other OpenGL objects, such as texture objects and buffer objects. Instead, they're created by calling `glBeginQuery`. This marks the beginning of our query. The query object has an internal counter that keeps track of the number of fragments that would make it to the framebuffer—if we hadn't shut off the color buffer's write mask. Beginning the query resets this counter to zero to start a fresh query.

Then we draw our bounding volume. The query object's internal counter is incremented every time a fragment passes the depth test, and thus is not hidden by our main occluder, the grid which we've already drawn. For some algorithms, it's useful to know exactly how many fragments were drawn, but for our purposes here, all we care about is whether the counter is zero or nonzero. This value corresponds to whether any part of the bounding volume is visible or if all fragments were discarded by the depth test.

When we're finished drawing our bounding volume, we mark the end of our query by calling `glEndQuery`. This tells OpenGL we're done with this query and lets us continue with another query or ask for the result back. Because we're drawing 27 spheres, we can improve the performance by using 27 different query objects. This way, we can queue up the drawing of all 27 bounding volumes without disrupting the pipeline by reading back the query results in between.

[Listing 17.3](#) illustrates the rendering of our bounding volumes, bracketed by the beginning and ending of our query. Then we proceed to redraw the main occluder and possibly draw the actual spheres. Notice the code for visualizing the bounding volume whereby we leave the color buffer's write mask enabled. This way, we can see and compare the different bounding volume shapes.

Listing 17.3. Beginning the Query, Drawing the Bounding Volume, Ending the Query, Then Moving on to Redraw the Actual Scene

```
// Called to draw scene objects
void DrawModels(void)
{
    GLint r, g, b;
    if (occlusionDetection || showBoundingVolume)
    {
        // Draw bounding boxes after drawing the main occluder
        DrawOccluder();
        // All we care about for bounding box is resulting depth values
        glShadeModel(GL_FLAT);
        glDisable(GL_LIGHTING);
        glDisable(GL_COLOR_MATERIAL);
        glDisable(GL_NORMALIZE);
        // Texturing is already disabled
        if (!showBoundingVolume)
            glColorMask(0, 0, 0, 0);
        // Draw 27 spheres in a color cube
        for (r = 0; r < 3; r++)
        {
            for (g = 0; g < 3; g++)
            {
                for (b = 0; b < 3; b++)
                {
                    if (showBoundingVolume)
                        glColor3f(r * 0.5f, g * 0.5f, b * 0.5f);
                }
            }
        }
    }
}
```

```

        glPushMatrix();
        glTranslatef(100.0f * r - 100.0f,
                     100.0f * g - 100.0f,
                     100.0f * b - 100.0f);
        glBeginQuery(GL_SAMPLES_PASSED, queryIDs[(r*9)+(g*3)+b]);
        switch (boundingVolume)
        {
        case 0:
            glutSolidCube(100.0f);
            break;
        case 1:
            glScalef(150.0f, 150.0f, 150.0f);
            glutSolidTetrahedron();
            break;
        case 2:
            glScalef(90.0f, 90.0f, 90.0f);
            glutSolidOctahedron();
            break;
        case 3:
            glScalef(40.0f, 40.0f, 40.0f);
            glutSolidDodecahedron();
            break;
        case 4:
            glScalef(65.0f, 65.0f, 65.0f);
            glutSolidIcosahedron();
            break;
        }
        glEndQuery(GL_SAMPLES_PASSED);
        glPopMatrix();
    }
}
if (!showBoundingVolume)
    glClear(GL_DEPTH_BUFFER_BIT);
// Restore normal drawing state
glShadeModel(GL_SMOOTH);
glEnable(GL_LIGHTING);
glEnable(GL_COLOR_MATERIAL);
glEnable(GL_NORMALIZE);
glColorMask(1, 1, 1, 1);
}
DrawOccluder();
// Turn on texturing just for spheres
glEnable(GL_TEXTURE_2D);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
// Draw 27 spheres in a color cube
for (r = 0; r < 3; r++)
{
    for (g = 0; g < 3; g++)
    {
        for (b = 0; b < 3; b++)
        {
            glColor3f(r * 0.5f, g * 0.5f, b * 0.5f);
            glPushMatrix();
            glTranslatef(100.0f * r - 100.0f,
                        100.0f * g - 100.0f,
                        100.0f * b - 100.0f);
            DrawSphere((r*9)+(g*3)+b);
            glPopMatrix();
        }
    }
}

```

```

        }
    }
    glDisable(GL_TEXTURE_2D);
    glDisable(GL_TEXTURE_GEN_S);
    glDisable(GL_TEXTURE_GEN_T);
}

```

`DrawSphere` contains the magic where we decide whether to actually draw the sphere. Our query results are waiting for us inside our 27 query objects. Let's find out which are hidden and which we have to draw.

Querying the Query Object

The moment of truth is here. The jury is back with its verdict. We want to draw as little as possible, so we're hoping each and every one of our queries resulted in no fragments being touched. But if you think about this grid of spheres, you know that's not going to happen.

No matter what angle we're looking at our grid, unless we zoom way in, there will always be at least 9 spheres in view. Worst case is you'll see all the spheres on three faces of our grid: 19 spheres. Still, in that worst case, we save ourselves from drawing 8 spheres. That's almost a 30% savings in per-vertex costs. Best case, we save 66%, skipping 18 spheres. If we zoom in on one sphere, we could conceivably avoid drawing 26 spheres!

So how do you determine your luck? You simply query the query object. That sounds confusing, but this is a regular old query for OpenGL state. It just happens to be from something called a *query object*. In [Listing 17.4](#), we call `glGetQueryObjectiv` to see whether the pass counter is zero, in which case we won't draw the sphere.

Listing 17.4. Checking the Query Results and Drawing the Sphere Only If We Have To

```

// Called to draw sphere
void DrawSphere(GLint sphereNum)
{
    GLboolean occluded = GL_FALSE;
    if (occlusionDetection)
    {
        GLint passingSamples;
        // Check if this sphere would be occluded
        glGetQueryObjectiv(queryIDs[sphereNum], GL_QUERY_RESULT,
                           &passingSamples);
        if (passingSamples == 0)
            occluded = GL_TRUE;
    }
    if (!occluded)
    {
        glutSolidSphere(50.0f, 100, 100);
    }
}

```

That's all there is to it. Each sphere's query is checked in turn, and we decide whether to draw the sphere. We've included a mode where we can disable the occlusion detection to see how badly our performance suffers. Depending on how many spheres are visible, you may see a boost of two times or more thanks to occlusion detection.

In addition to the query result, you can also query to find out whether the result is immediately available. If we didn't render the 27 bounding volumes back to back, and instead asked for each result immediately, the bounding box rendering might still have been in the pipeline and the result may not have been ready yet. You can query `GL_QUERY_RESULT_AVAILABLE` to find out whether

the result is ready. If it's not, querying `GL_QUERY_RESULT` will stall until the result is available. So instead of stalling, you could find something useful for your application to do while you wait for the results to be ready. In our case, we planned ahead to do a bunch of work in between to be certain our first query result would be ready by the time we finished our 27th query.

Other state queries include the currently active query name (which query is in the middle of a `glBeginQuery`/`glEndQuery`, if any) and the number of bits in the implementation's pass counter. An implementation is allowed to advertise a 0-bit counter, in which case occlusion queries are useless and shouldn't be used. In [Listing 17.5](#), we check for that case during an application's initialization right after checking for extension strings and entrypoints.

Listing 17.5. Ensuring Occlusion Queries Are Truly Supported

```

// Make sure required functionality is available!
version = glGetString(GL_VERSION);
if ((version[0] == '1') && (version[1] == '.') &&
    (version[2] >= '5') && (version[2] <= '9'))
{
    glVersion15 = GL_TRUE;
}
if (!glVersion15 && !gltIsExtSupported("GL_ARB_occlusion_query"))
{
    fprintf(stderr, "Neither OpenGL 1.5 nor GL_ARB_occlusion_query"
                " extension is available!\n");
    Sleep(2000);
    exit(0);
}
// Load the function pointers
if (glVersion15)
{
    glBeginQuery = gltGetExtensionPointer("glBeginQuery");
    glDeleteQueries = gltGetExtensionPointer("glDeleteQueries");
    glEndQuery = gltGetExtensionPointer("glEndQuery");
    glGenQueries = gltGetExtensionPointer("glGenQueries");
    glGetQueryiv = gltGetExtensionPointer("glGetQueryiv");
    glGetQueryObjectiv = gltGetExtensionPointer("glGetQueryObjectiv");
    glGetQueryObjectuiv = gltGetExtensionPointer("glGetQueryObjectuiv");
    glIsQuery = gltGetExtensionPointer("glIsQuery");
}
else
{
    glBeginQuery = gltGetExtensionPointer("glBeginQueryARB");
    glDeleteQueries = gltGetExtensionPointer("glDeleteQueriesARB");
    glEndQuery = gltGetExtensionPointer("glEndQueryARB");
    glGenQueries = gltGetExtensionPointer("glGenQueriesARB");
    glGetQueryiv = gltGetExtensionPointer("glGetQueryivARB");
    glGetQueryObjectiv = gltGetExtensionPointer("glGetQueryObjectivARB");
    glGetQueryObjectuiv = gltGetExtensionPointer("glGetQueryObjectuivARB");
    glIsQuery = gltGetExtensionPointer("glIsQueryARB");
}

if (!glBeginQuery || !glDeleteQueries || !glEndQuery || !glGenQueries ||
    !glGetQueryiv || !glGetQueryObjectiv || !glGetQueryObjectuiv ||
    !glIsQuery)
{
    fprintf(stderr, "Not all entrypoints were available!\n");
    Sleep(2000);
    exit(0);
}
// Make sure query counter bits is non-zero
glGetQueryiv(GL_SAMPLES_PASSED, GL_QUERY_COUNTER_BITS, &queryCounterBits);

```

```

if (queryCounterBits == 0)
{
    fprintf(stderr, "Occlusion queries not really supported!\n");
    fprintf(stderr, "Available query counter bits: 0\n");
    Sleep(2000);
    exit(0);
}

```

The only other query to be aware of is `glIsQuery`. This command just checks whether the specified name is the name of an existing query object, in which case it returns `GL_TRUE`. Otherwise, it returns `GL_FALSE`.

Summary

When rendering complex scenes, sometimes we waste hardware resources by rendering objects that will never be seen. We can try to avoid the extra work by testing whether an object will show up in the final image. By drawing a bounding box, or some other simple bounding volume, around the object, we can cheaply approximate the object in the scene. If occluders in the scene hide the bounding box, they would also hide the actual object. By wrapping the bounding box rendering with a query, we can count the number of pixels that would be hit. If the bounding box hits no pixels, we can guarantee that the original object would also not be drawn, so we can skip rendering it. Performance improvements can be dramatic, depending on the complexity of the objects in the scene and how often they are occluded.

Reference

glBeginQuery

Purpose: Marks the beginning of an occlusion query.

Include File: `<glext.h>`

Syntax:

```
void glBeginQuery(GLenum target, GLuint id);
```

Description: This function starts an occlusion query with the specified name by resetting its samples-passed counter to zero. If the query object does not yet exist, it is created. An error is thrown if a query is already in progress or if the specified query object name is zero.

Parameters:

`target` `GLenum`: The type of query object being restarted. Must be `GL_SAMPLES_PASSED`.

`id` `GLuint`: The name of the query object being restarted.

Returns: None.

See Also: `glEndQuery`, `glGenQueries`, `glDeleteQueries`, `glIsQuery`

glDeleteQueries

Purpose: Deletes one or more query objects.

Include File: <glext.h>**Syntax:**

```
void glDeleteQueries(GLsizei n, const GLuint *ids);
```

Description: This function deletes query objects. The contents are deleted, and the names are marked as unused. If an unused query object name or name zero is specified for deletion, that name is silently ignored.

Parameters:

n **GLsizei**: The number of query objects to delete.

ids **const GLuint ***: Pointer to an array containing the names of the query objects to delete.

Returns: None.

See Also: [glBeginQuery](#), [glEndQuery](#), [glGenQueries](#), [glIsQuery](#)

glEndQuery

Purpose: Marks the end of an occlusion query.

Include File: <glext.h>**Syntax:**

```
void glEndQuery(GLenum target);
```

Description: This function marks the end of the current occlusion query. If no occlusion query is in progress, an error is thrown.

Parameters:

target **GLenum**: The type of query object being ended. Must be [GL_SAMPLES_PASSED](#).

Returns: None.

See Also: [glBeginQuery](#), [glDeleteQueries](#), [glQueryObjectiv](#), [glGetQueryObjectuiv](#)

glGenQueries

Purpose: Returns unused query object names.

Include File: <glext.h>**Syntax:**

```
void glGenQueries(GLsizei n, const GLuint *ids);
```

Description: This function returns unused query object names. The query objects can subsequently be created and started with [glBeginQuery](#).

Parameters:

- n* **GLsizei**: The number of query object names to return.
- ids* **GLuint ***: Pointer to an array to fill with unused query object names.
- Returns:** None.
- See Also:** [glDeleteQueries](#), [glBeginQuery](#), [glIsQuery](#)

glGetQueryiv

Purpose: Queries properties of a query object target.

Include File: `<glext.h>`

Syntax:

```
void glGetQueryiv(GLenum target, GLenum pname,
  ➔ GLint *params);
```

Description: This function queries the state of the specified query object target. This is state not specific to a particular query object, but rather shared by all query objects.

Parameters:

target **GLenum**: The type of query object being queried. Must be `GL_SAMPLES_PASSED`.

pname **GLenum**: The query object target state being queried. It can be one of the following constants:

`GL_CURRENT_QUERY`: The name of the currently active occlusion query object. If no occlusion query object is active, zero is returned.

`GL_QUERY_COUNTER_BITS`: The implementation-dependent number of bits in the counter used to accumulate passing samples.

params **GLint ***: A pointer to the location where the query results are written.

Returns: None.

See Also: [glGetQueryObjectiv](#), [glGetQueryObjectuiv](#), [glIsQuery](#)

glGetQueryObject

Purpose: Queries properties of a query object.

Include File: `<glext.h>`

Syntax:

```
void glGetQueryObjectiv(GLuint id, GLenum pname,
    GLint *params);
void glGetQueryObjectuiv(GLuint id, GLenum pname,
    GLuint *params);
```

Description: This function queries the state of the query object with the specified name. If the name does not correspond to an existing query object, or if the query object is currently active (inside a `glBeginQuery/glEndQuery`), an error is thrown.

Parameters:

id `GLuint`: The query object name whose state is being queried.

pname `GLenum`: The query object state being queried. It can be one of the following constants:

- `GL_QUERY_RESULT`: Indicates the value of the query object's passed samples counter.
- `GL_QUERY_RESULT_AVAILABLE`: Indicates whether the above result is immediately available. If a delay would occur waiting for the query result, `GL_FALSE` is returned. Otherwise, `GL_TRUE` is returned, which also indicates that the results of all previous queries are available as well.

params `GLint */GLuint *:` A pointer to the location where the query results are written.

Returns: None.

See Also: `glBeginQuery`, `glEndQuery`, `glGetQueryiv`, `glIsQuery`

glIsQuery

Purpose: Queries whether a name is a query object name.

Include File: `<glext.h>`

Syntax:

```
GLboolean glIsQuery(GLuint id);
```

Description: This function queries whether the specified name is the name of a query object.

Parameters:

id `GLuint`: The query object name to be queried.

Returns: `GLboolean`: `GL_TRUE` is returned if a query object with this name has previously been created (with `glBeginQuery`) and not yet deleted. Otherwise, `GL_FALSE` is returned.

See Also: `glBeginQuery`, `glEndQuery`, `glGenQueries`, `glDeleteQueries`

Chapter 18. Depth Textures and Shadows

by Benjamin Lipchak

WHAT YOU'LL LEARN IN THIS CHAPTER:

How To	Functions You'll Use
Draw your scene from the light's perspective	<code>gluLookAt</code> / <code>gluPerspective</code>
Copy texels from the depth buffer into a depth texture	<code>glCopyTexImage2D</code>
Use eye linear texture coordinate generation	<code>glTexGen</code>
Set up shadow comparison	<code>glTexParameter</code>

Shadows are an important visual cue, both in reality and in rendered scenes. At a very basic level, shadows give us information about the location of objects in relation to each other and to light sources, even if the light sources are not visible in the scene. When it comes to games, shadows can make an already immersive environment downright spooky. Imagine turning the corner in a torch-lit dungeon and stepping into the shadow of your worst nightmare. Peter Pan had it easy.

In [Chapter 5](#), "Color, Materials, and Lighting: The Basics," we described a low-tech way of projecting an object onto a flat plane, in effect "squishing" it to appear as a shadow. Another technique utilizing the stencil buffer, known as shadow volumes, has been widely used, but it tends to require significant pre-processing of geometry and high fill rates to the stencil buffer. In OpenGL 1.4, a more elegant approach to shadow generation has been made possible: shadow mapping.

The theory behind shadow mapping is simple. What parts of your scene would fall in shadow? Answer: The parts that light doesn't directly hit. Think of yourself in the light's position in your virtual scene. What would the light see if it were the camera? Everything the light sees would be lit. Everything else falls in shadow. [Figure 18.1](#) will help you visualize the difference between the camera's viewpoint and the light's viewpoint.

Figure 18.1. The camera and the light have different perspectives on the scene.



When the scene is rendered from the light's perspective, the side effect is a depth buffer full of useful information. At every pixel in the resulting depth buffer, we know the relative distance from the light to the nearest surface. These surfaces are lit by the light source. Every other surface farther away from the light source remains in shadow.

What we'll do is take that depth buffer, copy it into a texture, and project it back on the scene, now rendered again from the normal camera angle. We'll use that projected texture to automatically determine what parts of what objects are in light, and which remain in shadow. Sounds easy, but each step of this technique requires careful attention.

Be That Light

Our first step is to draw the scene from the light's perspective. We'll use several built-in GLUT objects to show off how well this technique works, even when casting shadows on nonplanar surfaces, such as other objects in the scene. You can change the viewpoint by manually setting the modelview matrix, but for this example, we use the `gluLookAt` helper function to facilitate the change:

```
gluLookAt(lightPos[0], lightPos[1], lightPos[2],
          0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);
```

Fit the Scene to the Window

In addition to this modelview matrix, we also need to set up the projection matrix to maximize the scene's size in the window. Even if the light is far away from the objects in the scene, to achieve the best utilization of the space in our shadow map, we would still like the scene to fill the available space. We'll set up the near and far clipping planes based on the distance from the light to the nearest and farthest objects in the scene. Also, we'll estimate the field of view to contain the entire scene as closely as possible:

```
// Save the depth precision for where it's useful
lightToSceneDistance = sqrt(lightPos[0] * lightPos[0] +
                            lightPos[1] * lightPos[1] +
                            lightPos[2] * lightPos[2]);
```

```

nearPlane = lightToSceneDistance - 150.0f;
if (nearPlane < 50.0f)
    nearPlane = 50.0f;
// Keep the scene filling the depth texture
fieldOfView = 17000.0f / lightToSceneDistance;
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(fieldOfView, 1.0f, nearPlane, nearPlane + 300.0f);

```

No Bells or Whistles, Please

When we draw the first pass of the scene, the light's viewpoint, we don't actually want to see it. We just want to tap into the resulting depth buffer. So we'll draw to the back buffer and never bother swapping. We can further accelerate this pass by masking writes to the color buffer. And because all we care about is the depth values, we obviously don't care about lighting, smooth shading, or anything else that isn't going to affect the result. Shut it all off. All we need to specify is the raw geometry:

```

glShadeModel(GL_FLAT);
glDisable(GL_LIGHTING);
glDisable(GL_COLOR_MATERIAL);
glDisable(GL_NORMALIZE);
glColorMask(0, 0, 0, 0);

```

The output from [Listing 18.1](#) is not visible, but [Figure 18.2](#) illustrates via grayscale what the depth buffer contains.

Listing 18.1. Rendering the Light's Viewpoint into the Shadow Map

```

// Called to draw scene objects
void DrawModels(void)
{
    // Draw plane that the objects rest on
    glColor3f(0.0f, 0.0f, 0.90f); // Blue
    glNormal3f(0.0f, 1.0f, 0.0f);
    glBegin(GL_QUADS);
        glVertex3f(-100.0f, -25.0f, -100.0f);
        glVertex3f(-100.0f, -25.0f, 100.0f);
        glVertex3f(100.0f, -25.0f, 100.0f);
        glVertex3f(100.0f, -25.0f, -100.0f);
    glEnd();

    // Draw red cube
    glColor3f(1.0f, 0.0f, 0.0f);
    glutSolidCube(48.0f);
    // Draw green sphere
    glColor3f(0.0f, 1.0f, 0.0f);
    glPushMatrix();
    glTranslatef(-60.0f, 0.0f, 0.0f);
    glutSolidSphere(25.0f, 50, 50);
    glPopMatrix();
    // Draw yellow cone
    glColor3f(1.0f, 1.0f, 0.0f);
    glPushMatrix();
    glRotatef(-90.0f, 1.0f, 0.0f, 0.0f);
    glTranslatef(60.0f, 0.0f, -24.0f);
    glutSolidCone(25.0f, 50.0f, 50, 50);
    glPopMatrix();
    // Draw magenta torus
    glColor3f(1.0f, 0.0f, 1.0f);
    glPushMatrix();

```

```

glTranslatef(0.0f, 0.0f, 60.0f);
glutSolidTorus(8.0f, 16.0f, 50, 50);
glPopMatrix();
// Draw cyan octahedron
	glColor3f(0.0f, 1.0f, 1.0f);
	glPushMatrix();
	glTranslatef(0.0f, 0.0f, -60.0f);
	glScalef(25.0f, 25.0f, 25.0f);
	glutSolidOctahedron();
	glPopMatrix();
}

// Called to regenerate the shadow map
void RegenerateShadowMap(void)
{
	GLfloat lightToSceneDistance, nearPlane, fieldOfView;
	GLfloat lightModelview[16], lightProjection[16];
	// Save the depth precision for where it's useful
	lightToSceneDistance = sqrt(lightPos[0] * lightPos[0] +
	                           lightPos[1] * lightPos[1] +
	                           lightPos[2] * lightPos[2]);
	nearPlane = lightToSceneDistance - 150.0f;
	if (nearPlane < 50.0f)
		nearPlane = 50.0f;
	// Keep the scene filling the depth texture
	fieldOfView = 17000.0f / lightToSceneDistance;
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluPerspective(fieldOfView, 1.0f, nearPlane, nearPlane + 300.0f);
	glGetFloatv(GL_PROJECTION_MATRIX, lightProjection);
	// Switch to light's point of view
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
	gluLookAt(lightPos[0], lightPos[1], lightPos[2],
	          0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);
	glGetFloatv(GL_MODELVIEW_MATRIX, lightModelview);
	glViewport(0, 0, shadowSize, shadowSize);
	// Clear the depth buffer only
	glClear(GL_DEPTH_BUFFER_BIT);

	// All we care about here is resulting depth values
	glShadeModel(GL_FLAT);
	glDisable(GL_LIGHTING);
	glDisable(GL_COLOR_MATERIAL);
	glDisable(GL_NORMALIZE);
	glColorMask(0, 0, 0, 0);
	// Overcome imprecision
	glEnable(GL_POLYGON_OFFSET_FILL);
	// Draw objects in the scene
	DrawModels();
	// Copy depth values into depth texture
	glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT,
	                 0, 0, shadowSize, shadowSize, 0);
	// Restore normal drawing state
	glShadeModel(GL_SMOOTH);
	glEnable(GL_LIGHTING);
	glEnable(GL_COLOR_MATERIAL);
	glEnable(GL_NORMALIZE);
	glColorMask(1, 1, 1, 1);
	glDisable(GL_POLYGON_OFFSET_FILL);
	// Set up texture matrix for shadow map projection
	glMatrixMode(GL_TEXTURE);
}

```

```

glLoadIdentity();
glTranslatef(0.5f, 0.5f, 0.5f);
glScalef(0.5f, 0.5f, 0.5f);
glMultMatrixf(lightProjection);
glMultMatrixf(lightModelview);
}

```

Figure 18.2. If we could see the depth buffer, this is what it would look like.



A New Kind of Texture

We want to copy the depth values from the depth buffer into a texture for use as the shadow map. OpenGL allows you to copy color values directly into textures via `glCopyTexImage2D`. Until OpenGL 1.4, this capability was possible only for color values. But now *depth textures* are available.

Depth textures simply add a new type of texture data. We've had base formats with red, green, and blue color data and/or alpha, luminosity, or intensity. To this list, we now add a depth base format. The internal formats that can be requested include `GL_DEPTH_COMPONENT16`, `GL_DEPTH_COMPONENT24`, and `GL_DEPTH_COMPONENT32`, each reflecting the number of bits per texel. Typically, you'll want a format that matches the precision of your depth buffer. OpenGL makes it easy by letting you use the generic `GL_DEPTH_COMPONENT` internal format that usually adopts whichever specific format matches your depth buffer.

After drawing the light's view into the depth buffer, we want to copy that data directly into a depth texture. This saves us the trouble of using both `glReadPixels` and `glTexImage2D`:

```

glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT,
    0, 0, shadowSize, shadowSize, 0);

```

Note that [Listing 18.1](#), for drawing the light's view and regenerating the shadow map, needs to be executed only when objects in the scene move or the light source moves. If the only thing moving is the camera angle, you can keep using the same depth texture. Remember, when only the camera moves, the light's view of the scene isn't affected. (The camera is invisible.) We can reuse the existing shadow map in this case.

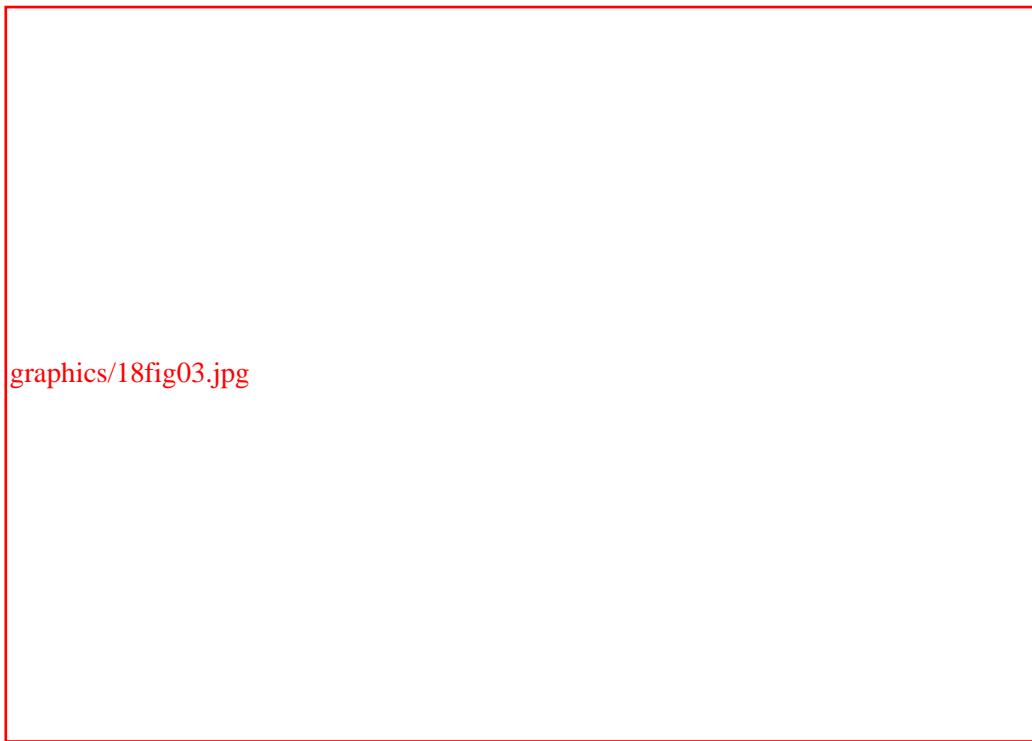
Draw the Shadows First!?

Yes, we will draw the shadows first. But, you ask, if a shadow is defined as the lack of light, why do we need to draw shadows at all? Strictly speaking, you don't need to draw them if you have a single spotlight. If you leave the shadows black, you'll achieve a stark effect that may suit your purposes well. But if you don't want pitch black shadows and still want to make out details inside the shadowed regions, you'll need to simulate some ambient lighting in your scene:

```
GLfloat lowAmbient[4] = {0.1f, 0.1f, 0.1f, 1.0f};
GLfloat lowDiffuse[4] = {0.35f, 0.35f, 0.35f, 1.0f};
glLightfv(GL_LIGHT0, GL_AMBIENT, lowAmbient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, lowDiffuse);
// Draw objects in the scene
DrawModels();
```

We've added a bit of diffuse lighting as well to help convey shape information. If you use only ambient lighting, you end up with ambiguously shaped solid-colored regions. [Figure 18.3](#) shows the scene so far, entirely in shadow.

Figure 18.3. The entire scene is in shadow before the lit areas are drawn.



Some OpenGL implementations support an extension, [GL_ARB_shadow_ambient](#), which makes this first shadow drawing pass unnecessary. In this case, both the shadowed regions and the lit regions are drawn simultaneously. More on that optimization later.

And Then There Was Light

Right now, we just have a very dimly lit scene. To make shadows, we need some brightly lit areas to contrast the existing dimly lit areas, turning them into shadows. But how do we determine which areas to light? This is key to the shadow mapping technique. After we've decided where to draw, we'll draw brighter simply by using greater lighting coefficients, twice as bright as the shadowed areas:

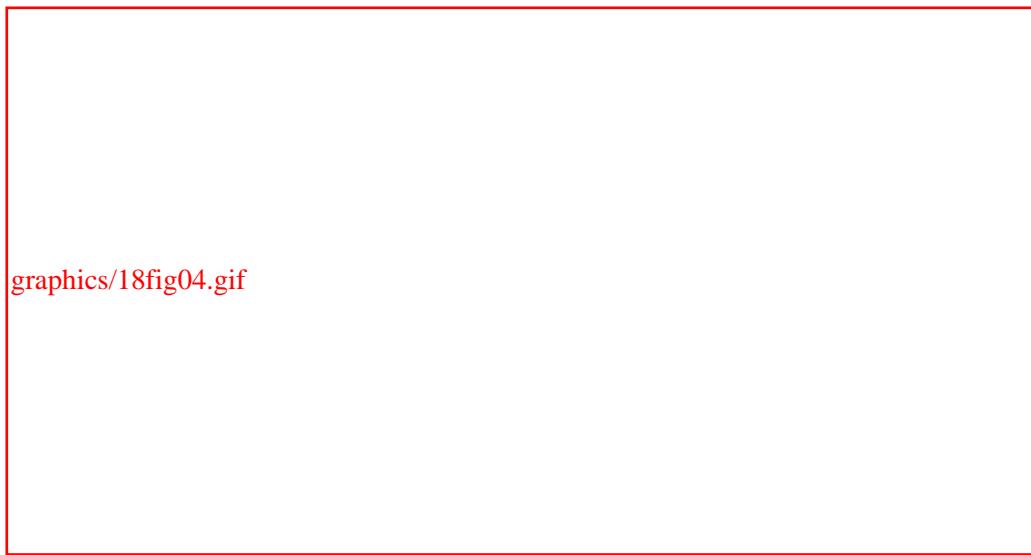
```
GLfloat ambientLight[] = { 0.2f, 0.2f, 0.2f, 1.0f};
GLfloat diffuseLight[] = { 0.7f, 0.7f, 0.7f, 1.0f};
...
glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
```

Projecting Your Shadow Map: The "Why"

Remember that texture matrix code from Listing 18.1? Now's the time to explain it. The goal here is to project the shadow map (the light's viewpoint) of the scene back onto the scene as if emitted from the light, but viewed from the camera's position. We're projecting those depth values, which represent the distance from the light to the first object hit by the light's rays. Reorienting the texture coordinates into the right coordinate space is going to take a bit of math. If you care only about the "how" and not the "why," you can safely skip over this section.

In [Chapter 4](#), "Geometric Transformations: The Pipeline," we explained the process of transforming vertices from object space to eye space, then to clip space, on to normalized device coordinates, and finally to window space. We have two different sets of matrices in play performing these transformations: one for the light view and the other for the regular camera view. [Figure 18.4](#) shows the two sets of transformations in use.

Figure 18.4. The large arrow in the center shows the transformations we need to apply to our eye linear texture coordinates.



Any texture projection usually begins with eye linear texture coordinate generation. This process will automatically generate our texture coordinates. Unlike object linear texture coordinate generation, the eye linear coordinates aren't tied to the geometry. Instead, it's as if there is a film projector casting the texture onto the scene. But it doesn't just project onto flat surfaces like a movie screen. Think about what happens when you walk in front of projector. The movie is projected on your irregularly shaped body. The same thing happens here.

We need to end up with texture coordinates that will index into our shadow map in the light's clip space. We start off with our projected eye linear texture coordinates in the camera's eye space. So

we need to first backtrack to world space and then transform to the light's eye space and finally to the light's clip space. This transformation can be summarized by the following series of matrix multiplications:

[graphics/18equ01.gif](#)

But wait, there's more. The light's clip space doesn't quite bring us home free. Remember that clip space is in the range $[-1,1]$ for each of the x , y , and z coordinates. The shadow map depth texture, like all standard 2D textures, needs to be indexed in the range $[0,1]$. Also, the depth values against which we're going to be comparing are in the range $[0,1]$, so we'll also need our z texture coordinate in that range. A simple scale by one half (S) and bias by one half (B) will do the trick:

[graphics/18equ02.gif](#)

If you're unfamiliar with OpenGL matrix notation, you're probably asking why these matrices are in reverse order. After all, we need to apply the inverse of the camera's modelview first, and the bias by one half translation is the last transformation we need. What's the deal? It's really simple, actually. OpenGL applies a matrix (M) to a coordinate (T) in a seemingly backward way, too. So, you want to read everything right to left when thinking about the order of transformations being applied to your coordinate:

[graphics/18equ03.gif](#)

This is standard representation. Nothing to see here. Move along.

Projecting Your Shadow Map: The "How"

We understand what matrix transformations need to be applied to our eye linear-generated texture coordinate to have something useful to index into our shadow map texture. But how do we apply these transformations?

Texture coordinates, like vertex positions, also are automatically subjected to matrix transformation. However, instead of both a modelview and a projection matrix, texture coordinates are multiplied by only a single texture matrix. There's nothing special to enable; the texture matrix is always applied to every texture coordinate, whether automatically generated or explicitly provided via immediate mode entrypoints, vertex arrays, and so on. You may not have known the texture matrix was at work because, by default, the identity matrix causes it to leave texture coordinates unchanged.

We'll use this texture matrix in combination with texture coordinate generation to achieve the necessary texture coordinate manipulation. To set up the texture matrix, we'll start with an identity matrix and multiply in each of our required transformations discussed in the preceding section:

```
glGetFloatv(GL_PROJECTION_MATRIX, lightProjection);
...
glGetFloatv(GL_MODELVIEW_MATRIX, lightModelview);
...
// Set up texture matrix for shadow map projection
glMatrixMode(GL_TEXTURE);
glLoadIdentity();
glTranslatef(0.5f, 0.5f, 0.5f);
glScalef(0.5f, 0.5f, 0.5f);
glMultMatrixf(lightProjection);
glMultMatrixf(lightModelview);
```

When setting our light's projection and modelview matrices before drawing the light's view, we conveniently queried and saved off these matrices so we could apply them later to the texture matrix. Our scale and bias operations to map $[-1,1]$ to $[0,1]$ are easily expressed as `glScalef` and `glTranslatef` calls.

But where's the multiplication by the inverse of the camera's modelview matrix? Glad you asked. OpenGL anticipated the need for this transformation when using eye linear texture coordinate generation. A post-multiply by the inverse of the current modelview matrix is included in the eye plane equations. All you have to do is make sure your camera's modelview is installed at the time you call `glTexGenfv`:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(cameraPos[0], cameraPos[1], cameraPos[2],
           0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);
...
GLfloat sPlane[4] = {1.0f, 0.0f, 0.0f, 0.0f};
GLfloat tPlane[4] = {0.0f, 1.0f, 0.0f, 0.0f};
GLfloat rPlane[4] = {0.0f, 0.0f, 1.0f, 0.0f};
GLfloat qPlane[4] = {0.0f, 0.0f, 0.0f, 1.0f};
...
// Set up the eye plane for projecting the shadow map on the scene
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_GEN_R);
glEnable(GL_TEXTURE_GEN_Q);
glTexGenfv(GL_S, GL_EYE_PLANE, sPlane);
glTexGenfv(GL_T, GL_EYE_PLANE, tPlane);
glTexGenfv(GL_R, GL_EYE_PLANE, rPlane);
glTexGenfv(GL_Q, GL_EYE_PLANE, qPlane);
...
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
glTexGeni(GL_Q, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
```

The Shadow Comparison

We have rendered our light view and copied it into a shadow map. We have our texture coordinates for indexing into the projected shadow map. The scene is dimly lit, ready for the real lights. The moment is near for completing our scene. We just need to combine the ingredients. First, there's some important state we can "set and forget" during initialization:

```
// Hidden surface removal
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LEQUAL);
// Set up some texture state that never changes
glGenTextures(1, &shadowTextureID);
glBindTexture(GL_TEXTURE_2D, shadowTextureID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
```

We set the depth test to less than or equal so that we can draw the lit pass on top of the dim pass. Otherwise, because the geometry is identical, the lit pass would always fail the depth test, and nothing would show up after the dimly lit shadow pass.

Then we generate and bind to our shadow map, which is the only texture used in this demo. We set our texture coordinate wrap modes to clamp. It makes no sense to repeat the projection. For example, if the light affects only a portion of the scene, but the camera is zoomed out to reveal other unlit parts of the scene, you don't want your shadow map to be repeated infinitely across

the scene. You want your texture coordinates clamped so that only the lit portion of the scene has the shadow map projected onto it.

Depth textures contain only a single source component representing the depth value. But texture environments expect four components: red, green, blue, and alpha. OpenGL gives you the flexibility as to how you want the depth mapped. Choices for the depth texture mode include `GL_ALPHA` (0,0,0,D), `GL_LUMINANCE` (D,D,D,1), and `GL_INTENSITY` (D,D,D,D). We're going to need the depth broadcast to all four channels, so we choose the intensity mode:

```
glTexParameteri(GL_TEXTURE_2D, GL_DEPTH_TEXTURE_MODE, GL_INTENSITY);
```

Obviously, we need to enable texturing to put the shadow map into effect. We set the compare mode to `GL_COMPARE_R_TO_TEXTURE`. If we don't set this, all we'll get is the depth value in the texture. But we want more than that. We want the depth value compared to our texture coordinate's R component:

```
// Set up shadow comparison
 glEnable(GL_TEXTURE_2D);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
                GL_COMPARE_R_TO_TEXTURE);
```

The R component of the texture coordinate represents the distance from the light source to an object's surface. The shadow map's depth value represents the distance from the light to the first surface it hits. By comparing one to the other, we can tell whether a surface was the first to be hit by a ray of light, or if that surface is farther away from the light, and hence is in the shadow cast by the first lit surface:

[graphics/18eau01](#)

Other comparison functions are also available. In fact, OpenGL 1.5 enables you to use all the same relational operators that you can use for depth test comparisons. `GL_LEQUAL` is the default, so we don't need to change it.

Another two settings we need to consider are the minification and magnification filters. We're going to use point sampling:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

However, some implementations may be able to smooth the edges of your shadows if you enable bilinear or trilinear filtering. On such an implementation, multiple comparisons are performed and the results are averaged. This is called *percentage-closer filtering*.

Great. We have a bunch of 0s and 1s. But we don't want to draw black and white. Now what? Easy. We just need to set up a texture environment mode, `GL_MODULATE`, that multiplies the 0s and 1s by the incoming color resulting from lighting:

```
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

Finally, we're done, right?! We have drawn our lit areas now. But wait. Where shadows appear, we just drew black over our previous ambient lighting pass. How do we preserve the ambient lighting for shadowed regions? Alpha testing will do the trick. We asked for intensity depth texture mode. Therefore, our 0s and 1s are present in the alpha component as well as the color components. Using an alpha test, we can tell OpenGL to discard any fragments in which we didn't get a 1:

```
// Enable alpha test so that shadowed fragments are discarded
 glEnable(GL_ALPHA_TEST);
 glAlphaFunc(GL_GREATER, 0.9f);
```

Okay. Now we're done. [Figure 18.5](#) shows the output from [Listing 18.2](#), shadows and all.

Listing 18.2. Drawing the Ambient Shadow and Lit Passes of the Scene

```

// Called to draw scene
void RenderScene(void)
{
    // Track camera angle
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0f, 1.0f, 1.0f, 1000.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(cameraPos[0], cameraPos[1], cameraPos[2],
              0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);
    glViewport(0, 0, windowHeight, windowHeight);
    // Track light position
    glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    if (showShadowMap)
    {
        // Display shadow map for educational purposes
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glMatrixMode(GL_TEXTURE);
        glPushMatrix();
        glLoadIdentity();
        glEnable(GL_TEXTURE_2D);
        glDisable(GL_LIGHTING);
        glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE, GL_NONE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        // Show the shadowMap at its actual size relative to window
        glBegin(GL_QUADS);
            glTexCoord2f(0.0f, 0.0f);
            glVertex2f(-1.0f, -1.0f);
            glTexCoord2f(1.0f, 0.0f);
            glVertex2f((GLfloat)shadowSize/(GLfloat)windowWidth)*2.0-1.0f,
                      -1.0f);
            glTexCoord2f(1.0f, 1.0f);
            glVertex2f((GLfloat)shadowSize/(GLfloat)windowWidth)*2.0-1.0f,
                      ((GLfloat)shadowSize/(GLfloat)windowHeight)*2.0-1.0f);
            glTexCoord2f(0.0f, 1.0f);
            glVertex2f(-1.0f,
                      ((GLfloat)shadowSize/(GLfloat)windowHeight)*2.0-1.0f);
        glEnd();
        glDisable(GL_TEXTURE_2D);
        glEnable(GL_LIGHTING);
        glPopMatrix();
        glMatrixMode(GL_PROJECTION);
        gluPerspective(45.0f, 1.0f, 1.0f, 1000.0f);
        glMatrixMode(GL_MODELVIEW);
    }
    else if (noShadows)
    {
        // Set up some simple lighting
        glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight);
    }
}

```

```

glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
// Draw objects in the scene
DrawModels();
}
else
{
    GLfloat sPlane[4] = {1.0f, 0.0f, 0.0f, 0.0f};
    GLfloat tPlane[4] = {0.0f, 1.0f, 0.0f, 0.0f};
    GLfloat rPlane[4] = {0.0f, 0.0f, 1.0f, 0.0f};
    GLfloat qPlane[4] = {0.0f, 0.0f, 0.0f, 1.0f};
    if (!ambientShadowAvailable)
    {
        GLfloat lowAmbient[4] = {0.1f, 0.1f, 0.1f, 1.0f};
        GLfloat lowDiffuse[4] = {0.35f, 0.35f, 0.35f, 1.0f};
        // Because there is no support for an "ambient"
        // shadow compare fail value, we'll have to
        // draw an ambient pass first...
        glLightfv(GL_LIGHT0, GL_AMBIENT, lowAmbient);
        glLightfv(GL_LIGHT0, GL_DIFFUSE, lowDiffuse);
        // Draw objects in the scene
        DrawModels();
        // Enable alpha test so that shadowed fragments are discarded
        glAlphaFunc(GL_GREATER, 0.9f);
        glEnable(GL_ALPHA_TEST);
    }
    glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
    // Set up shadow comparison
    glEnable(GL_TEXTURE_2D);
    glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
                    GL_COMPARE_R_TO_TEXTURE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

    // Set up the eye plane for projecting the shadow map on the scene
    glEnable(GL_TEXTURE_GEN_S);
    glEnable(GL_TEXTURE_GEN_T);
    glEnable(GL_TEXTURE_GEN_R);
    glEnable(GL_TEXTURE_GEN_Q);
    glTexGenfv(GL_S, GL_EYE_PLANE, sPlane);
    glTexGenfv(GL_T, GL_EYE_PLANE, tPlane);
    glTexGenfv(GL_R, GL_EYE_PLANE, rPlane);
    glTexGenfv(GL_Q, GL_EYE_PLANE, qPlane);
    // Draw objects in the scene
    DrawModels();
    glDisable(GL_ALPHA_TEST);
    glDisable(GL_TEXTURE_2D);
    glDisable(GL_TEXTURE_GEN_S);
    glDisable(GL_TEXTURE_GEN_T);
    glDisable(GL_TEXTURE_GEN_R);
    glDisable(GL_TEXTURE_GEN_Q);
}
if (glGetError() != GL_NO_ERROR)
    fprintf(stderr, "GL Error!\n");
// Flush drawing commands
glutSwapBuffers();
}
// This function does any needed initialization on the rendering
// context.
void SetupRC()
{

```

```

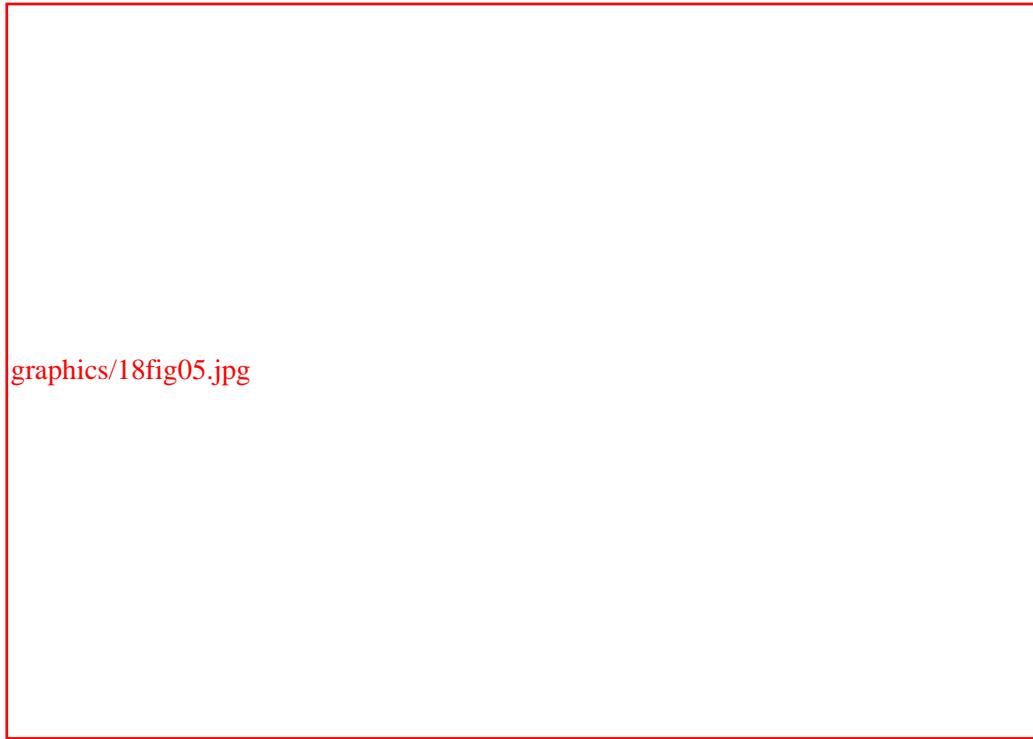
const GLubyte *version;
fprintf(stdout, "Shadow Mapping Demo\n\n");
// Make sure required functionality is available!
version = glGetString(GL_VERSION);
if (((version[0] != '1') || (version[1] != '.')) ||
    (version[2] < '4') || (version[2] > '9'))) && // 1.4+
(!glIsExtSupported("GL_ARB_shadow")))
{
    fprintf(stderr, "Neither OpenGL 1.4 nor GL_ARB_shadow"
            " extension is available!\n");
    Sleep(2000);
    exit(0);
}

// Check for optional extension
if (glIsExtSupported("GL_ARB_shadow_ambient"))
{
    ambientShadowAvailable = GL_TRUE;
}
else
{
    fprintf(stderr, "GL_ARB_shadow_ambient extension not available!\n");
    fprintf(stderr, "Extra ambient rendering pass will be required.\n\n");
    Sleep(2000);
}
fprintf(stdout, "Controls:\n");
fprintf(stdout, "\t\t\t\tControl camera movement\n");
fprintf(stdout, "\t\t\t\tControl light movement\n\n");
fprintf(stdout, "\t\t\t\tMove +/- in x direction\n");
fprintf(stdout, "\t\t\t\tMove +/- in y direction\n");
fprintf(stdout, "\t\t\t\tMove +/- in z direction\n\n");
fprintf(stdout, "\t\t\t\tChange polygon offset factor +/- \n\n");
fprintf(stdout, "\t\t\t\tToggle showing current shadow map\n");
fprintf(stdout, "\t\t\t\tToggle showing scene without shadows\n\n");
fprintf(stdout, "\t\t\t\tExit demo\n\n");
// Black background
glClearColor(0.0f, 0.0f, 0.0f, 1.0f );
// Hidden surface removal
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LEQUAL);
glPolygonOffset(factor, 0.0f);
// Set up some lighting state that never changes
glShadeModel(GL_SMOOTH);
glEnable(GL_LIGHTING);
glEnable(GL_COLOR_MATERIAL);
glEnable(GL_NORMALIZE);
glEnable(GL_LIGHT0);

// Set up some texture state that never changes
glGenTextures(1, &shadowTextureID);
glBindTexture(GL_TEXTURE_2D, shadowTextureID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_DEPTH_TEXTURE_MODE, GL_INTENSITY);
if (ambientShadowAvailable)
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FAIL_VALUE_ARB,
                    0.5f);
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
glTexGeni(GL_Q, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
RegenerateShadowMap();

```

}

Figure 18.5. A brightly lit pass is added to the previous ambient shadow pass.

Two Out of Three Ain't Bad

In [Listing 18.2](#), you'll notice code hinging on the `ambientShadowAvailable` variable. The minimum requirement for the rest of this example is OpenGL 1.4 support, or at least support for the `GL_ARB_shadow` extension. If, however, your implementation supports the `GL_ARB_shadow_ambient` extension, you can cut down the amount of work significantly.

Currently, we've described three rendering passes: one to draw the light's perspective into the shadow map, one to draw the dimly lit ambient pass, and one to draw the shadow-compared lighting. Remember, the shadow map needs to be regenerated only when the light position or objects in the scene change. So sometimes there are three passes, and other times just two. With `GL_ARB_shadow_ambient`, we can eliminate the ambient pass entirely.

Instead of 0s and 1s resulting from the shadow comparison, this extension allows us to replace another value for the 0s when the comparison fails. So if we set the fail value to a half, the shadowed regions are still halfway lit, the same amount of lighting we were previously achieving in our ambient pass:

```
if (ambientShadowAvailable)
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FAIL_VALUE_ARB,
                    0.5f);
```

This way, we also don't need to enable the alpha test.

A Few Words About Polygon Offset

Even on a surface closest to the light source, you will always discover minor differences in the floating-point values associated with the R texture coordinate and the shadow map's depth value. This can result in "self-shadowing," whereby the imprecision leads to a surface shadowing itself. You can mitigate this problem by applying a depth offset when rendering into the shadow map:

```
// Overcome imprecision
 glEnable(GL_POLYGON_OFFSET_FILL);
 ...
 glPolygonOffset(factor, 0.0f);
```

While the depth offset will help guarantee that surfaces that shouldn't be shadowed won't be, it also artificially shifts the position of shadows. A balance needs to be struck when it comes to polygon offset usage. [Figure 18.6](#) illustrates what you'll see if you don't use enough depth offset.

Figure 18.6. Shadow comparison imprecision looks a lot like depth buffer "Z-fighting."



Summary

Shadow mapping is a useful technique for achieving realistic lighting without a lot of additional processing. The light's viewpoint can be used to determine what objects are lit and which remain in shadow. Depth textures are special textures designed to store the contents of your depth buffer for use as a shadow map. Eye linear texture coordinate generation is the basis for projected textures. The texture matrix can be used to reorient the texture coordinates back into the light's clip space. Shadow comparison can be used to make the distinction between shadowed and lit regions. The [GL_ARB_shadow_ambient](#) extension can be used to reduce the number of passes that must be rendered.

Reference

glAlphaFunc

Purpose: Sets the alpha test function.

Include File: `<gl.h>`

Syntax:

```
void glAlphaFunc(GLenum func, GLclampf ref);
```

Description: This function establishes the alpha test function to be used along with a reference alpha value. When alpha testing is enabled, each fragment's alpha value is compared against this reference value according to the alpha test function's relational operator. If the comparison is not true, the fragment is discarded. This test occurs after the scissor test, but before stencil and depth testing.

Parameters:

func `GLenum`: The relational operator to use for alpha test comparisons. It can be one of the following constants:

`GL_ALWAYS`: The alpha test will always pass.

`GL_EQUAL`: The alpha test will pass if the fragment's alpha value is equal to the reference value.

`GL_GREATER`: The alpha test will pass if the fragment's alpha value is greater than or equal to the reference value.

`GL_LESS`: The alpha test will pass if the fragment's alpha value is less than or equal to the reference value.

`GL_NOTEQUAL`: The alpha test will pass if the fragment's alpha value is not equal to the reference value.

ref

`GLclampf`: The reference alpha value against which incoming alpha values are compared. This value is clamped to the range [0,1].

Returns: None.

See Also: `glDepthFunc`, `glScissor`, `glStencilFunc`

glColorMask

Purpose: Sets the writemask for sending colors to the framebuffer.

Include File: `<gl.h>`

Syntax:

```
void glColorMask(GLboolean red, GLboolean green,
    GLboolean blue, GLboolean alpha);
```

Description: This function establishes the writemask used for allowing or disallowing writes of individual color components to the framebuffer. A true value indicates that the component should be written. Otherwise, the component will remain unwritten.

Parameters:

red `GLboolean`: The writemask selector for the red color component.
green `GLboolean`: The writemask selector for the green color component.
blue `GLboolean`: The writemask selector for the blue color component.
alpha `GLboolean`: The writemask selector for the alpha color component.

Returns: None.

See Also: `glDepthMask`, `glStencilMask`

glCopyTexSubImage

Purpose: Replaces part of a texture image with pixels from the framebuffer.

Include File: `<gl.h>`

Syntax:

```
void glCopyTexSubImage1D(GLenum target, GLint
    level, GLint xoffset, GLint x, GLint y, GLsizei
    width);
void glCopyTexSubImage2D(GLenum target, GLint
    level, GLint xoffset, GLint yoffset, GLint x,
    GLint y, GLsizei width, GLsizei height);
void glCopyTexSubImage3D(GLenum target, GLint
    level, GLint xoffset, GLint yoffset, GLint zoffset
    , GLint x, GLint y, GLsizei width);
```

Description: This function copies color or depth values out of the framebuffer, replacing some or all of the texels in a previously specified texture image. If the internal format of the texture is a color format, the color values are copied from the current `GL_READ_BUFFER`. If the internal format is a depth format, depth values are copied from the depth buffer.

Parameters:

target `GLenum`: The target texture, which must be `GL_TEXTURE_1D` for `glCopyTexSubImage1D`, `GL_TEXTURE_2D` for `glCopyTexSubImage2D`, and `GL_TEXTURE_3D` for `glCopyTexSubImage3D`.

level *GLint*: The level-of-detail mipmap array to be partially or wholly replaced.

xoffset *GLint*: Offset in the x direction where replaced values will start.

yoffset *GLint*: Offset in the y direction where replaced values will start.

zoffset *GLint*: Offset in the z direction where replaced values will start.

x *GLint*: The x window coordinate of the left edge of the region to copy from the framebuffer.

y *GLint*: The y window coordinate of the lower edge of the region to copy from the framebuffer.

width *GLsizei*: The width of the region to copy from the framebuffer.

height *GLsizei*: The height of the region to copy from the framebuffer.

Returns: None.

See Also: [glCopyPixels](#), [glCopyTexImage](#), [glReadBuffer](#), [glTexImage](#), [glTexSubImage](#)

glPolygonOffset

Purpose: Sets the depth offset applied to polygons.

Include File: `<gl.h>`

Syntax:

```
void glPolygonOffset (GLfloat factor, GLfloat units);
```

Description: This function establishes the depth offset factor and units applied to polygons. The depth offset can be enabled and disabled individually for polygons in point, line, or fill mode using the tokens `GL_POLYGON_OFFSET_POINT`, `GL_POLYGON_OFFSET_LINE`, and `GL_POLYGON_OFFSET_FILL`, respectively. The depth offset adds or subtracts a single computed value for all the fragments of each polygon. If enabled, this occurs during rasterization.

Parameters:

factor *GLfloat*: A scale factor used to generate a variable depth offset by multiplying it with the maximum depth slope of the polygon.

units *GLfloat*: A constant depth offset, multiplied by the minimum granularity of the depth buffer.

Returns: None.

See Also: [glDepthFunc](#), [glPolygonMode](#)

Chapter 19. Programmable Pipeline: This Isn't Your Father's OpenGL

by Benjamin Lipchak

WHAT YOU'LL LEARN IN THIS CHAPTER:

- The responsibilities of the conventional fixed functionality OpenGL pipeline
- The pipeline stages that can be replaced by new programmable pipeline shaders
- The shader extensions that expose this new functionality

Graphics hardware has traditionally been designed to quickly perform the same rigid set of hard-coded computations. Different steps of the computation can be skipped, and parameters can be adjusted, but the computations themselves remain fixed. That's why this old paradigm of GPU design is called *fixed functionality*.

There has been a trend toward designing general-purpose graphics processors. Just like CPUs, these GPUs can be programmed with arbitrary sequences of instructions to perform virtually any imaginable computation. The biggest difference is that GPUs are tuned for the floating-point operations most common in the world of graphics.

Think of it this way: Fixed functionality is like a cookie recipe. OpenGL allows you to change the recipe a bit here and there. Change the amount of each ingredient, change the temperature of the oven. You don't want chocolate chips? Fine. Disable them. But one way or another, you end up with cookies.

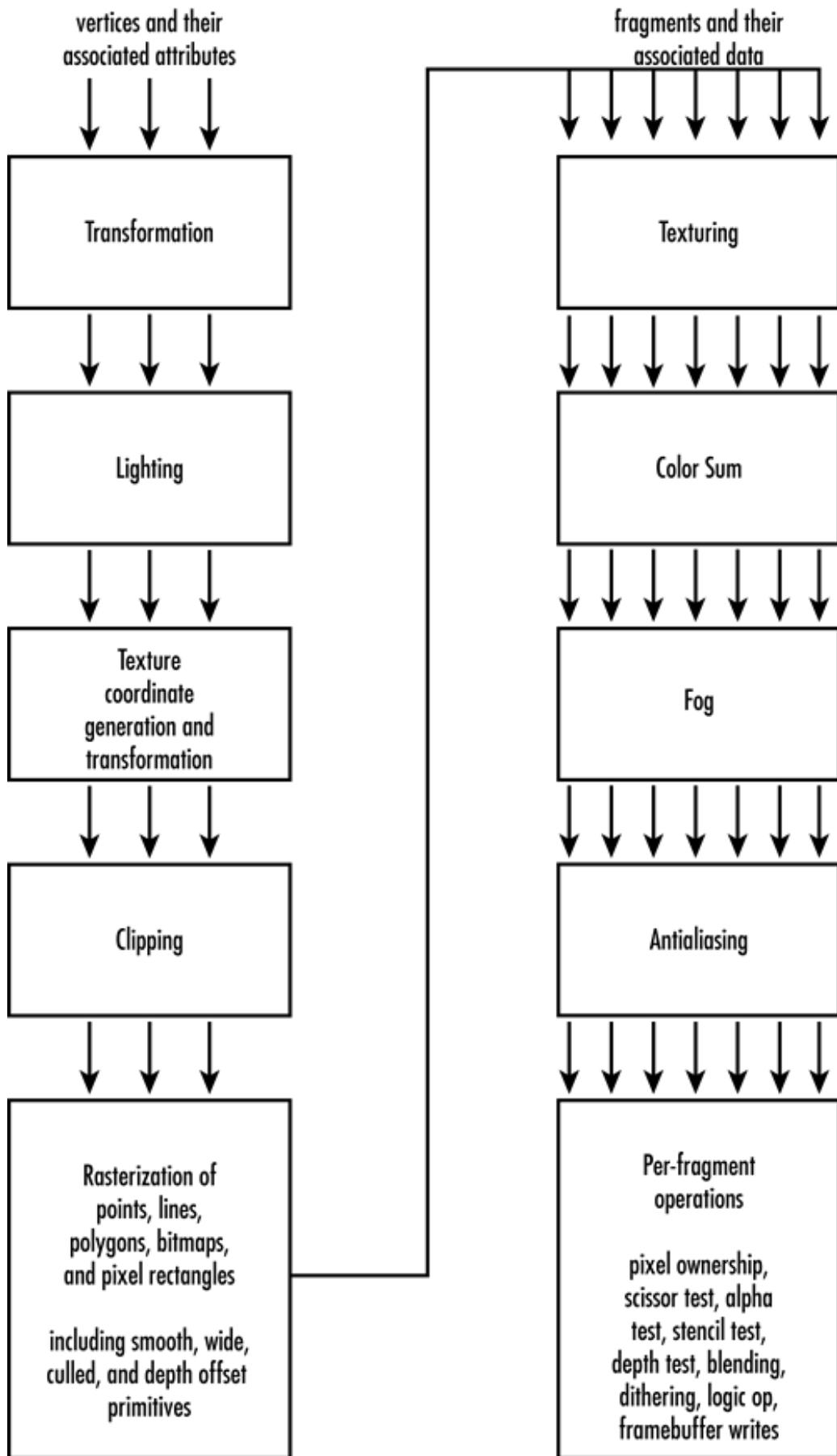
Enter programmability. Want to pick your own ingredients? Fine. Want to cook in a microwave or a frying pan or on the grill? Have it your way. Instead of cookies, you can bake a cake or grill sirloin or heat up leftovers. The possibilities are endless. The entire kitchen and all its ingredients, appliances, pots, and pans are at your disposal. These are the inputs and outputs, instruction set, and temporary register storage of a programmable pipeline stage.

In this chapter, we cover the conventional OpenGL pipeline and then describe the parts of it that can be replaced by programmable stages.

Out with the Old

Before we talk about replacing it, let's consider the conventional OpenGL rendering pipeline. The first several stages operate per-vertex. Then the primitive is rasterized to produce fragments. Finally, fragments are textured, fogged, and other per-fragment operations are applied before writing each fragment to the framebuffer. [Figure 19.1](#) diagrams the fixed functionality pipeline.

Figure 19.1. This fixed functionality rendering pipeline represents the old way of doing things.



The per-vertex and per-fragment stages of the pipeline are discussed separately in the following sections.

Fixed Vertex Processing

The per-vertex stages start with a set of vertex attributes as input. These attributes include object-space position, normal, primary and secondary colors, a fog coordinate, and texture coordinates. The final result of per-vertex processing is clip-space position, front-facing and back-facing primary and secondary colors, a fog coordinate, texture coordinates, and point size. What happens in between is broken into four stages.

Vertex Transformation

In fixed functionality, the vertex position is transformed from object space to clip space. This is achieved by multiplying the object space coordinate first by the modelview matrix to put it into eye space. Then it's multiplied by the projection matrix to reach clip space.

The application has control over the contents of the two matrices, but these matrix multiplications always occur. The only way to "skip" this stage would be to load identity matrices, so you end up with the same position you started with.

Each vertex's normal is also transformed, this time from object space to eye space for use during lighting. The normal is multiplied by the inverse of the modelview matrix, after which it is optionally rescaled or normalized. Lighting wants the normal to be a unit vector, so unless you're passing in unit length normal vectors and have a modelview matrix that leaves them unit length, you'll need to either rescale them (if your modelview introduced only uniform scaling) or fully normalize them.

[Chapters 4](#), "Geometric Transformations: The Pipeline," and [5](#), "Color, Materials, and Lighting: The Basics," covered transformations and normals.

Lighting

Lighting takes the vertex color, normal, and position as its raw data inputs. Its output is two colors, primary and secondary, and in some cases a different set of colors for front and back faces. Controlling this stage are the color material properties, light properties, and a variety of `glEnable`/`glDisable` toggles.

Lighting is highly configurable; you can enable some number of lights (up to eight or more), each with myriad parameters such as position, color, and type. You can specify material properties to simulate different surface appearances. You also can enable two-sided lighting to generate different colors for front- and back-facing polygons.

You can skip lighting entirely by disabling it. However, when it is enabled, the same hard-coded equations are always used. See [Chapters 5](#) and [6](#), "More on Colors and Materials," for a refresher on fixed functionality lighting details.

Texture Coordinate Generation and Transformation

The final per-vertex stage of the fixed functionality pipeline involves processing the texture coordinates. Each texture coordinate can optionally be generated automatically by OpenGL. There are several choices of generation equations to use. In fact, a different mode can be chosen for each component of each texture coordinate. Or, if generation is disabled, the current texture coordinate associated with the vertex is used instead.

Whether or not texture generation is enabled, each texture coordinate is always transformed by its texture matrix. If it's an identity matrix, the texture coordinate is not affected.

This texture coordinate processing stage is covered in [Chapters 8](#), "Texture Mapping: The Basics," and [9](#), "Texture Mapping: Beyond the Basics."

Clipping

If any of the vertices transformed in the preceding sections happen to fall outside the view volume, clipping must occur. Clipped vertices are discarded, and depending on the type of primitive being drawn, new vertices may be generated at the intersection of the primitive and the view volume. Colors, texture coordinates, and other vertex properties are assigned to the newly generated vertices by interpolating their values along the clipped edge. [Figure 19.2](#) illustrates a clipped primitive.

Figure 19.2. All three of this triangle's vertices are clipped out, but six new vertices are introduced.



The application may also enable user clip planes. These clip planes further restrict the clip volume so that even primitives within the view volume can be clipped. This technique is often used in medical imaging to "cut" into a volume of, for example, MRI data to inspect tissues deep within the body.

Fixed Fragment Processing

The per-fragment stages start out with a fragment and its associated data as input. This associated data is composed of various values interpolated across the line or triangle, including one or more texture coordinates, primary and secondary colors, and a fog coordinate. The result of per-fragment processing is a single color that will be passed along to subsequent per-fragment operations, including depth test and blending. Again, four stages of processing are applied.

Texture Application and Environment

Texture application is the most important per-fragment stage. Here, you take all the fragment's texture coordinates and its primary color as input. The output will be a new primary color. How this happens is influenced by which texture units are enabled for texturing, which texture images are bound to those units, and what texture function is set up by the texture environment.

For each enabled texture unit, the 1D, 2D, 3D, or cube map texture bound to that unit is used as the source for a lookup. Depending on the format of the texture and the texture function specified on that unit, the result of the texture lookup will either replace or be blended with the fragment's primary color. The resulting color from each enabled texture unit is then fed in as a color input to the next enabled texture unit. The result from the last enabled texture unit is the final output for the texturing stage.

Many configurable parameters affect the texture lookup, including texture coordinate wrap modes, border colors, minification and magnification filters, level-of-detail clamps and biases, depth texture and shadow compare state, and whether mipmap chains are automatically generated. Fixed functionality texturing was covered in detail in [Chapters 8 and 9](#).

Color Sum

The color sum stage starts with two inputs: a primary and a secondary color. The output is a single color. There's not a lot of magic here. If color sum is enabled, or if lighting is enabled, the primary and secondary colors' red, green, and blue channels are added together and then clamped back into the range $[0,1]$. If color sum is not enabled, the primary color is passed through as the result. The alpha channel of the result always comes from the primary color's alpha. The secondary color's alpha is never used by the fixed functionality pipeline.

Fog Application

If fog is enabled, the fragment's color is blended with a constant fog color based on a computed fog factor. That factor is computed according to one of three hard-coded equations: linear, exponential, or second-order exponential. These equations base the fog factor on the current fog coordinate, which may be the approximate distance from the vertex to the eye, or an arbitrary value set per-vertex by the application.

For more details on fixed functionality fog, see [Chapter 6](#).

Antialiasing Application

Finally, if the fragment belongs to a primitive that has smoothing enabled, one piece of associated data is a coverage value. That value is 1.0 in most cases, but for fragments on the edge of a smooth point, line, or polygon, the coverage is somewhere between 0.0 and 1.0. The fragment's alpha value is multiplied by this coverage value, which will with subsequent blending produce smooth edges for these primitives. [Chapter 6](#) discussed this behavior.

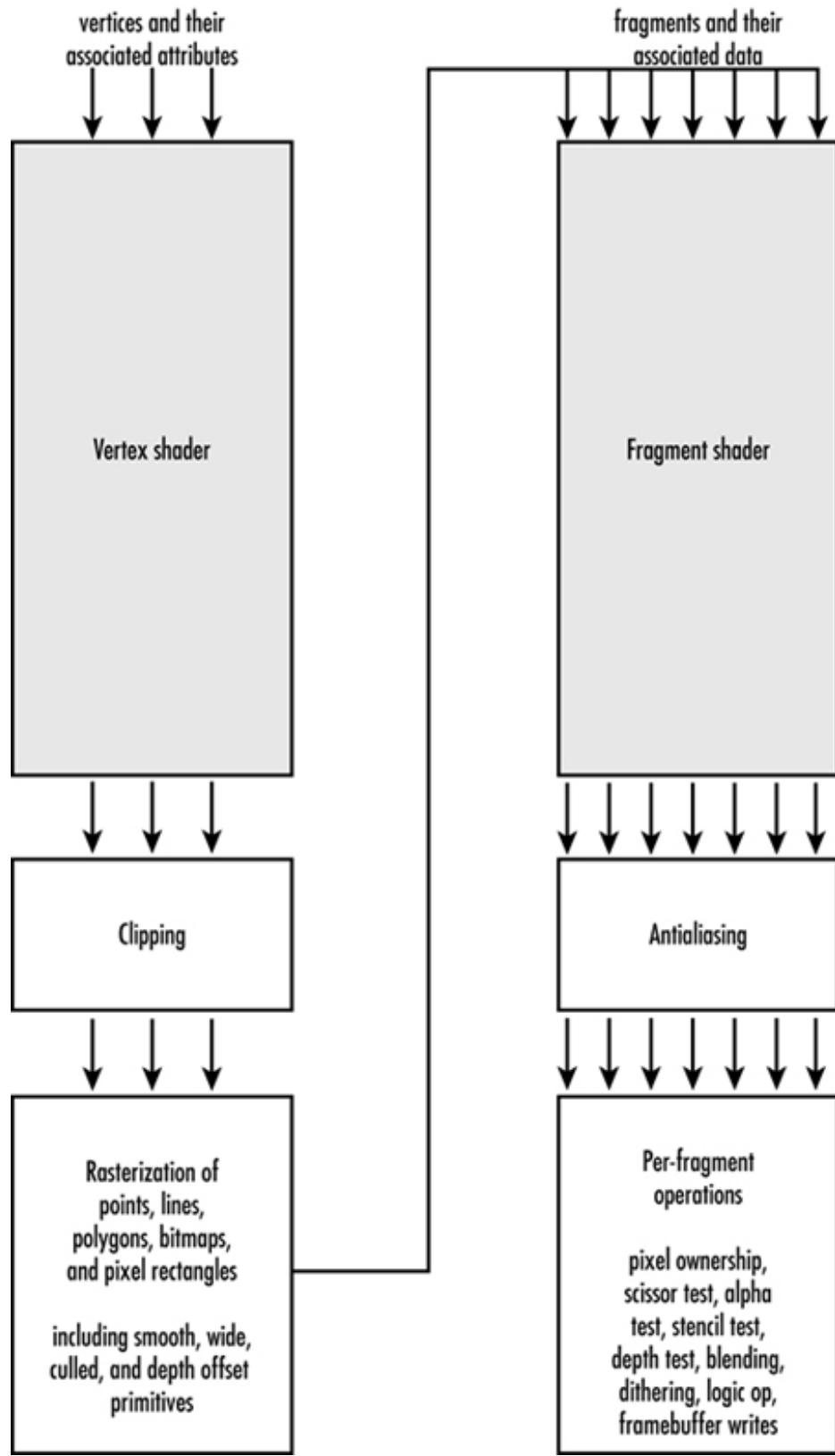
In with the New

That trip down memory lane was intended to both refresh your memory on the various stages of the current pipeline and to give you an appreciation of the configurable but hard-coded computations that happen each step of the way. Now forget everything you just read. We're going to replace the majority of it and roll in the new world order: shaders.

Shaders are also sometimes called *programs*, and the terms are usually interchangeable. And that's what shaders are—application-defined customized programs that take over the responsibilities of fixed functionality pipeline stages. I prefer the term *shader* because it avoids confusion with the typical definition of *program*, which can mean any old application.

[Figure 19.3](#) illustrates the simplified pipeline where previously hard-coded stages are subsumed by custom programmable shaders.

Figure 19.3. The block diagram looks simpler, but in reality these shaders can do everything the original fixed stages could do, plus more.



Programmable Vertex Shaders

As suggested by Figure 19.3, the inputs and outputs of a vertex shader remain the same as those of the fixed functionality stages being replaced. The raw vertices and all their attributes are fed into the vertex shader, rather than the fixed transformation stage. Out the other side, the vertex shader spits texture coordinates, colors, point size, and a fog coordinate, which are passed along to the clipper, just like the output from the fixed functionality lighting stage. A vertex shader is a drop-in replacement for those three per-vertex stages.

Replacing Vertex Transformation

What you do in your vertex shader is entirely up to you. The absolute minimum (if you want anything to draw) would be to output a clip-space vertex position. Every other output is optional and at your sole discretion. How you generate your clip-space vertex position is your call. Traditionally, and to emulate fixed functionality transformation, you would want to multiply your input position by the modelview and projection matrices to get your clip-space output.

But say you have a fixed projection and you're sending in your vertices already in clip space. In that case, you don't need to do any transformation. Just copy the input position to the output position. Or, on the other hand, maybe you want to turn your Cartesian coordinates into polar coordinates. You could add extra instructions to your vertex shader to perform those computations.

Replacing Lighting

If you don't care what the vertex's colors are, you don't have to perform any lighting computations. You can just copy the color inputs to the color outputs, or if you know the colors will never be used later, you don't have to output them at all, and they will become undefined. Beware, if you do try to use them later after not outputting them from the vertex shader, undefined usually means garbage!

If you do want to generate more interesting colors, you have limitless ways of going about it. You could emulate fixed functionality lighting by adding instructions that perform these conventional computations, maybe customizing them here or there. You could also color your vertices based on their positions, their surface normals, or any other input vector.

Replacing Texture Coordinate Processing

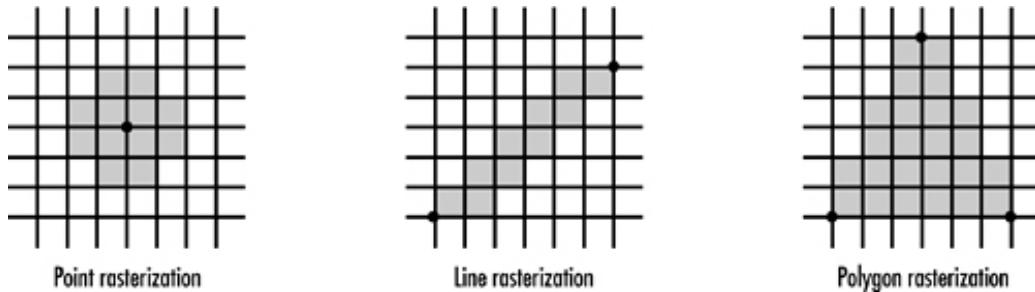
If you don't need texture coordinate generation, you don't need to code it into your vertex shader. The same goes for texture coordinate transformation. If you don't need it, don't waste precious shader cycles implementing it. You can just copy your input texture coordinates to their output counterparts. Or, as with colors, if you won't use the texture coordinate later, don't waste your time outputting it at all. For example, if your graphics card supports eight texture units, but you're going to use only three of them for texturing later in the pipeline, there's no point in outputting the other five. Doing so would just consume resources unnecessarily.

You understand the input and output interfaces of vertex shaders, largely the same as their fixed functionality counterparts. But there's been a lot of hand waving about adding code to perform the desired computations within the shader. This would be a great place for an example of a vertex shader, wouldn't it? Alas, this chapter covers only the what, where, and why of shaders. The next four chapters are devoted to the how, so you'll have to be patient and use your imagination. Consider this the calm before the storm. In a few pages, you'll be staring at more shaders than you ever hoped to see.

Fixed Functionality Glue

In between the vertex shader and fragment shader, there remains a couple of fixed functionality stages that act as glue between the two shaders. One of them is the clipping stage described previously, which clips the current primitive against the view volume and in so doing possibly adds or removes vertices. After clipping, the perspective divide by W occurs, yielding normalized device coordinates. These coordinates are subjected to viewport transformation and depth range transformation, which yield the final window-space coordinates. Then it's on to rasterization.

Rasterization is the fixed functionality stage responsible for taking the processed vertices of a primitive and turning them into fragments. Whether a point, line, or polygon primitive, this stage produces the fragments to "fill in" the primitive and interpolates all the colors and texture coordinates so that the appropriate values are assigned to each fragment. [Figure 19.4](#) illustrates this process.

Figure 19.4. Rasterization turns vertices into fragments.

Depending on how far apart the vertices of a primitive are, the ratio of fragments to vertices tends to be relatively high. For a highly tessellated object, though, you might find all three vertices of a triangle mapping to the same single fragment. As a general rule, significantly more fragments are processed than vertices, but as with all rules, there are exceptions.

Rasterization is also responsible for making lines the desired width and points the desired size. It may apply stipple patterns to lines and polygons. It generates partial coverage values at the edges of smooth points, lines, and polygons, which later are multiplied into the fragment's alpha value during antialiasing application. If requested, rasterization culls out front- or back-facing polygons and applies depth offsets.

In addition to points, lines, and polygons, rasterization also generates the fragments for bitmaps and pixel rectangles (drawn with `glDrawPixels`). But these primitives don't originate from normal vertices. Instead, where interpolated data is usually assigned to fragments, those values are adopted from the current raster position. See [Chapter 7, "Imaging with OpenGL,"](#) for more details on this subject.

Programmable Fragment Shaders

The same texture coordinates, fog coordinate, and colors are available to the fragment shader as were previously available to the fixed functionality texturing stage. The same single color output is expected out of the fragment shader that was previously expected from the fixed functionality fog stage. Just as with vertex shaders, you may choose your own adventure in between the input interface and output interface.

Replacing Texturing

The single most important capability of a fragment shader is performing texture lookups. For the most part, these texture lookups are unchanged from fixed functionality in that most of the texture state is set up outside the fragment shader. The texture image is specified and all its parameters are set the same as though you weren't using a fragment shader. The main difference is that you decide within the shader when and if to perform a lookup and what to use as the texture coordinate.

You're not limited to using texture coordinate 0 to index into texture image 0. You can mix and match coordinates with different textures, using the same texture with different coordinates or the same coordinate with different textures. Or you can even compute a texture coordinate on the fly within the shader. This flexibility was impossible with fixed functionality.

The texture environment previously included a texture function that determined how the incoming fragment color was mixed with the texture lookup results. That function is now ignored, and it's up to the shader to combine colors with texture results. In fact, you might choose to perform no texture lookups at all and rely only on other computations to generate the final color result. A fragment shader could simply copy its primary color input to its color output and call it a day. Not very interesting, but such a "passthrough" shader might be all you need when combined with a fancy vertex shader.

Replacing Color Sum

Replacing the color sum is simple. This stage just adds together the primary and secondary colors. If that's what you want to happen, you just add an instruction to do that. If you're not using the secondary color for anything, ignore it.

Replacing Fog

Fog application is not as easy to emulate as color sum, but it's still reasonably easy. First, you need to calculate the fog factor, which is an equation based on the fragment's fog coordinate and some constant value such as density. Fixed functionality dictated the use of linear, exponential, or second-order exponential equations, but with shaders you can make up your own equation. Then you blend in a constant fog color with the fragment's unfogged color, using the fog factor to determine how much of each goes into the blend. You can achieve all this in just a handful of instructions. Or you can *not* add any instructions and forget about fog. The choice is yours.

Introduction to Shader Extensions

Enough with the hypotheticals. If you've made it this far, you must have worked up an appetite for some real shaders by now. In the following sections, we introduce the different OpenGL extensions that expose programmable shaders. These extensions were developed and approved by the OpenGL Architecture Review Board (ARB), and as such are widely supported by graphics card vendors throughout the industry.

Low-Level Extensions

The low-level extensions are `GL_ARB_vertex_program` and `GL_ARB_fragment_program`, used for replacing the fixed functionality vertex stages and fragment stages, respectively.

Much like assembly language versus C, this first set of extensions operates at a low level, giving more direct access to the features and resources of the GPU. As with assembly language, you make the trade-off between programming at a more cumbersome, detail-oriented, and often complex level in exchange for the full control of the hardware and improved performance.

Strictly speaking, you're not really coding at the assembly level because each hardware vendor has a unique GPU design, each with its own native instruction representation and instruction set. Each has its own limits on the number of registers, constants, and instructions. What you're capturing in these low-level extensions is just the lowest common denominator of functionality that's available from all vendors.

[Listings 19.1](#) and [19.2](#) are your first exposure to these low-level shaders. Consider them to be "Hello World" shaders, even though technically they don't say hello at all.

Listing 19.1. A Simple `GL_ARB_vertex_program` Vertex Shader

```
!!ARBvp1.0
# This is our Hello World vertex shader
# notice how comments are preceded by '#'
ATTRIB iPos = vertex.position;           # input position
ATTRIB iPrC = vertex.color.primary;       # input primary color
OUTPUT oPos = result.position;           # output position
OUTPUT oPrC = result.color.primary;       # output primary color
OUTPUT oScC = result.color.secondary;     # output secondary color
PARAM mvp[4] = { state.matrix.mvp };      # modelview * projection matrix
TEMP tmp;                                # temporary register
DP4 tmp.x, iPos, mvp[0];                  # Multiply input position by MVP
DP4 tmp.y, iPos, mvp[1];
DP4 tmp.z, iPos, mvp[2];
DP4 tmp.w, iPos, mvp[3];
```

```

MOV oPos, tmp;                      # Output clip-space coord
MOV oPrC, iPrC;                     # Copy primary color input to output
RCP tmp.w, tmp.w;                   # tmp now contains 1/W instead of W
MUL tmp.xyz, tmp, tmp.w;            # tmp now contains persp-divided coords
MAD oScC, tmp, 0.5, 0.5;            # map from [-1,1] to [0,1] and output
END

```

Listing 19.2. A Simple `GL_ARB_fragment_program` Fragment Shader

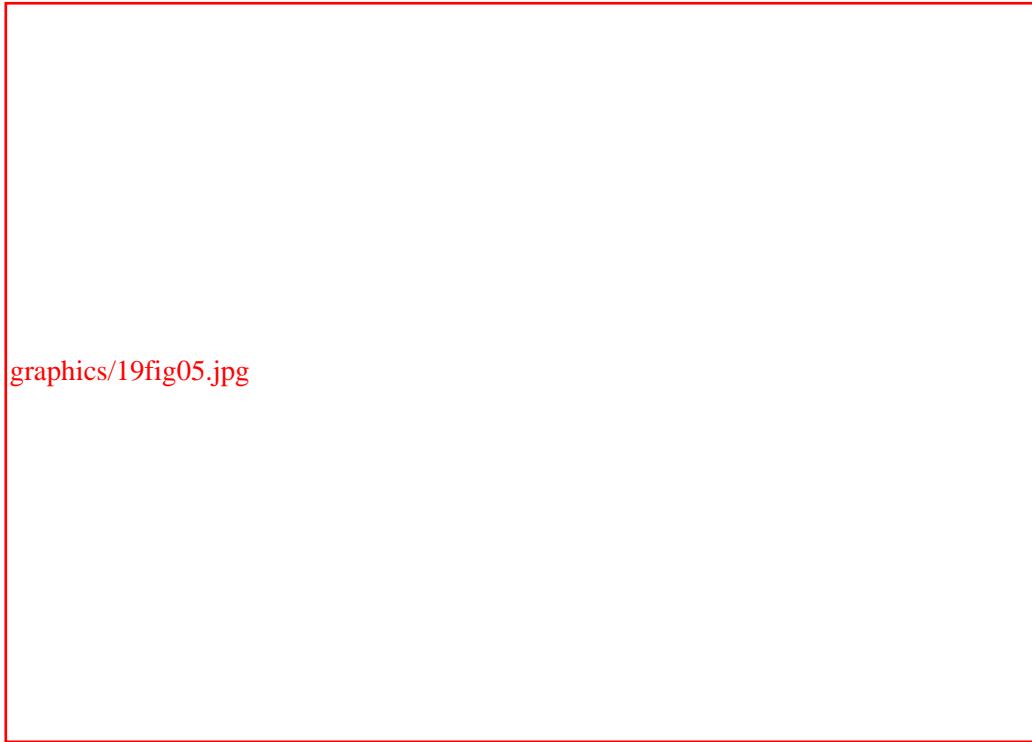
```

!!ARBfp1.0
# This is our Hello World fragment shader
ATTRIB iPrC = fragment.color.primary;      # input primary color
ATTRIB iScC = fragment.color.secondary;    # input secondary color
OUTPUT oCol = result.color;                # output color
LRP oCol.rgb, 0.5, iPrC, iScC;            # 50/50 mix of two colors
MOV oCol.a, iPrC.a;                      # ignore secondary color alpha
END

```

If these shaders are not self-explanatory, don't despair! [Chapter 20](#), "Low-Level Shading: Coding to the Metal," will make sense of it all. Basically, the vertex shader emulates fixed functionality vertex transformation by multiplying the object-space vertex position by the modelview/projection matrix. Then it copies its primary color unchanged. Finally, it generates a secondary color based on the post-perspective divide normalized device coordinates. Because they will be in the range $[-1,1]$, you also have to divide by 2 and add 1/2 to get colors in the range $[0,1]$. The fragment shader is left with the simple task of blending the primary and secondary colors together. [Figure 19.5](#) shows a sample scene rendered with these shaders.

Figure 19.5. The colors are pastel tinted by the objects' positions in the scene.



High-Level Extensions

Programming GPUs in a high-level language means less code, more readable code, and thus more productivity. The OpenGL Shading Language (GLSL) is the name of this language. It looks a lot

like C, but with built-in data types and functions that are useful to vertex and fragment shaders.

Four extensions are involved here: `GL_ARB_shader_objects`, `GL_ARB_vertex_shader`, `GL_ARB_fragment_shader`, and `GL_ARB_shading_language_100`. The first extension describes the mechanism for loading and switching between shaders and is shared by the next two extensions, one covering vertex shader specifics and one for fragment shader specifics. The fourth extension describes the GLSL language itself, again shared by vertex shaders and fragment shaders.

There is a confusing similarity in extension names between the low level and the high level: `GL_ARB_*_program` versus `GL_ARB_*_shader`. Just remember that the low-level ones are called *programs*, and the high-level ones are called *shaders*. This distinction is reinforced only by their extension names. In reality, they're all shaders.

Notice how [Listings 19.3](#) and [19.4](#), which perform the same computations as the low-level Hello World shaders, are representable in fewer lines of more readable code.

Listing 19.3. A Simple GLSL Vertex Shader

```
void main(void)
{
    // This is our Hello World vertex shader
    // notice how comments are preceded by '///'
    // normal MVP transform
    vec4 clipCoord = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_Position = clipCoord;
    // Copy the primary color
    gl_FrontColor = gl_Color;
    // Calculate NDC
    vec3 ndc = clipCoord.xyz / clipCoord.w;
    // Map from [-1,1] to [0,1] before outputting
    gl_SecondaryColor = (ndc * 0.5) + 0.5;
}
```

Listing 19.4. A Simple GLSL Fragment Shader

```
// This is our Hello World fragment shader
void main(void)
{
    // Mix primary and secondary colors, 50/50
    gl_FragColor = mix(gl_Color, vec4(vec3(gl_SecondaryColor), 1.0), 0.5);
}
```

[Chapter 21](#), "High-Level Shading," will help you understand this code if it isn't readable enough already.

Summary

In this chapter, we outlined the conventional per-vertex and per-fragment pipeline stages, setting the stage for their wholesale replacement by programmable stages. We briefly introduced both the low-level and high-level shading extensions that step in for their fixed functionality counterparts.

High-level shader compilers are improving rapidly, and like C compilers, they soon will be generating hardware code that's as good or better than hand-coded assembly. Although the low-level extensions are currently very popular, expect the high-level extensions to gain mindshare in the near future as GPU compiler technology continues to advance.

Chapter 20. Low-Level Shading: Coding to the Metal

by Benjamin Lipchak

WHAT YOU'LL LEARN IN THIS CHAPTER:

How To	Functions You'll Use
Specify shader text	<code>glProgramStringARB</code>
Switch between shaders	<code>GLBindProgramARB</code>
Create and delete shaders	<code>glGenProgramsARB/glDeleteProgramsARB</code>
Set program parameters	<code>glProgramEnvParameter*ARB/glProgramLocalParameter*ARB</code>
Query program parameters	<code>glGetProgramEnvParameter*ARB/glGetProgramLocalParameter*ARB</code>
Set vertex attributes	<code>glVertexAttrib*ARB/glVertexAttribPointerARB</code> <code>glEnableVertexAttribArrayARB/glDisableVertexAttribArrayARB</code>
Query vertex attributes	<code>glGetVertexAttrib*vARB/glGetVertexAttribPointervARB</code>
Query program object state	<code>glGetProgramivARB/glGetStringARB/glIsProgramARB</code>

Low-level shaders provide relatively direct access to the current generation of underlying shader hardware. Every cycle of the shader can be scheduled with a vector instruction that operates on four components at a time. Low-level vertex and fragment shaders use the same commands for loading and managing shaders, and they offer nearly the same instruction sets. Their biggest difference is in their inputs and outputs.

As described in [Chapter 19](#), "Programmable Pipeline: This Isn't Your Father's OpenGL," vertex shaders take unprocessed vertices and their attributes (position, normal, colors, texture coordinates, and so on), and output clip-space position and new processed attributes (colors, texture coordinates, and so on). Fragment shaders, on the other hand, take fragments and their associated data as input, and they output a final fragment color and possibly a new depth.

We could devote an entire book to shaders, but we'll try to cover all the most important aspects in this chapter. Feel free to consult the extension specifications (`GL_ARB_vertex_program` and `GL_ARB_fragment_program`) as a complete reference. After you read this chapter, those specs may actually be decipherable!

Managing Low-Level Shaders

In the following sections, we describe mostly what goes on inside the shader. First, though, you need to load shaders into OpenGL and be able to turn shaders on and off and switch between them. Then you can start writing shaders.

Creating and Binding Shaders

Like texture objects, buffer objects, occlusion queries, and other OpenGL objects, low-level shaders are loaded into objects, too—program objects in this case. First, you generate an unused program object name and then create it by binding it for the first time:

```
// Create shader objects, set shaders
```

```
glGenProgramsARB(2, ids);
 glBindProgramARB(GL_VERTEX_PROGRAM_ARB, ids[0]);
 glBindProgramARB(GL_FRAGMENT_PROGRAM_ARB, ids[1]);
```

In its initial state, the program object has no shader associated with it. If you try enabling it and drawing something, an error is thrown. So now you're ready to load up some real shaders.

Loading Shaders

You pass shaders into OpenGL as ASCII strings via `glProgramStringARB`, which takes a shader type, format, length, and pointer to the string containing the shader text:

```
glProgramStringARB(GL_VERTEX_PROGRAM_ARB, GL_PROGRAM_FORMAT_ASCII_ARB,
                    strlen(vpString), vpString);
glProgramStringARB(GL_FRAGMENT_PROGRAM_ARB, GL_PROGRAM_FORMAT_ASCII_ARB,
                    strlen(fpString), fpString);
```

The first argument indicates whether you're replacing the currently bound low-level vertex shader or fragment shader.

The format argument is used just for future expandability, such as to accept a possible Unicode or binary bytecode representation. Currently, `GL_PROGRAM_FORMAT_ASCII_ARB` is the only game in town.

Your string need not be null-terminated. `glProgramStringARB` looks only at the number of characters you tell it to in the third argument, and those characters should all be part of the actual shader text. A null terminator, if it is present, should not be included in the length argument you pass in. Conveniently, the standard C string function `strlen` behaves this way, returning the length of a string minus its terminator.

When OpenGL receives the `glProgramStringARB` command, it proceeds to parse the shader. If all goes well, your shader is compiled and optimized as necessary for the underlying hardware and is ready for rendering when you enable vertex and/or fragment shading.

If you have any syntax or semantic errors, or if the shader is too complex for the implementation to handle, an error is thrown. In this case, the currently bound shader is not replaced, and whatever shader was there before (if any) remains in place. You can find out where and why a problem occurred by querying for the error position and error string.

The error position is the byte offset into the shader string where the error occurred. If no error occurs, you get back `-1`. If the error is a semantic restriction that can be discovered only after the whole shader is parsed (for example, trying to use the same texture unit for both 2D and 3D texturing), the error position is set to the length of the shader.

The error string is the most useful way to diagnose your problem. It tells you the type of error that occurred and may provide additional hints, such as the line number where the error occurred. Using the error string to find the error in your shader is a lot easier than using the error position and trying to count out hundreds of characters by hand!)

[Listing 20.1](#) shows the code used to set up low-level shaders.

Listing 20.1. Setting Up Low-Level Shaders

```
// Create, set and enable shaders
glGenProgramsARB(2, ids);
 glBindProgramARB(GL_VERTEX_PROGRAM_ARB, ids[0]);
 glProgramStringARB(GL_VERTEX_PROGRAM_ARB, GL_PROGRAM_FORMAT_ASCII_ARB,
                    strlen(vpString), vpString);
```

```

glGetIntegerv(GL_PROGRAM_ERROR_POSITION_ARB, &errorPos);
if (errorPos != -1)
{
    fprintf(stderr, "Error in vertex shader at position %d!\n", errorPos);
    fprintf(stderr, "Error string: %s\n",
            glGetString(GL_PROGRAM_ERROR_STRING_ARB));
    Sleep(5000);
    exit(0);
}
glBindProgramARB(GL_FRAGMENT_PROGRAM_ARB, ids[1]);
glProgramStringARB(GL_FRAGMENT_PROGRAM_ARB, GL_PROGRAM_FORMAT_ASCII_ARB,
                    strlen(fpString), fpString);
glGetIntegerv(GL_PROGRAM_ERROR_POSITION_ARB, &errorPos);
if (errorPos != -1)
{
    fprintf(stderr, "Error in fragment shader at position %d!\n", errorPos);
    fprintf(stderr, "Error string: %s\n",
            glGetString(GL_PROGRAM_ERROR_STRING_ARB));
    Sleep(5000);
    exit(0);
}
if (useVertexShader)
    glEnable(GL_VERTEX_PROGRAM_ARB);
if (useFragmentShader)
    glEnable(GL_FRAGMENT_PROGRAM_ARB);

```

Try adding a typographical error into one of the shaders loaded by the sample code in [Listing 20.1](#). See how well the error string on your OpenGL implementation helps you narrow down the problem.

Deleting Shaders

With shaders, just like other OpenGL objects, you need to clean up after yourself when you're done. Deleting your shaders frees the resources and makes the names available for use again later:

```
glDeleteProgramsARB(2, ids);
```

Setting Up the Extensions

One more detail stands between us and diving into the actual shaders. Low-level vertex and fragment shaders are not part of core OpenGL. In previous chapters, we covered functionality that used to be extensions but have since been promoted to the core. However, with high-level shaders gaining popularity, these low-level shaders will likely never be promoted and will live their lives forever as ARB-approved extensions.

Their extension status does not make much difference when it comes to using them. At least on Windows platforms, any functionality more recent than OpenGL 1.1, whether core or extension, requires its entrypoint function pointers to be queried before use. Before doing that, you must check also for the presence of the extensions in the extension string. The sample code also uses the secondary color feature, which either requires OpenGL 1.4 or the `GL_EXT_secondary_color` extension. [Listing 20.2](#) shows how to check whether an OpenGL implementation supports the required features.

Listing 20.2. Checking for the Presence of OpenGL Features

```

// Make sure required functionality is available!
if (!glIsExtSupported("GL_ARB_vertex_program"))
{
    fprintf(stderr, "GL_ARB_vertex_program extension is unavailable!\n");
}

```

```

    Sleep(2000);
    exit(0);
}
if (!gltIsExtSupported( "GL_ARB_fragment_program" ))
{
    fprintf(stderr, "GL_ARB_fragment_program extension is unavailable!\n");
    Sleep(2000);
    exit(0);
}
version = glGetString(GL_VERSION);
if (((version[0] != '1') || (version[1] != '.') || (version[2] < '4') || (version[2] > '9')) && // 1.4+
    (!gltIsExtSupported( "GL_EXT_secondary_color" )))
{
    fprintf(stderr, "Neither OpenGL 1.4 nor GL_EXT_secondary_color"
            " extension is available!\n");
    Sleep(2000);
    exit(0);
}
glGenProgramsARB = gltGetExtensionPointer( "glGenProgramsARB" );
glBindProgramARB = gltGetExtensionPointer( "glBindProgramARB" );
glProgramStringARB = gltGetExtensionPointer( "glProgramStringARB" );
glDeleteProgramsARB = gltGetExtensionPointer( "glDeleteProgramsARB" );
if (gltIsExtSupported( "GL_EXT_secondary_color" ))
    glSecondaryColor3f = gltGetExtensionPointer( "glSecondaryColor3fEXT" );
else
    glSecondaryColor3f = gltGetExtensionPointer( "glSecondaryColor3f" );
if (!glGenProgramsARB || !glBindProgramARB || !glProgramStringARB ||
    !glDeleteProgramsARB || !glSecondaryColor3f)
{
    fprintf(stderr, "Not all entrypoints were available!\n");
    Sleep(2000);
    exit(0);
}

```

Instruction Sets

Each cycle, or instruction slot, of a low-level vertex or fragment shader can have a different instruction opcode, such as `ADD`, `MUL`, or `MOV`, to perform addition, multiplication, or copy, respectively. These opcodes are your basic shader building blocks. Most instruction opcodes are followed by a single output and one or more inputs, separated by commas. Each instruction ends with a semicolon.

```
MUL myResult, myTemp, 2.0;    # multiplies each component of myTemp by 2,
                             # stores in myResult
```

With few exceptions, the vertex shader and fragment shader instruction sets are identical. In the following sections, we first discuss the significant overlap between the two, and then we deal with the instructions that are specific to one shader type or the other.

Common Instructions

The instruction set can be categorized by the number of input arguments, whether the inputs are vector or scalar, and whether the result is vector or scalar. *Vector* in this case means a four-component vector, whereas a *scalar* is a single component. [Table 20.1](#) categorizes all the instructions that are common to both vertex shaders and fragment shaders.

Table 20.1. Common Instruction Set

Instruction	Inputs	Output	Description
ABS	1 vector	vector	Takes the absolute value of the input vector.
ADD	2 vectors	vector	Adds two vectors together.
DP3	2 vectors	scalar	Takes the dot product of the first three components of input vectors.
DP4	2 vectors	scalar	Takes the dot product of all four components of input vectors.
DPH	2 vectors	scalar	Takes the dot product of the first three components of input vectors and adds in the fourth component of the second input.
DST	2 vectors	vector	Computes a distance vector given two specially formatted input vectors; see the specification for details.
EX2	1 scalar	scalar	Computes 2 to the power of the scalar input.
FLR	1 vector	vector	Takes the floor—that is, the largest integer less than or equal to the input.
FRC	1 vector	vector	Takes the fractional portion, or the part left over after subtracting the floor from the input.
LG2	1 scalar	scalar	Computes the base 2 logarithm of the scalar input.
LIT	1 vector	vector	Computes lighting coefficients given a specially formatted input vector; see the specification for details.
MAD	3 vectors	vector	Multiplies the first two vectors and then adds the third.
MAX	2 vectors	vector	Takes the maximum of each component of the two input vectors.
MIN	2 vector	vector	Takes the minimum of each component of the two input vectors.
MOV	1 vector	vector	Copies the vector.
MUL	2 vectors	vector	Multiplies two vectors together.
POW	2 scalars	scalar	Computes the first scalar input to the power of the second scalar input.
RCP	1 scalar	scalar	Takes the reciprocal of the scalar input.
RSQ	1 scalar	scalar	Takes the reciprocal of the square root of the absolute value of the scalar input.
SGE	2 vectors	vector	Results in 1 for an output component if the corresponding component of the first input vector is greater than or equal to its counterpart in the second input vector; otherwise 0.
SLT	2 vectors	vector	Results in 1 for an output component if the corresponding component of the first input vector is less than its counterpart in the second input vector; otherwise 0.
SUB	2 vectors	vector	Subtracts the second vector from the first.
SWZ	1 vector	vector	Copies but with extended swizzling capabilities such as swizzling in 0 or 1 and per-component negate; see the specification for details.
XPD	2 vectors	vector	Computes the cross product of the first three components of each input vector; the fourth component is undefined.

All instructions that output a vector operate independently on each component of the vector. For example, **MUL** actually performs four independent multiplication operations:

```
MUL myResult, myTemp1, myTemp2;
# This is the same as:
# myResult.x = myTemp1.x * myTemp2.x
# myResult.y = myTemp1.y * myTemp2.y
# myResult.z = myTemp1.z * myTemp2.z
# myResult.w = myTemp1.w * myTemp2.w
```

On the other hand, instructions that output a single scalar actually replicate that scalar to all components of the result vector:

```
RCP myResult, myTemp.x;
# This is the same as:
# myResult.x = myResult.y = myResult.z = myResult.w = 1.0 / myTemp.x
```

Vertex-Specific Instructions

Only three instructions are specific to low-level vertex shaders: **ARL**, **EXP**, and **LOG**. [Table 20.2](#) describes these instructions.

Table 20.2. Vertex-Specific Instruction Set

Instruction	Inputs	Output	Description
ARL	1 vector	address	Loads an address register.
EXP	1 scalar	vector	Computes a rough approximation of 2 to the power of the scalar input with special output formatting; see the specification for details.
LOG	1 scalar	vector	Computes a rough approximation of the base 2 logarithm of the scalar input with special output formatting; see the specification for details.

The **ARL** instruction is a special-purpose instruction used to load an address register, which is a single-component signed integer register type used for relative addressing. Before the address register is loaded, the address is floored so that it becomes the greatest integer less than or equal to the scalar input. We discuss relative addressing in a later section, "[Addresses](#)."

The **EXP** and **LOG** instructions are lower precision approximations of their **EX2** and **LG2** counterparts, except they put the result only in the third component of the result vector. The first two components are filled with some other marginally useful approximation factors, and the fourth component is 1. Because these instructions provide no additional benefit except possibly improved performance on some OpenGL implementations, they were removed from the fragment shader instruction set.

Fragment-Specific Instructions

After the low-level vertex program extension was approved by the ARB, work began on a counterpart fragment program extension. All the instructions from the vertex extension were considered for inclusion, and only the three listed in [Table 20.2](#) were removed.

Relative addressing within fragment shaders is not yet a feature available in today's hardware, so the **ARL** instruction was not included. Also, the low-precision **EXP** and **LOG** instructions were not particularly interesting compared to the full-precision versions, so they were dropped as well.

Quite a number of instructions particularly useful in the fragment domain were added. [Table 20.3](#) lists them.

Table 20.3. Fragment-Specific Instruction Set

Instruction	Inputs	Output	Description
CMP	3 vectors	vector	Copies component from the second input if the first input's component is less than zero; otherwise copies component from the third input.
COS	1 scalar	scalar	Computes the cosine of the scalar input, which is in radians and need not be in the range $[-\pi, \pi]$.
KIL	1 vector	none	Kills the current fragment processing and bypasses subsequent pipeline stages if any component of the input vector is less than zero.
LRP	3 vectors	vector	Interpolates linearly between the second and third input vectors based on the first input vector, as in $r = i0*i1 + (1-i0)*i2$.
SCS	1 scalar	2 scalars	Computes both the cosine in the first component and sine in the second component based on the scalar input, which must be radians in the range $[-\pi, \pi]$.
SIN	1 scalar	scalar	Computes the sine of the scalar input, which is in radians and need not be in the range $[-\pi, \pi]$.
TEX	1 vector	vector	Performs a nonprojective texture lookup using the input vector's first three components as the texture coordinate.
TXB	1 vector	vector	Performs a nonprojective texture lookup using the input vector's first three components as the texture coordinate and the fourth component as an LOD bias.
TXP	1 vector	vector	Performs a projective texture lookup whereby the input vector's first three components are first divided by its fourth component before being used as the texture coordinate.

Fragment shaders introduce a new type of instruction: the *texture* instruction. The rest of the instructions fall into the *ALU* category because they perform arithmetic operations. **TEX**, **TXB**, and **TXP** are the three instructions that fall nicely into this new texture category.

Each of these first three texture instructions performs a texture lookup on the specified texture target (1D, 2D, 3D, CUBE) of the specified texture unit. For example, to sample from the cube map on texture unit 0, you use

```
TEX myResult, myTexCoord, texture[0], CUBE;
```

KIL, a unique instruction that can be used to stop all further fragment shader execution and discard the fragment, actually falls into the texture instruction category, too. It doesn't perform a texture lookup, but it may be implemented in hardware using the same resources.

Variable Types

The instruction set is great, but those opcodes won't do a thing for you without data to operate on and a place to store the result. The six different types of variables described in Table 20.4 can be used as inputs and/or outputs in your low-level shaders.

Temporary

TEMP

read-write

This is a standard all-purpose temporary storage register.

Parameter

PARAM

read-only

This is a constant register that never changes over the course of shader execution.

Attribute

ATTRIB

read-only

This is a shader input.

Output

OUTPUT

write-only

This is a shader output.

Address

ADDRESS

write-only

This is a signed integer constant, available only within vertex shaders.

Alias

ALIAS

n/a

This is just another variable name given to a variable of one of the other types.

Table 20.4. Variable Types

Variable Type	Declaration	Access	Description

All variables represent four-component floating-point vectors, except for address registers, which are signed integers.

Temporaries

Temporaries are the main work horses of low-level shaders. Unless you're writing to an output register, chances are you're writing to a temp. Before you can use a variable as a temp, you have to declare it. You can declare multiple temps at the same time if you want:

```
TEMP diffuseColor, specColor, myTexCoord;
```

Now you can use these variable names as inputs or outputs of any instruction.

Parameters

Parameters are variables that never change during each run of a shader. You can think of them as constants, except that some parameters actually can be changed outside the shader. In any case, you can't write to a parameter during shader execution. You can use them only as instruction inputs. The three types of parameters are *inline constants*, *state-bound parameters*, and *program parameters*. Parameters can be declared with variable names, but they don't have to be, as we illustrate in the following sections.

Inline Constants

Inline constants really are constants. They're set to specific values within the text of the shader, and they can never change. You can either set all four components of the vector to the same value, set each component to a unique value, or have unspecified components filled out with default values:

```
PARAM two = 2.0;                                # all 4 components contain 2
PARAM quarters = { 0.0, 0.25, 0.5, 0.75 }; # 4 unique values
PARAM pi = { 3.14159 };                         # vector gets padded out to
                                                # PI, 0, 0, 1
```

Be aware of the subtle differences between the use of braces and no braces! For example, `2.0` and `{2.0}` both have the same value in the first component, but the other three components differ. This is a common low-level shader writing pitfall.

You don't need to declare parameters if you don't want to. You can use them directly as instruction inputs:

```
MUL tripleCoord, myCoord, 3.0f;
MUL scaledResult, {0.1, 0.2, 0.3, 0.4}, myResult;
```

State-Bound Parameters

For convenience, you can access a variety of OpenGL state in the form of state-bound parameters. When OpenGL state changes, the parameters are automatically updated to reflect the changes. This makes emulating fixed functionality more straightforward. For example, instead of manually loading up the modelview/projection (MVP) matrix into four parameter vectors, you can just use the MVP already available in OpenGL state to transform your vertex position:

```
DP4 result.position.x, vertex.position, state.matrix.mvp.row[0];
DP4 result.position.y, vertex.position, state.matrix.mvp.row[1];
DP4 result.position.z, vertex.position, state.matrix.mvp.row[2];
DP4 result.position.w, vertex.position, state.matrix.mvp.row[3];
```

Bindable state includes all transformation matrices and the properties of texture coordinate generation, color material, lighting, fog, clip planes, point size and attenuation, texture environment colors, and depth range. Some bindable state parameters are specific to vertex shaders or specific to fragment shaders. Refer to the [GL_ARB_vertex_program](#) and [GL_ARB_fragment_program](#) extension specifications for the complete list of state parameter bindings available to low-level vertex and fragment shaders.

Program Parameters

In addition to the inline constants hard-coded into the text and the parameters bound to specific OpenGL state, you can use a third category of generic parameters. They can be loaded with any values and then reloaded later with different values.

Program parameters are divided into two categories: *program local* and *program environment* parameters. The local ones are specific to a single shader, whereas the environment parameters are shared by all shaders of a given type. That is, vertex shaders share one set of environment parameters, and fragment shaders share their own set. The parameters are loaded with the commands `glProgramLocalParameter4*ARB` and `glProgramEnvParameter4*ARB`, which take a slot number and four values. They are then referenced by `program.local[n]` and `program.env[n]` within the shader text.

So, if parameters are constants, why would you want to use program parameters instead of just hard-coding the constants into your shader? Certainly, the shader would be more readable if the constant value were explicit in the shader text. Let's consider an example. Maybe you're rendering a scene with a flickering candle, and the brightness of the flicker changes with every frame of rendering. You might have a shader that ends with something like this:

```
MUL finalColor, litColor, program.local[0]; # local 0 contains flicker factor
# in range [0,1]
```

You could then reuse the shader unchanged and simply update the local parameter once per frame:

```
glProgramLocalParameter4fARB(GL_FRAGMENT_PROGRAM_ARB, 0,
                           0.75f, 0.75f, 0.75f, 0.75f);
renderScene();
glProgramLocalParameter4fARB(GL_FRAGMENT_PROGRAM_ARB, 0,
                           0.2f, 0.2f, 0.2f, 0.2f);
renderScene();
glProgramLocalParameter4fARB(GL_FRAGMENT_PROGRAM_ARB, 0,
                           0.5f, 0.5f, 0.5f, 0.5f);
renderScene();
```

The parameter is constant during each execution of the vertex or fragment shader, but it isn't constant over all rendered primitives over time. You can change program parameters (or state-

bound parameters for that matter) as often as you like outside `glBegin/glEnd` pairs.

Parameter Arrays

You can declare an array of parameters that can be indexed either with absolute addressing or relative addressing. With absolute addressing, you supply the exact array index you want to use. Relative addressing is discussed later in the section "Addresses ."

The following are some examples of parameter array declarations. You can declare the array size if you want or let it be sized automatically. If you declare the size but then provide values beyond the size you declared, the shader will fail to parse:

```
# This one is explicitly sized to 10 vectors
PARAM myArray[10] = { 2.0, {0.1, 0.2, 0.3, 0.4}, program.env[0..5],
                      state.fog.color, -1.0};
# This one automatically gets sized to 6 vectors
PARAM myOtherArray[] = { state.matrix.mvp, state.matrix.texture[0].row[1..2] };
```

Absolute addressing of the arrays is simply a matter of providing an index when using it in an instruction:

```
# Scale by 2, then subtract 1 courtesy of the multiply then add (MAD) instruction
MAD scaledAndBiased, myColor, myArray[0], myArray[9];
```

Attributes

Like parameters, attributes are also read-only inputs. But unlike parameters, attributes tend to change on a per-execution basis. Each new vertex being shaded has a new position, and possibly new input colors and texture coordinates. The same is true of each fragment.

With attributes, like parameters, you can choose to declare them up front, or you can use them directly within an instruction:

```
TEMP nDotC;
ATTRIB vNorm = vertex.normal;           # declared attribute
DP3 nDotC, vNorm, vertex.color.primary; # color was not declared
```

Vertex shaders and fragment shaders have their own sets of input attributes, so we cover them separately.

Vertex Attributes

Table 20.5 lists all the input attributes available to vertex shaders. They correspond to all the OpenGL current vertex state that can be changed per-vertex within a `glBegin /glEnd` pair.

`vertex.position`

`(x,y,z,w)`

Object-space position

`vertex.normal`

`(x,y,z,1)`

Normal

`vertex.color`

(r,g,b,a)

Primary color

`vertex.color.primary`

(r,g,b,a)

Primary color

`vertex.color.secondary`

(r,g,b,a)

Secondary color

`vertex.fogcoord`

$(f,0,0,1)$

Fog coordinate

`vertex.texcoord`

(s,t,r,q)

Texture coordinate on unit 0

`vertex.texcoord[n]`

(s,t,r,q)

Texture coordinate on unit n

`vertex.attrib[n]`

(x,y,z,w)

Generic attribute n

Table 20.5. Vertex Attributes

Attribute Binding	Components	Description

The one vertex attribute you've probably never seen before is the generic attribute. These

attributes have been introduced so that you can specify any kind of per-vertex data, not necessarily one of the kinds previously available with fixed functionality. Binormals, tangent vectors, you name it—anything you'd like to stream into a vertex shader, you can send in through a generic attribute.

You can use the many flavors of the `glVertexAttrib*ARB` command to set these generic attributes, or you can put generic attributes in a vertex array using `glVertexAttribPointerARB` and `glEnableVertexAttribArrayARB`.

One point to keep in mind is that on some implementations, these generic attributes overlap with the fixed functionality attributes. One important case is that calling `glVertexAttrib` on attribute 0 is guaranteed to be the same as calling `glVertex`, and vice versa. But you need to be more careful with all the other possible aliasing conflicts. Table 20.6 lists the conflicts between generic and fixed functionality attributes.

`vertex.attrib[0]`

Vertex position

`vertex.position`

`vertex.attrib[1]`

None

`none`

`vertex.attrib[2]`

Normal

`vertex.normal`

`vertex.attrib[3]`

Primary color

`vertex.color, vertex.color.primary`

`vertex.attrib[4]`

Secondary color

`vertex.color.secondary`

`vertex.attrib[5]`

Fog coordinate

`vertex.fogcoord`

`vertex.attrib[6]`

None

none

`vertex.attrib[7]`

None

none

`vertex.attrib[8]`

Texture coordinate 0

`vertex.texcoord`

`vertex.attrib[8+n]`

Texture coordinate `n`

`vertex.texcoord[n]`

Table 20.6. Vertex Attribute Aliasing

Generic Binding	Overlapping Attribute	Overlapping Binding

If you call the command to change one attribute on each line of the table, the other becomes undefined (for example, calling `glVertexAttrib` on attribute 2 undefines the normal set with `glNormal`, and vice versa). Also, you cannot bind to both attributes on each line of the table within the same shader, or your shader will fail to parse. This helps catch accidental aliasing bugs. This result would occur if, for example, you tried to use both `vertex.attrib[4]` and `vertex.color.secondary` within your shader.

Fragment Attributes

Fragment attributes are the fragment's position and other associated data, interpolated across the primitive. No generic attributes are available here—just the same interpolants available via fixed functionality. Table 20.7 lists all the fragment attributes and their fragment shader bindings.

`fragment.position`

$(x, y, z, 1/w)$

Window-space position, reciprocal of clip-space w

`fragment.color`

(r, g, b, a)

Primary color

`fragment.color.primary` (r,g,b,a)

Primary color

`fragment.color.secondary` (r,g,b,a)

Secondary color

`fragment.texcoord` (s,t,r,q)

Texture coordinate 0

`fragment.texcoord[n]` (s,t,r,q) Texture coordinate *n*`fragment.fogcoord` $(f,0,0,1)$

Fog coordinate/distance

Table 20.7. Fragment Attributes

Attribute Binding	Components	Description

Outputs

Outputs are write-only registers that can be used to store the result of an instruction. Like input attributes, outputs are also necessarily different between low-level vertex and fragment shaders.

Vertex Outputs

The set of vertex output registers for the most part represents the colors and coordinates that will be interpolated across the primitive to which the vertex belongs and will become available as fragment shader input attributes. Table 20.8 lists all the low-level vertex shader outputs.

`result.position` (x,y,z,w)

Clip-space position

`result.color`

(r,g,b,a)

Front-facing primary color

`result.color.primary`

(r,g,b,a)

Front-facing primary color

`result.color.secondary`

(r,g,b,a)

Front-facing secondary color

`result.color.front`

(r,g,b,a)

Front-facing primary color

`result.color.front.primary`

(r,g,b,a)

Front-facing primary color

`result.color.front.secondary`

(r,g,b,a)

Front-facing secondary color

`result.color.back`

(r,g,b,a)

Back-facing primary color

`result.color.back.primary`

(r,g,b,a)

Back-facing primary color

`result.color.back.secondary`

(r,g,b,a)

Back-facing secondary color

`result.fogcoord`

$(f, *, *, *)$

Fog coordinate

`result.pointsize`

$(s, *, *, *)$

Point size

`result.texcoord`

(s, t, r, q)

Texture coordinate 0

`result.texcoord[n]`

(s, t, r, q)

Texture coordinate n

Table 20.8. Vertex Outputs

Output Binding	Components	Description

The fog coordinate and point size outputs are scalar. Only the first component is used, and the others are ignored. The point size is used only during rasterization to affect the size of the point primitive being generated. It does not become available as a fragment shader input.

Notice that there are four color outputs from the vertex shader and only two color input attributes in the fragment shader. The reason is that the orientation of the primitive is determined during rasterization, at which point either the front-facing or back-facing colors are passed along to the fragment shader.

Any output that isn't written by the vertex shader becomes undefined, so if you then try to use it as an input in the fragment shader, you get garbage. Moral of the story: Make sure that you match up your fragment shader with a vertex shader that generates all the needed interpolants!

Fragment Outputs

Fragment shaders have only two outputs, a final color and a depth (see Table 20.9).

`result.color`

(r, g, b, a)

Color

`result.depth``(*,*,*d,*)`

Depth coordinate

Table 20.9. Fragment Outputs

Output Binding	Components	Description

The output color is passed along to subsequent per-fragment operations, such as alpha test and blending, and finally is stored in the framebuffer.

The depth output is handled a bit differently than other outputs. Whereas all other outputs are undefined if you don't write them, fragment depth defaults to the depth produced by rasterization if you don't write it. If you do write to the depth output, it overrides the rasterization depth, and this depth is passed along to subsequent stencil and depth test stages.

Aliases

An alias isn't actually its own type of register. It's just a way of giving a new variable name to an existing register. Temporaries are limited resources, so aliases let you give meaningful new names to "recycled" registers. Here's a contrived example:

```
TEMP baseMap, outColor;
ALIAS lightMap = baseMap;
MOV outColor, fragment.color;
TEX baseMap, fragment.texcoord[0], texture[0], 2D;
MUL outColor, outColor, baseMap;
# This next texture lookup puts its result in the same
# physical temp as the last lookup, but gives it a new
# name to make the shader more easily readable
TEX lightMap, fragment.texcoord[1], texture[1], 2D;
MUL outColor, outColor, lightMap;
```

Addresses

Address registers are used for relative addressing of parameter arrays. This type of addressing gives you access into an array using an arbitrarily computed index. Relative addressing is allowed only in low-level vertex shaders, not in fragment shaders.

Only the first component of an address register is used. The other three components might as well not exist; they can neither be read nor written. Before using relative addressing, you have to declare your address register and then write to it with the [ARL](#) (address register load) instruction:

```
ADDRESS myAddress;
TEMP computedAddress, chosenOffset;
PARAM offsets[] = { 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0 };
...
```

```
# ARL is a glorified FLR instruction that only writes to address registers
ARL myAddress.x, computedAddress;
MOV chosenOffset, offsets[myAddress.x];
```

Input and Output Modifiers

A few operations can be applied to input and output registers as part of each instruction. They are described in the following sections.

Input Negate

The first operation we'll discuss is input negate. You can negate each input argument to an instruction by putting a minus sign in front of it:

```
MOV negativeVal, -positiveVal;
```

Input Swizzle

Another modifier to input arguments is the swizzle suffix. This suffix swizzles, or rearranges, the components of an input register. This example takes a parameter vector and reverses the order of its components:

```
PARAM someConstant = { 1, 2, 3, 4 };
...
# The following swizzle results in 4,3,2,1
MUL result, result, someConstant.wzyx;
```

The swizzle can be any combination of the components `x`, `y`, `z`, and `w`, such as `.zzzz`, `.xywy`, or even the redundant `.xyzw`. It can also be a single component, where `.x` is equivalent to `.xxxx`. Low-level fragment shaders let you use the letters `r`, `g`, `b`, and `a` for your swizzles as well because colors are more predominant than coordinates within fragment shaders. `.abgr` is the same as `.wzyx`.

For scalar instructions that operate on a single input channel (`COS`, `EX2`, `EXP`, `LG2`, `LOG`, `POW`, `RCP`, `RSQ`, `SCS`, and `SIN`), you are forced to use a suffix to select the single component that will be used:

```
RCP oneOverZ, myCoord.z;
```

Output Writemask

Swizzle suffixes determine which components to make available from each input register. Similarly, writemask suffixes determine which output components are written and which remain untouched:

```
MOV myResult.xyw, foo; # 3rd component stays as-is
```

The same component letters can be used for writemasks as for swizzles. Vertex shaders can use `x`, `y`, `z`, and `w`, whereas fragment shaders can also use `r`, `g`, `b`, and `a` for their writemasks. This is just in the name of readability. `myColor.rgb` is a lot easier to understand at first glance than `myColor.xyz`.

Output Clamp

The final modifier is output clamp, also known as saturation. It clamps the result of an instruction to the range $[0,1]$. This modifier is most useful for colors, and therefore is available only in the

low-level fragment shader.

To do an output clamp, you just add the `_SAT` suffix to your fragment shader instruction, as follows:

```
ADD      myResult, primaryColor, secondaryColor;  # This could overflow
                           # outside [0,1]
ADD_SAT myResult, primaryColor, secondaryColor;  # Here we clamp to [0,1]
```

The only instruction this doesn't make sense for is the `KIL` instruction, which has no output.

Note that OpenGL automatically clamps the final color and depth outputs from your fragment shader before using them in subsequent pipeline stages, so you don't need to add `_SAT` yourself on the final writes to `result.color` or `result.depth`. The output clamp modifier is just there to facilitate the clamping of intermediate computations.

If you want to clamp a register value within a vertex shader, your best bet is to use the `MIN` and `MAX` instructions. This sequence can also be used in a fragment shader to clamp to an arbitrary range other than $[0,1]$:

```
MIN myValue, myValue, 1;
MAX myValue, myValue, 0;
```

Resource Consumption and Queries

OpenGL implementations have a limited number of resources available to low-level shaders. These resources include temporaries, parameters, instructions, and a few others. If you want your shader to run fast, or even run at all, you need to pay attention to these limits and try to minimize your resource consumption.

Parser Limits

The first set of limits is the parser limits. These limits dictate the maximum number of resources that can be present in your shader for OpenGL to even consider trying to compile it. If you exceed any of these limits when calling `glProgramStringARB`, your shader will fail to parse, an error will be thrown, and the error string will reflect which resource you overused.

Table 20.10 lists each resource, the way to query its parser limit via `glGetProgramivARB`, and the minimum number that must be supported by all implementations. Limits are different for vertex shaders and fragment shaders, and some limits apply only to one or the other.

Table 20.10. Parser Resource Limit Queries

Resource	Query	VS Min.	FS Min.
Instructions	<code>GL_MAX_PROGRAM_INSTRUCTIONS_ARB</code>	128	72
Temporaries	<code>GL_MAX_PROGRAM_TEMPORARIES_ARB</code>	12	16
Parameters	<code>GL_MAX_PROGRAM_PARAMETERS_ARB</code>	96	24
Program env parameters	<code>GL_MAX_PROGRAM_ENV_PARAMETERS_ARB</code>	96	24
Program local parameters	<code>GL_MAX_PROGRAM_LOCAL_PARAMETERS_ARB</code>	96	24
Attributes	<code>GL_MAX_PROGRAM_ATTRIBS_ARB</code>	16	10

Addresses	<code>GL_MAX_PROGRAM_ADDRESS_REGISTERS_ARB</code>	1	n/a
ALU instructions	<code>GL_MAX_PROGRAM_ALU_INSTRUCTIONS_ARB</code>	n/a	48
Texture instructions	<code>GL_MAX_PROGRAM_TEX_INSTRUCTIONS_ARB</code>	n/a	24
Texture indirections	<code>GL_MAX_PROGRAM_TEX_INDIRECTIONS_ARB</code>	n/a	4

If your shader parsed successfully, you can call `glGetProgramivARB` to find out how many of each resource was counted by the parser. [Table 20.11](#) lists these query tokens.

Table 20.11. Parser Resource Consumption Queries

Resource	Query
Instructions	<code>GL_PROGRAM_INSTRUCTIONS_ARB</code>
Temporaries	<code>GL_PROGRAM_TEMPORARIES_ARB</code>
Parameters	<code>GL_PROGRAM_PARAMETERS_ARB</code>
Attributes	<code>GL_PROGRAM_ATTRIBS_ARB</code>
Addresses (VS only)	<code>GL_PROGRAM_ADDRESS_REGISTERS_ARB</code>
ALU instructions (FS only)	<code>GL_PROGRAM_ALU_INSTRUCTIONS_ARB</code>
Texture instructions (FS only)	<code>GL_PROGRAM_TEX_INSTRUCTIONS_ARB</code>
Texture indirections (FS only)	<code>GL_PROGRAM_TEX_INDIRECTIONS_ARB</code>

Texture Indirections

You may find yourself asking, "What's a texture indirection?" Or you might not ask yourself that question until the first time you try loading a big shader, and you get an error string complaining about texture indirections. This is as good a place as any to address this resource.

Fixed functionality always used texture coordinate interpolants to sample from textures. But fragment shaders introduce the ability to use any arbitrarily computed temporary register as a texture coordinate. In fact, you can use the result of one texture lookup as the texture coordinate for another lookup. This is called a *dependent lookup*.

A *chain* of dependent lookups is simply one dependent lookup after another, where the result of one lookup becomes the texture coordinate for the next lookup, repeated some number of times. Some hardware implementations have an internal limit on the length of the dependency chain that can be used within a fragment shader. This is one of the most common fragment shader pitfalls, and if you take a moment to familiarize yourself with texture indirections, you can avoid hitting the ceiling of this resource.

The `GL_ARB_fragment_program` specification provides a fairly simple algorithm used by parsers for counting texture indirections. In the specification, see issue 24, "What is a texture indirection, and how is it counted?" It's easy enough that you can run the algorithm in your head when scanning your own fragment shader text to find out where indirections are being introduced and how to eliminate unnecessary ones.

Native Limits

The parser limits reflect what you pass into the parser. All implementations should count resources exactly the same way against the parser limits. But when your shader falls within these parse limits and successfully parses, you enter a world where all hardware is different. Optimizing compilers take over from here.

Native resource limits reflect more closely what the hardware really has to offer. For example, some hardware implementations might take eight native instructions to perform a sine/cosine (SCS), whereas others might take just one cycle. In both cases, the parser counts this as just one instruction, but the native resource count varies.

Even if, on ideal mythical hardware, all instructions consumed just one native cycle, your parser limits and native limits might still be different. An implementation might advertise twice as many resources for its parser limit than it can actually support in hardware. This would give an optimizer the opportunity to try to reduce the shader enough that it would fit within the native limits. Such optimizations include instruction rescheduling, dead code removal, constant folding, and temporary register collapsing. Significantly reducing a shader's native resource consumption is possible, so it makes sense to give the compiler bigger shaders to take a crack at.

Tables 20.12 and 20.13 list the native limit queries as well as the queries to find out the native consumption of a successfully compiled and optimized shader. They all are queried via `glGetProgramivARB`.

Table 20.12. Native Resource Limit Queries

Resource	Query
Instructions	<code>GL_MAX_PROGRAM_NATIVE_INSTRUCTIONS_ARB</code>
Temporaries	<code>GL_MAX_PROGRAM_NATIVE_TEMPORARIES_ARB</code>
Parameters	<code>GL_MAX_PROGRAM_NATIVE_PARAMETERS_ARB</code>
Attributes	<code>GL_MAX_PROGRAM_NATIVE_ATTRIBS_ARB</code>
Addresses (VS only)	<code>GL_MAX_PROGRAM_NATIVE_ADDRESS_REGISTERS_ARB</code>
ALU instructions (FS only)	<code>GL_MAX_PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB</code>
Texture instructions (FS only)	<code>GL_MAX_PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB</code>
Texture indirections (FS only)	<code>GL_MAX_PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB</code>

Table 20.13. Native Resource Consumption Queries

Resource	Query
Instructions	<code>GL_PROGRAM_NATIVE_INSTRUCTIONS_ARB</code>
Temporaries	<code>GL_PROGRAM_NATIVE_TEMPORARIES_ARB</code>
Parameters	<code>GL_PROGRAM_NATIVE_PARAMETERS_ARB</code>
Attributes	<code>GL_PROGRAM_NATIVE_ATTRIBS_ARB</code>
Addresses (VS only)	<code>GL_PROGRAM_NATIVE_ADDRESS_REGISTERS_ARB</code>
ALU instructions (FS only)	<code>GL_PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB</code>

Texture instructions (FS only)	<code>GL_PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB</code>
Texture indirections (FS only)	<code>GL_PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB</code>
All native limits satisfied? (0/1)	<code>GL_PROGRAM_UNDER_NATIVE_LIMITS_ARB</code>

Notice the last entry in [Table 20.12](#). If `glProgramStringARB` returns without error, you can then perform this single query to determine whether all resources fell within native limits. If the result is true, you can expect your shader to be hardware accelerated. If it is false, your shader may be executed in software, which can be painfully slow. Or if you prefer, you can query each resource individually to get a finer-detailed perspective on which native resources are near or exceeding their limits.

Other Queries

Like everything else in OpenGL, any state you can set can also be queried. Program parameters can be queried with the `glGetProgramEnvParameter*ARB` and `glGetProgramLocalParameter*ARB` commands. You can read back your whole shader text via `glGetProgramStringARB`. You can check the current vertex attributes with `glGetVertexAttrib*ARB` or the vertex attribute array pointer with `glGetVertexAttribPointervARB`. To see whether a given name represents an existing shader, call `glIsProgramARB`. You can find the details of all these queries in the reference section.

Shader Options

The shader grammar and behavior can be altered with options. An `OPTION` line appears at the beginning of a shader before any real instructions. `GL_ARB_vertex_program` and `GL_ARB_fragment_program` provide the options listed in the following sections, but future extensions may introduce additional options.

Position-Invariant Vertex Option

Usually, a vertex shader is required to output to `result.position`. If this option is present, you cannot output to `result.position`, and instead the vertex is automatically transformed to clip space for you:

```
!!ARBvp1.0
OPTION ARB_position_invariant;
```

Using this option is not only a convenience when you don't need fancy vertex transformation, but it also helps ensure that the transformation will be the same with or without vertex shaders so you don't have to worry about precision artifacts when multipass rendering.

Fog Application Fragment Options

Each fog application fragment option is another convenience option. Fragment shaders subsume the fog stage and thus are responsible for performing their own fog computations. However, if you specify one of the three fog options shown here, the work will be done for you, just like in fixed functionality. You just choose linear, exponential, or second-order exponential:

```
!!ARBfp1.0
OPTION ARB_fog_linear;
OPTION ARB_fog_exp;      # You can't actually specify more than one of these
OPTION ARB_fog_exp2;    # in your shader or it will fail to parse!
```

Precision Hint Fragment Option

Some fragment shader hardware implementations may support multiple floating-point calculation and internal storage precisions that either exceed or fall short of OpenGL's minimum precision requirements. You can hint to the driver whether you would prefer your fragment shader to be run on more or less than the default precision by using one of these options. Remember, this is just a hint, and some implementations simply ignore the hint:

```
!!ARBfp1.0
OPTION ARB_precision_hint_nicest;      # Again, you can only have
OPTION ARB_precision_hint_fastest;      # one of these options
```

Summary

As you can tell by the length and density of this chapter, low-level shaders are a mix of rocket science and brain surgery. And we haven't even talked about any applications yet; that will start in [Chapter 22](#), "Vertex Shading: Do-It-Yourself Transform, Lighting, and Texgen." Seeing these shaders in action will make them less intimidating.

What we covered here is the mechanics of low-level shaders. You've been exposed to myriad instruction opcodes, variable types, and input and output modifiers. Finally, we discussed queries and shader options. Soon enough we'll put all this information to work for us.

Reference

glBindProgramARB

Purpose: **Binds a shader.**

Include File: `<glext.h>`

Syntax:

```
void glBindProgramARB(GLenum target, GLuint program);
```

Description: This function binds a low-level shader to the vertex shader or fragment shader target. If the shader has not been bound before, it is created. Subsequent changes to or queries of the target affect or return state from the bound shader.

Parameters:

target `GLenum`: The type of shader being bound. It can be one of the following constants:

`GL_VERTEX_PROGRAM_ARB`: Bind a vertex shader.

`GL_FRAGMENT_PROGRAM_ARB`: Bind a fragment shader.

program `GLuint`: The name of the shader to bind.

Returns: None.

See Also: `glDeleteProgramsARB`, `glProgramStringARB`, `glGenProgramsARB`

glDeleteProgramsARB

Purpose: Deletes one or more shaders.**Include File:** `<glext.h>`**Syntax:**

```
void glDeleteProgramsARB(GLsizei n, const GLuint
→ *programs);
```

Description: This function deletes shaders. The contents are deleted, and the names are marked as unused. If such a buffer object is currently bound in the current context, all such bindings are reset to zero. If an unused shader name or name zero is specified for deletion, that name is silently ignored.

Parameters:

n `GLsizei`: The number of shaders to delete.

programs `GLuint *:` Pointer to an array containing the names of the shaders to delete.

Returns: None.**See Also:** `glBindProgramARB`, `glProgramStringARB`, `glGenProgramsARB`, `glIsProgramARB`**glDisableVertexAttribArrayARB****Purpose:** Disables a vertex attribute array.**Include File:** `<glext.h>`**Syntax:**

```
void glDisableVertexAttribArrayARB(GLuint index);
```

Description: This function behaves like `glDisableClientState`, but because the number of vertex attributes is unbounded, conventional vertex array flags can't be used. Instead, a vertex attribute number is passed into this entrypoint indicating which vertex attribute array to disable.

Parameters:

index `GLuint`: The vertex attribute array number to disable.

Returns: None.**See Also:** `glEnableVertexAttribArrayARB`, `glVertexAttribPointerARB`**glEnableVertexAttribArrayARB****Purpose:** Enables a vertex attribute array.**Include File:** `<glext.h>`**Syntax:**

```
void glEnableVertexAttribArrayARB(GLuint index);
```

Description: This function behaves like `glEnableClientState`, but because the number of vertex attributes is unbounded, conventional vertex array flags can't be used. Instead, a vertex attribute number is passed into this entrypoint indicating which vertex attribute array to enable.

Parameters:

`index` `GLuint`: The vertex attribute array number to enable.

Returns: None.

See Also: `glDeleteProgramsARB`, `glProgramStringARB`, `glGenProgramsARB`

glGenProgramsARB

Purpose: Returns unused low-level shader names.

Include File: `<glext.h>`

Syntax:

```
void glGenProgramsARB(GLsizei n, GLuint *programs);
```

Description: This function returns unused shader names. The names can subsequently be bound with `glBindProgramARB`.

Parameters:

`n` `GLsizei`: The number of low-level shader names to return.

`programs` `GLuint *`: Pointer to an array to fill with unused shader names.

Returns: None.

See Also: `glBindProgramARB`, `glProgramStringARB`, `glDeleteProgramsARB`, `glIsProgramARB`

glGetProgramivARB

Purpose: Queries shader properties.

Include File: `<glext.h>`

Syntax:

```
void glGetProgramivARB(GLenum target, GLenum pname
→ , GLint *params);
```

Description: This function queries a property of the current low-level vertex or fragment shader.

Parameters:

target GLenum: The type of shader being queried. It can be one of the following constants:

`GL_VERTEX_PROGRAM_ARB`: Queries the current vertex shader.

`GL_FRAGMENT_PROGRAM_ARB`: Queries the current fragment shader.

pname GLenum: The shader property to query. It can be one of the following constants:

`GL_PROGRAM_LENGTH_ARB`: Length of the shader in bytes.

`GL_PROGRAM_FORMAT_ARB`: Format of the shader (always ASCII).

`GL_PROGRAM_BINDING_ARB`: Name of the currently bound shader.

`GL_PROGRAM_INSTRUCTIONS_ARB`: Number of parsed instructions in the current shader.

`GL_MAX_PROGRAM_INSTRUCTIONS_ARB`: Maximum parsable number of instructions.

`GL_PROGRAM_NATIVE_INSTRUCTIONS_ARB`: Number of native instructions in the current shader.

`GL_MAX_PROGRAM_NATIVE_INSTRUCTIONS_ARB`: Maximum available native instructions.

`GL_PROGRAM_TEMPORARIES_ARB`: Number of parsed temporaries in the current shader.

`GL_MAX_PROGRAM_TEMPORARIES_ARB`: Maximum parsable number of temporaries.

`GL_PROGRAM_NATIVE_TEMPORARIES_ARB`: Number of native temporaries in the current shader.

`GL_MAX_PROGRAM_NATIVE_TEMPORARIES_ARB`: Maximum available native temporaries.

`GL_PROGRAM_PARAMETERS_ARB`: Number of parsed parameters in the current shader.

`GL_MAX_PROGRAM_PARAMETERS_ARB`: Maximum parsable number of parameters.

`GL_PROGRAM_NATIVE_PARAMETERS_ARB`: Number of native parameters in the current shader.

`GL_MAX_PROGRAM_NATIVE_PARAMETERS_ARB`: Maximum available native parameters.

`GL_PROGRAM_ATTRIBS_ARB`: Number of parsed attributes in the current shader.

`GL_MAX_PROGRAM_ATTRIBS_ARB`: Maximum parsable number of attributes.

GL_PROGRAM_NATIVE_ATTRIBS_ARB: Number of native attributes in the current shader.

GL_MAX_PROGRAM_NATIVE_ATTRIBS_ARB: Maximum available native attributes.

GL_PROGRAM_ADDRESS_REGISTERS_ARB: Number of parsed address registers in the current shader (vertex shader only).

GL_MAX_PROGRAM_ADDRESS_REGISTERS_ARB: Maximum parsable number of address registers (vertex shader only).

GL_PROGRAM_NATIVE_ADDRESS_REGISTERS_ARB: Number of native address registers in the current shader (vertex shader only).

GL_MAX_PROGRAM_NATIVE_ADDRESS_REGISTERS_ARB: Maximum available native address registers (vertex shader only).

GL_MAX_PROGRAM_LOCAL_PARAMETERS_ARB: Maximum parsable number of program local parameters.

GL_MAX_PROGRAM_ENV_PARAMETERS_ARB: Maximum parsable number of program environment parameters.

GL_PROGRAM_UNDER_NATIVE_LIMITS_ARB: Zero if current shader exceeds any native resource limit; one otherwise.

GL_PROGRAM_ALU_INSTRUCTIONS_ARB: Number of parsed ALU instructions in the current shader (fragment shader only).

GL_MAX_PROGRAM_ALU_INSTRUCTIONS_ARB: Maximum parsable number of ALU instructions (fragment shader only).

GL_PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB: Number of native ALU instructions in the current shader (fragment shader only).

GL_MAX_PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB: Maximum available native ALU instructions (fragment shader only).

GL_PROGRAM_TEX_INSTRUCTIONS_ARB: Number of parsed texture instructions in the current shader (fragment shader only).

GL_MAX_PROGRAM_TEX_INSTRUCTIONS_ARB: Maximum parsable number of texture instructions (fragment shader only).

GL_PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB: Number of native texture instructions in the current shader (fragment shader only).

GL_MAX_PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB: Maximum available native texture instructions (fragment shader only).

GL_PROGRAM_TEX_INDIRECTIONS_ARB: Number of parsed texture indirections in the current shader (fragment shader only).

GL_MAX_PROGRAM_TEX_INDIRECTIONS_ARB: Maximum parsable number of texture indirections (fragment shader only).

GL_PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB: Number of native texture indirections in the current shader (fragment shader only).

GL_MAX_PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB: Maximum available native texture indirections (fragment shader only).

params `GLint *`: A pointer to the location where the query results are written.

Returns: None.

See Also: `glProgramStringARB`, `glGetProgramStringARB`

glGetProgramEnvParameter*vARB

Purpose: Queries a program environment parameter.

Include File: `<glext.h>`

Syntax:

```
void glGetProgramEnvParameterdvARB(GLenum target,
➥ GLuint index,
➥ GLdouble *params);
void glGetProgramEnvParameterfvARB(GLenum target,
➥ GLuint index,
➥ GLfloat *params);
```

Description: This function retrieves a single environment parameter from the collection shared by all vertex shaders or all fragment shaders. Either four single-precision floats or four double-precision floats are returned.

Parameters:

target `GLenum`: The type of shader whose environment parameter is being queried. It can be one of the following constants:

`GL_VERTEX_PROGRAM_ARB`: Query a vertex shader environment parameter.

`GL_FRAGMENT_PROGRAM_ARB`: Query a fragment shader environment parameter.

index `GLuint`: The number of the environment parameter vector to query.

params `GLdouble*/ GLfloat*`: A pointer to the location where the environment parameter vector is written.

Returns: None.

See Also: `glGetProgramLocalParameter*vARB`, `glProgramEnvParameter*ARB`

glGetProgramLocalParameter*vARB

Purpose: Queries a program local parameter.

Include File: `<glext.h>`

Syntax:

```
void glGetProgramLocalParameterdvARB(GLenum target
➥ , GLuint index,
➥ GLdouble
➥ *params);
void glGetProgramLocalParameterfvARB(GLenum target
➥ , GLuint index,
➥ GLfloat *params);
```

Description: This function retrieves a single local parameter from the currently bound vertex or fragment shader. Either four single-precision floats or four double-precision floats are returned.

Parameters:

target **GLenum:** The type of shader whose local parameter is being queried. It can be one of the following constants:

GL_VERTEX_PROGRAM_ARB: Query a local parameter from the current vertex shader.

GL_FRAGMENT_PROGRAM_ARB: Query a local parameter from the current fragment shader.

index **GLuint:** The number of the local parameter vector to query.

params **GLdouble*/ GLfloat*:** A pointer to the location where the local parameter vector is written.

Returns: None.

See Also: [glGetProgramEnvParameter*vARB](#), [glProgramLocalParameter*ARB](#)

glGetProgramStringARB

Purpose: Queries current shader text.

Include File: `<glext.h>`

Syntax:

```
void glGetProgramStringARB(GLenum target, GLenum
➥ pname, GLvoid *string);
```

Description: This function reads back the shader text from the currently bound vertex or fragment shader into the application-provided pointer. The application should first query the length of the current shader to make sure enough space is available to hold the entire shader.

Parameters:

target GLenum: The type of shader text being queried. It can be one of the following constants:

- GL_VERTEX_PROGRAM_ARB: Read back the vertex shader text.
- GL_FRAGMENT_PROGRAM_ARB: Read back the fragment shader text.

pname GLenum: The type of string to return. It must be GL_PROGRAM_STRING_ARB.

string GLvoid *: A pointer to the location where the shader text will be stored.

Returns: None.

See Also: [glProgramStringARB](#), [glGetProgramivARB](#)

glGetVertexAttrib*vARB

Purpose: Queries a property of a vertex attribute.

Include File: `<glext.h>`

Syntax:

```
void glGetVertexAttribdvARB(GLuint index, GLenum
  ↪ pname, GLdouble *params);
void glGetVertexAttribfvARB(GLuint index, GLenum
  ↪ pname, GLfloat *params);
void glGetVertexAttribivARB(GLuint index, GLenum
  ↪ pname, GLint *params);
```

Description: This function queries a property from the indicated numbered vertex attribute.

Parameters:

index GLuint: The number of the vertex attribute to query.

pname GLenum: The vertex attribute property to query. It can be one of the following constants:

GL_VERTEX_ATTRIB_ARRAY_ENABLED_ARB: One if the vertex attribute array is enabled; zero if not.

GL_VERTEX_ATTRIB_ARRAY_SIZE_ARB: Number of components per vertex attribute array element.

GL_VERTEX_ATTRIB_ARRAY_STRIDE_ARB: Stride between vertex attribute array elements.

GL_VERTEX_ATTRIB_ARRAY_TYPE_ARB: Data type of vertex attribute array elements.

GL_VERTEX_ATTRIB_ARRAY_NORMALIZED_ARB: One if fixed-point data is normalized when converted to floating-point; zero otherwise.

GL_CURRENT_VERTEX_ATTRIB_ARB: Current vertex attribute state.

params `GLdouble*/ GLfloat*/ GLint*`: A pointer to the location where the query result will be stored.

Returns: None.

See Also: `glVertexAttrib*ARB`, `glVertexAttribPointerARB`, `glGetVertexAttribPointervARB`

glGetVertexAttribPointervARB

Purpose: Queries a vertex attribute array pointer.

Include File: `<glext.h>`

Syntax:

```
void glGetVertexAttribPointervARB(GLuint index,
→ GLenum pname,
                           GLvoid **pointer);
```

Description: This function gets the data pointer of the specified vertex attribute array.

Parameters:

index `GLuint`: The number of the vertex attribute array pointer being queried.

pname `GLenum`: The property of the vertex attribute array being queried. It must be `GL_VERTEX_ATTRIB_ARRAY_POINTER_ARB`.

pointer `GLvoid **`: A pointer to the location where the queried vertex attribute array pointer will be stored.

Returns: None.

See Also: `glVertexAttribPointerARB`, `glGetVertexAttrib*ARB`

glIsProgramARB

Purpose: Queries whether a name is a shader name.

Include File: `<glext.h>`

Syntax:

```
GLboolean glIsProgramARB(GLuint program);
```

Description: This function queries whether the specified name is the name of a shader.

Parameters:

program `GLuint`: The shader name to be queried.

Returns: `GLboolean`: `GL_TRUE` is returned if a shader with this name has previously been bound and not yet deleted. Otherwise, `GL_FALSE` is returned.

See Also: [glBindProgramARB](#), [glDeleteProgramsARB](#), [glGenProgramsARB](#)

glProgramEnvParameter*ARB

Purpose: Sets a program environment parameter.

Include File: `<glext.h>`

Syntax:

```
void glProgramEnvParameter4dARB(GLenum target,
    ➔ GLuint index,
                    GLdouble x,
    ➔ GLdouble y, GLdouble z, GLdouble w);
void glProgramEnvParameter4dvARB(GLenum target,
    ➔ GLuint index,
                    const GLdouble
    ➔ *params);
void glProgramEnvParameter4fARB(GLenum target,
    ➔ GLuint index,
                    GLfloat x, GLfloat
    ➔ y, GLfloat z, GLfloat w);
void glProgramEnvParameter4fvARB(GLenum target,
    ➔ GLuint index,
                    const GLfloat
    ➔ *params);
```

Description: This function sets a single program environment parameter of the collection shared by all vertex shaders or all fragment shaders.

Parameters:

target `GLenum`: The type of shader whose program environment parameter is being set. It can be one of the following constants:

`GL_VERTEX_PROGRAM_ARB`: Set a vertex shader program environment parameter.

`GL_FRAGMENT_PROGRAM_ARB`: Set a fragment shader program environment parameter.

index `GLuint`: The number of the program environment parameter vector to set.

x, y, z, w `GLdouble/ GLfloat`: The components of the new program environment parameter.

params `GLdouble */ GLfloat *`: A pointer to the four components of the new program environment parameter.

Returns: None.

See Also: [glProgramLocalParameter*ARB](#), [glGetProgramEnvParameter*ARB](#)

glProgramLocalParameter*ARB**Purpose:** Sets a program local parameter.**Include File:** `<glext.h>`**Syntax:**

```
void glProgramLocalParameter4dARB(GLenum target,
    ➔ GLuint index,
                GLdouble x,
    ➔ GLdouble y, GLdouble z, GLdouble w);
void glProgramLocalParameter4dvARB(GLenum target,
    ➔ GLuint index,
                const GLdouble
    ➔ *params);
void glProgramLocalParameter4fARB(GLenum target,
    ➔ GLuint index,
                GLfloat x,
    ➔ GLfloat y, GLfloat z, GLfloat w);
void glProgramLocalParameter4fvARB(GLenum target,
    ➔ GLuint index,
                const GLfloat
    ➔ *params);
```

Description: This function sets a single local parameter of the currently bound vertex or fragment shader.**Parameters:**

target `GLenum`: The type of shader whose local parameter is being set. It can be one of the following constants:

`GL_VERTEX_PROGRAM_ARB`: Set a local parameter of the current vertex shader.

`GL_FRAGMENT_PROGRAM_ARB`: Set a local parameter of the current fragment shader.

index `GLuint`: The number of the program local parameter vector to set.

x, y, z, w `GLdouble/ GLfloat`: The components of the new program local parameter.

params `GLdouble */ GLfloat *`: A pointer to the four components of the new program local parameter.

Returns: None.**See Also:** `glProgramEnvParameter*ARB`, `glGetProgramLocalParameter*ARB`**glProgramStringARB****Purpose:** Sets the low-level shader text.**Include File:** `<glext.h>`**Syntax:**

```
void glProgramStringARB(GLenum target, GLenum
  ↪ format, GLsizei len,
                const GLvoid *string);
```

Description: This function sends new shader text into OpenGL, where it is parsed, compiled, and optimized. If successful, the new shader text replaces any pre-existing shader. However, if the new shader fails to parse or compile, the current shader, if any, remains in place.

Parameters:

target GLenum: The type of shader to replace with new shader text:
 GL_VERTEX_PROGRAM_ARB: Set shader text for the currently bound vertex shader.
 GL_FRAGMENT_PROGRAM_ARB: Set shader text for the currently bound fragment shader.
format GLenum: The format of the shader text, GL_PROGRAM_FORMAT_ASCII_ARB.
len GLsizei: The length of the shader text, measured in bytes, not including any null terminator at the end of the string.
string GLvoid *: A pointer to the new shader text.

Returns: None.

See Also: glBindProgramARB, glGenProgramsARB, glDeleteProgramsARB

glVertexAttrib*ARB

Purpose: Sets generic vertex attribute.

Include File: <glext.h>

Syntax:

```
void glVertexAttrib1sARB(GLuint index, GLshort x);
void glVertexAttrib1fARB(GLuint index, GLfloat x);
void glVertexAttrib1dARB(GLuint index, GLdouble x);
void glVertexAttrib2sARB(GLuint index, GLshort x,
  ↪ GLshort y);
void glVertexAttrib2fARB(GLuint index, GLfloat x,
  ↪ GLfloat y);
void glVertexAttrib2dARB(GLuint index, GLdouble x,
  ↪ GLdouble y);
void glVertexAttrib3sARB(GLuint index, GLshort x,
  ↪ GLshort y, GLshort z);
void glVertexAttrib3fARB(GLuint index, GLfloat x,
  ↪ GLfloat y, GLfloat z);
void glVertexAttrib3dARB(GLuint index, GLdouble x,
  ↪ GLdouble y, GLdouble z);
void glVertexAttrib4sARB(GLuint index,
                      GLshort x, GLshort y,
  ↪ GLshort z, GLshort w);
```

```

void glVertexAttrib4fARB(GLuint index,
                         GLfloat x, GLfloat y,
                         GLfloat z, GLfloat w);
void glVertexAttrib4dARB(GLuint index,
                         GLdouble x, GLdouble y,
                         GLdouble z, GLdouble w);
void glVertexAttrib4NubARB(GLuint index,
                           GLubyte x, GLubyte y,
                           GLubyte z, GLubyte w);
void glVertexAttrib1svARB(GLuint index, const
                           GLshort *v);
void glVertexAttrib1fvARB(GLuint index, const
                           GLfloat *v);
void glVertexAttrib1dvARB(GLuint index, const
                           GLdouble *v);
void glVertexAttrib2svARB(GLuint index, const
                           GLshort *v);
void glVertexAttrib2fvARB(GLuint index, const
                           GLfloat *v);
void glVertexAttrib2dvARB(GLuint index, const
                           GLdouble *v);
void glVertexAttrib3svARB(GLuint index, const
                           GLshort *v);
void glVertexAttrib3fvARB(GLuint index, const
                           GLfloat *v);
void glVertexAttrib3dvARB(GLuint index, const
                           GLdouble *v);
void glVertexAttrib4bvARB(GLuint index, const
                           GLbyte *v);
void glVertexAttrib4svARB(GLuint index, const
                           GLshort *v);
void glVertexAttrib4ivARB(GLuint index, const
                           GLint *v);
void glVertexAttrib4ubvARB(GLuint index, const
                           GLubyte *v);
void glVertexAttrib4usvARB(GLuint index, const
                           GLushort *v);
void glVertexAttrib4uivARB(GLuint index, const
                           GLuint *v);
void glVertexAttrib4fvARB(GLuint index, const
                           GLfloat *v);
void glVertexAttrib4dvARB(GLuint index, const
                           GLdouble *v);
void glVertexAttrib4NbvARB(GLuint index, const
                           GLbyte *v);
void glVertexAttrib4NsvARB(GLuint index, const
                           GLshort *v);
void glVertexAttrib4NivARB(GLuint index, const
                           GLint *v);
void glVertexAttrib4NubvARB(GLuint index, const
                           GLubyte *v);
void glVertexAttrib4NusvARB(GLuint index, const
                           GLushort *v);
void glVertexAttrib4NuivARB(GLuint index, const
                           GLuint *v);

```

```
→ GLuint *v);
```

Description: This function sets a generic vertex attribute of the current vertex. The versions with *N* first normalize the values to the range [0,1] for unsigned types or [-1,1] for signed types.

Parameters:

index **GLuint**: The number of the generic vertex attribute to set.

x, y, z, w all types: The components of the new generic vertex attribute. If *y*, *z*, or *w* is unspecified, those components are padded with 0, 0, and 1, respectively.

v all type pointers: A pointer to the components of the new generic vertex attribute.

Returns: None.

See Also: [glVertexAttribPointerARB](#), [glGetVertexAttrib*vARB](#)

glVertexAttribPointerARB

Purpose: Sets the pointer and other properties of a generic vertex attribute array.

Include File: `<glext.h>`

Syntax:

```
void glVertexAttribPointerARB(GLuint index, GLint
→ size, GLenum type,
    GLboolean normalized, GLsizei stride, const
→ GLvoid *pointer);
```

Description: This function sets the array pointer, size, type, and stride for a generic vertex attribute. It will optionally normalize integer type values to the range [0,1] for unsigned types or [-1,1] for signed types.

Parameters:

index **GLuint**: The number of the generic vertex attribute array pointer being set.

size **GLint**: The number of components in each element of the generic vertex attribute array.

type **GLenum**: The type of the generic vertex attribute array data. It can be one of the following constants:

GL_DOUBLE: Double-precision floating-point.

GL_FLOAT: Single-precision floating-point.

GL_BYTE: Signed one-byte integer.

GL_SHORT: Signed two-byte integer.

GL_INT: Signed four-byte integer.

`GL_UBYTE`: Unsigned one-byte integer.

`GL USHORT`: Unsigned two-byte integer.

`GL_UINT`: Unsigned four-byte integer.

`normalized` `GLboolean`: `GL_TRUE` if the fixed-point data should be normalized; `GL_FALSE` otherwise.

`stride` `GLsizei`: The number of bytes to skip between elements of the array.

`pointer` `const GLvoid *`: A pointer to the generic vertex attribute array data.

Returns: None.

See Also: `glGetVertexAttribPointervARB`, `glVertexAttrib*ARB`

Chapter 21. High-Level Shading: The Real Slim Shader

by Benjamin Lipchak

WHAT YOU'LL LEARN IN THIS CHAPTER:

How To	Functions You'll Use
Create shader/program objects	<code>glCreateShaderObjectARB</code> / <code>glCreateProgramObjectARB</code>
Specify shaders and compile	<code>glShaderSourceARB</code> / <code>glCompileShaderARB</code>
Attach/detach shaders and link	<code>glAttachObjectARB</code> / <code>glDetachObjectARB</code> / <code>glLinkProgramARB</code>
Switch between programs	<code>glUseProgramObjectARB</code>
Specify a uniform	<code>glUniform*ARB</code>
Get error and warning information	<code>glGetInfoLogARB</code>

You could, in theory, write all your applications in assembly language. But you don't, and there are good reasons for not doing so: development time efficiency, readability, maintainability, and portability, to name just a few. The benefits of assembly are becoming scarce with today's high-level language compilers generating code that performs as well as, or even outperforms, hand-written assembly.

So then why did we bother discussing low-level shading in the preceding chapters instead of skipping straight to high-level shaders, based on the OpenGL Shading Language (GLSL)? The answer is based on availability. The ARB-standardized low-level shader extensions, covered in [Chapter 20](#), "Low-Level Shading: Coding to the Metal," have been available for years and have been the only game in town. Though in development for years, no high-level alternative was available on OpenGL until recently. So game and application developers who wanted to incorporate shading capabilities embraced the low-level extensions as their only choice at the time.

The adoption of low-level extensions was relatively rapid because these extensions exposed functionality not previously available. It is likely that the migration to GLSL will be slower because high-level shaders, for the most part, expose only equivalent functionality, just in an easier-to-use way. Low-level shaders, on the other hand, are entrenched with a two-year headstart. Applications with long development cycles may take the "if it ain't broke, don't fix it" mentality, keeping low-level shaders alive and kicking for years to come. But in the long run, GLSL will be the tool of choice, and low-level shaders will be a footnote in OpenGL history.

Managing High-Level Shaders

The GLSL ARB extensions are quite a departure from the core OpenGL API and other extensions in terms of the API they introduce. No more of the Gen/Bind/Delete that you're used to; GLSL gets all new entrypoints.

Shader Objects

GLSL uses two types of objects: shader objects and program objects. The first objects we will look at, shader objects, are loaded with shader text and compiled.

Creating and Deleting

You create shader objects by calling `glCreateShaderObjectARB` and passing it the type of shader you want, either vertex or fragment. This function returns a handle to the shader object used to reference it in subsequent calls:

```
GLhandleARB myVertexShader = glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
GLhandleARB myFragmentShader = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);
```

Beware that object creation may fail, in which case 0 (zero) will be returned. This may happen if OpenGL runs out of memory resources or object handles. When you're done with your shader objects, you should clean up after yourself:

```
glDeleteObjectARB(myVertexShader);
glDeleteObjectARB(myFragmentShader);
```

Some other OpenGL objects, including texture objects, unbind an object during deletion if it's currently in use. GLSL objects are different. `glDeleteObjectARB` simply marks the object for future deletion, which will occur as soon as the object is no longer being used.

Specifying Shader Text

A shader object's goal is simply to accept shader text and compile it. Your shader text can be hard-coded as a string-literal, read from a file on disk, or generated on the fly. One way or another, it needs to be in a string so you can load it into your shader object:

```
GLcharARB *myStringPtrs[1];
myStringPtrs[0] = vertexShaderText;
glShaderSourceARB(myVertexShader, 1, myStringPtrs, NULL);
myStringPtrs[0] = fragmentShaderText;
glShaderSourceARB(myFragmentShader, 1, myStringPtrs, NULL);
```

`glShaderSourceARB` is set up to accept multiple individual strings. The second argument is a count that indicates how many string pointers to look for. The strings are strung together into a single long string before being compiled. This capability can be useful if you're loading reusable subroutines from a library of functions:

```
GLcharARB *myStringPtrs[3];
myStringPtrs[0] = vsMainText;
myStringPtrs[1] = myNoiseFuncText;
myStringPtrs[2] = myBlendFuncText;
glShaderSourceARB(myVertexShader, 3, myStringPtrs, NULL);
```

If all your strings are null-terminated, you don't need to specify string lengths in the fourth argument and can pass in `NULL` instead. But for shader text that is not null-terminated, you need to provide the length; lengths do not include the null terminator if present. You can use -1 for

strings that are null-terminated. The following code passes in a pointer to an array of lengths along with the array of string pointers:

```
GLint fsLength = strlen(fragmentShaderText);
myStringPtrs[0] = fragmentShaderText;
glShaderSourceARB(myFragmentShader, 1, myStringPtrs, &fsLength);
```

Compiling Shaders

After your shader text is loaded into a shader object, you need to compile it. Compiling parses your shader and makes sure there are no errors:

```
glCompileShaderARB(myVertexShader);
glCompileShaderARB(myFragmentShader);
```

You can query a flag in each shader object to see whether the compile was successful. Each shader object also has an information log that contains error messages if the compilation failed. It might also contain warnings or other useful information even if your compilation was successful. These logs are primarily intended for use as a tool while you are developing your GLSL application:

```
glCompileShaderARB(myVertexShader);
glGetObjectParameterivARB(myVertexShader, GL_OBJECT_COMPILE_STATUS_ARB,
                         &success);
if (!success)
{
    GLbyte infoLog[MAX_INFO_LOG_SIZE];
    glGetInfoLogARB(myVertexShader, MAX_INFO_LOG_SIZE, NULL, infoLog);
    fprintf(stderr, "Error in vertex shader compilation!\n");
    fprintf(stderr, "Info log: %s\n", infoLog);
    Sleep(10000);
    exit(0);
}
```

The returned info log string is always null terminated. If you don't want to allocate a large static array to store the info log, you can find out the exact size of the info log before querying it:

```
glGetObjectParameterivARB(myVertexShader, GL_OBJECT_INFO_LOG_LENGTH_ARB,
                         &infoLogSize);
```

Program Objects

The second type of object GLSL uses is a program object. This object acts as a container for shader objects, linking them together into a single executable. You can specify a GLSL shader for each replaceable section of the conventional OpenGL pipeline. Currently, only vertex and fragment stages are replaceable, but that list could be extended in the future to include additional stages.

Creating and Deleting

Program objects are created and deleted the same way as shader objects. The difference is that there's only one kind of program object, so its creation entrypoint doesn't take an argument:

```
GLhandleARB myProgram = glCreateProgramObjectARB();
...
glDeleteObjectARB(myProgram);
```

Attaching and Detaching

A program object is a container. You need to attach your shader objects to it if you want GLSL instead of fixed functionality:

```
glAttachObjectARB(myProgram, myVertexShader);
glAttachObjectARB(myProgram, myFragmentShader);
```

You can even attach multiple shader objects of the same type to your program object. Similar to loading multiple shader source strings into a single shader object, this makes it possible to include function libraries shared by more than one of your program objects.

You can choose to replace only part of the pipeline with GLSL and leave the rest to fixed functionality. Just don't attach shaders for the parts you want to leave alone. Or if you're switching between GLSL and fixed functionality for part of the pipeline, you can detach a previously attached shader object. You can even detach both shaders, in which case you're back to full fixed functionality:

```
glDetachObjectARB(myProgram, myVertexShader);
glDetachObjectARB(myProgram, myFragmentShader);
```

Linking Programs

Before you can use GLSL for rendering, you have to link your program object. This process takes each of the previously compiled shader objects and links them into a single executable:

```
glLinkProgramARB(myProgram);
```

You can query a flag in the program object to see whether the link was successful. The object also has an information log that contains error messages if the link failed. The log might also contain warnings or other useful information even if your link was successful:

```
glLinkProgramARB(myProgram);
glGetObjectParameterivARB(myProgram, GL_OBJECT_LINK_STATUS_ARB, &success);
if (!success)
{
    GLbyte infoLog[MAX_INFO_LOG_SIZE];
    glGetInfoLogARB(myProgram, MAX_INFO_LOG_SIZE, NULL, infoLog);
    fprintf(stderr, "Error in program linkage!\n");
    fprintf(stderr, "Info log: %s\n", infoLog);
    Sleep(10000);
    exit(0);
}
```

Validating Programs

If your link was successful, odds are good that your shaders will be executable when it comes time to render. But some things aren't known at link time, such as the values assigned to texture samplers, described in subsequent sections. A sampler may be set to an invalid value, or multiple samplers of different types may be illegally set to the same value. At link time, you don't know what the state is going to be when you render, so errors cannot be thrown at that time. When you validate, however, it looks at the current state so you can find out once and for all whether your GLSL shaders are going to execute when you draw that first triangle:

```
glValidateProgramARB(myProgram);
glGetObjectParameterivARB(myProgram, GL_OBJECT_VALIDATE_STATUS_ARB, &success);
if (!success)
{
    GLbyte infoLog[MAX_INFO_LOG_SIZE];
    glGetInfoLogARB(myProgram, MAX_INFO_LOG_SIZE, NULL, infoLog);
    fprintf(stderr, "Error in program validation!\n");
    fprintf(stderr, "Info log: %s\n", infoLog);
    Sleep(10000);
    exit(0);
```

```
}
```

Again, if the validation fails, an explanation and possibly tips for avoiding the failure are included in the program object's info log. Note that validating your program object before rendering with it is not a requirement, but if you do try to use a program object that would have failed validation, your rendering commands will fail and throw OpenGL errors.

Using Programs

Finally, you're ready to turn on your program. Unlike other OpenGL features, GLSL mode is not toggled with `glEnable/glDisable`:

```
glUseProgramObject(myProgram);
```

You can use this function to enable GLSL with a particular program object and also to switch between different program objects. To disable GLSL and switch back to fixed functionality, you also use this function, passing in 0:

```
glUseProgramObject(0);
```

You can query for the current program object handle at any time:

```
currentProgObj = glGetHandleARB(GL_PROGRAM_OBJECT_ARB);
```

Now that you know how to manage your shaders, you can focus on their contents. The syntax of GLSL is largely the same as that of C/C++, so you should be able to dive right in.

Setting Up the Extensions

Like low-level shader extensions, GLSL is not yet part of core OpenGL at the time of this book's printing. Therefore, the presence of the desired extensions must be queried, and entrypoint function pointers must also be obtained, as in [Listing 21.1](#).

You must check four GLSL extensions: `GL_ARB_shader_objects`, `GL_ARB_vertex_shader`, `GL_ARB_fragment_shader`, and `GL_ARB_shading_language_100`. The first contains the shared functionality between vertex and fragment shaders. The last reflects the version of the OpenGL Shading Language available. Updated versions will likely be made available over time.

Listing 21.1. Checking for the Presence of OpenGL Features

```
// Make sure required functionality is available!
if (!gltIsExtSupported("GL_ARB_vertex_shader") ||
    !gltIsExtSupported("GL_ARB_fragment_shader") ||
    !gltIsExtSupported("GL_ARB_shader_objects") ||
    !gltIsExtSupported("GL_ARB_shading_language_100"))
{
    fprintf(stderr, "GLSL extensions not available!\n");
    Sleep(2000);
    exit(0);
}
glCreateShaderObjectARB = gltGetExtensionPointer("glCreateShaderObjectARB");
glCreateProgramObjectARB = gltGetExtensionPointer("glCreateProgramObjectARB");
glAttachObjectARB = gltGetExtensionPointer("glAttachObjectARB");
glDetachObjectARB = gltGetExtensionPointer("glDetachObjectARB");
glDeleteObjectARB = gltGetExtensionPointer("glDeleteObjectARB");
glShaderSourceARB = gltGetExtensionPointer("glShaderSourceARB");
glCompileShaderARB = gltGetExtensionPointer("glCompileShaderARB");
glLinkProgramARB = gltGetExtensionPointer("glLinkProgramARB");
```

```

glValidateProgramARB = gltGetExtensionPointer("glValidateProgramARB");
glUseProgramObjectARB = gltGetExtensionPointer("glUseProgramObjectARB");
glGetObjectParameterivARB = gltGetExtensionPointer("glGetObjectParameterivARB");
glGetInfoLogARB = gltGetExtensionPointer("glGetInfoLogARB");
glUniform1fARB = gltGetExtensionPointer("glUniform1fARB");
glGetUniformLocationARB = gltGetExtensionPointer("glGetUniformLocationARB");
if (!glCreateShaderObjectARB || !glCreateProgramObjectARB ||
    !glAttachObjectARB || !glDetachObjectARB || !glDeleteObjectARB ||
    !glShaderSourceARB || !glCompileShaderARB || !glLinkProgramARB ||
    !glValidateProgramARB || !glUseProgramObjectARB ||
    !glGetObjectParameterivARB || !glGetInfoLogARB ||
    !glUniform1fARB || !glGetUniformLocationARB)
{
    fprintf(stderr, "Not all entrypoints were available!\n");
    Sleep(2000);
    exit(0);
}

```

Variables

Variables and functions must be declared in advance. Your variable name can use any letters (case-sensitive), numbers, or an underscore, but it can't begin with a number. Also, your variable cannot begin with the prefix `gl_`, which is reserved for built-in variables and functions. A list of reserved keywords available in the OpenGL Shading Language specification is also off-limits.

Basic Types

In addition to the Boolean, integer, and floating-point types found in C, GLSL introduces some data types commonly used in shaders. [Table 21.1](#) lists these basic data types.

Table 21.1. Basic Data Types

Type	Description
<code>void</code>	A data type required for functions that don't return a value. Functions that take no arguments can optionally use <code>void</code> as well.
<code>bool</code>	A Boolean variable used primarily for conditionals and loops. It can be assigned to keywords <code>true</code> and <code>false</code> , or any expression that evaluates to a Boolean.
<code>int</code>	A variable represented by a signed integer with at least 16 bits. It can be expressed in decimal, octal, or hexadecimal. It is primarily used for loop counters and array indexing.
<code>float</code>	A floating-point variable approximating IEEE single-precision. It can be expressed in scientific notation (for example, $0.0001 = 1e-4$).
<code>bvec2</code>	A two-component Boolean vector.
<code>bvec3</code>	A three-component Boolean vector.
<code>bvec4</code>	A four-component Boolean vector.
<code>ivec2</code>	A two-component integer vector.
<code>ivec3</code>	A three-component integer vector.
<code>ivec4</code>	A four-component integer vector.

<code>Vec2</code>	A two-component floating-point vector.
<code>Vec3</code>	A three-component floating-point vector.
<code>Vec4</code>	A four-component floating-point vector.
<code>Mat2</code>	A 2x2 floating-point matrix. Matrices are accessed in column-major order.
<code>Mat3</code>	A 3x3 floating-point matrix.
<code>Mat4</code>	A 4x4 floating-point matrix.
<code>sampler1D</code>	A special-purpose constant used by built-in texture functions to reference a specific 1D texture. It can be declared only as a uniform or function argument.
<code>sampler2D</code>	A constant used for referencing a 2D texture.
<code>sampler3D</code>	A constant used for referencing a 3D texture.
<code>samplerCube</code>	A constant used for referencing a cube map texture.
<code>sampler1DShadow</code>	A constant used for referencing a 1D depth texture with shadow comparison.
<code>sampler2DShadow</code>	A constant used for referencing a 2D depth texture with shadow comparison.

Structures

Structures can be used to group basic data types into a user-defined data type. When defining the structure, you can declare instances of the structure at the same time, or you can declare them later:

```
struct surface {
    float indexOfRefraction;
    float reflectivity;
    vec3 color;
    float turbulence;
} myFirstSurf;
surface mySecondSurf;
```

You can assign one structure to another (`=`) or compare two structures (`==`, `!=`). For both of these operations, the structures must be of the same declared type. Two structures are considered equal if each of their member fields is component-wise equal. To access a single field of a structure, you use the selector `(.)`:

```
vec3 totalColor = myFirstSurf.color + mySecondSurf.color;
```

Structure definitions must contain at least one member. Arrays, discussed next, may be included in structures, but only when a specific array size is provided. Unlike C language structures, GLSL does not allow bit fields. Structures within structures are allowed as long as the member structure is declared in advance or in place, not later in the shader:

```
struct superSurface {
    vec3 points[30];      // Sized arrays are okay
    surface surf;         // Okay, as surface was defined earlier
    struct velocity {     // Okay, velocity struct defined in place
        float speed;
        vec3 direction;
    } velo;
    subSurface sub;       // ILLEGAL!!  Forward declaration
};
struct subSurface {
    int id;
};
```

Arrays

One-dimensional arrays of any type (including structures) can be declared. You don't need to declare the size of the array as long as it is always indexed with a constant integer expression. Otherwise, you must declare its size up front:

```
vec4 lightPositions[8];
surface mySurfaces[];
const int numSurfaces = 5;
surface myFiveSurfaces[numSurfaces];
```

You also must declare an explicit size for your array when the array is declared as a parameter in a function declaration or as a member of a structure.

Qualifiers

Variables can be declared with an optional type qualifier. [Table 21.2](#) lists the available qualifiers.

Table 21.2. Type Qualifiers

Qualifier	Description
<code>const</code>	Constant value initialized during declaration. It is read-only during shader execution. It is also used with a function call argument to indicate it's a constant that can't be written within the function.
<code>attribute</code>	Read-only per-vertex data, available only within vertex shaders. This data comes from current vertex state or from vertex arrays. It must be declared globally (outside all functions).
<code>uniform</code>	Another value that remains constant during each shader execution, but unlike a <code>const</code> , a <code>uniform</code> 's value is not known at compile time and is initialized outside the shader. A <code>uniform</code> is shared by the currently active vertex and fragment shaders and must be declared globally.
<code>varying</code>	Outputs of the vertex shader, such as colors or texture coordinates, that correspond to read-only interpolated inputs of the fragment shader. They must be declared globally.
<code>in</code>	A qualifier used with a function call argument to indicate it's only an input, and any changes to the variable within the called function shouldn't affect the value in the calling function. This is the default behavior for function arguments if no qualifier is present.
<code>out</code>	A qualifier used with a function call argument to indicate it's only an output, so no value needs to be actually passed into the function.
<code>inout</code>	A qualifier used with a function call argument to indicate it's both an input and an output. A value is passed in from the calling function, and that value is replaced by the called function.

Built-in Variables

Built-in variables allow interaction with fixed functionality. They don't need to be declared before use. [Tables 21.3](#) and [21.4](#) list most of the built-in variables. Refer to the GLSL specification for built-in uniforms and constants.

Table 21.3. Built-in Vertex Shader Variables

Name	Type	Description
gl_Position	vec4	Output for transformed vertex position that will be used by fixed functionality primitive assembly, clipping, and culling; all vertex shaders must write to this variable.
gl_PointSize	float	Output for the size of the point to be rasterized, measured in pixels.
gl_ClipVertex	vec4	Output for the coordinate to use for user clip-plane clipping.
gl_Color	vec4	Input attribute corresponding to per-vertex primary color.
gl_SecondaryColor	vec4	Input attribute corresponding to per-vertex secondary color.
gl_Normal	vec3	Input attribute corresponding to per-vertex normal.
gl_Vertex	vec4	Input attribute corresponding to object-space vertex position.
gl_MultiTexCoordn	vec4	Input attribute corresponding to per-vertex texture coordinate <i>n</i> .
gl_FogCoord	float	Input attribute corresponding to per-vertex fog coordinate.
gl_FrontColor	vec4	Varying output for front primary color.
gl_BackColor	vec4	Varying output for back primary color.
gl_FrontSecondaryColor	vec4	Varying output for front secondary color.
gl_BackSecondaryColor	vec4	Varying output for back secondary color.
gl_TexCoord[]	vec4	Array of varying outputs for texture coordinates.
gl_FogFragCoord	float	Varying output for the fog coordinate.

Table 21.4. Built-in Fragment Shader Variables

Name	Type	Description
gl_FragCoord	vec4	Read-only input containing the window-space <i>x</i> , <i>y</i> , <i>z</i> , and <i>1/w</i> .
gl_FrontFacing	bool	Read-only input whose value is true if part of a front-facing primitive.
gl_FragColor	vec4	Output for the color to use for subsequent per-pixel operations.
gl_FragDepth	float	Output for the depth to use for subsequent per-pixel operations; if unwritten, the fixed functionality depth is used instead.
gl_Color	vec4	Interpolated read-only input containing the primary color.
gl_SecondaryColor	vec4	Interpolated read-only input containing the secondary color.
gl_TexCoord[]	vec4	Array of interpolated read-only inputs containing texture coordinates.
gl_FogFragCoord	float	Interpolated read-only input containing the fog coordinate.

Expressions

The following sections describe various operators and expressions found in GLSL.

Operators

All the familiar C operators are available in GLSL with few exceptions. See [Table 21.5](#) for a complete list.

Table 21.5. Operators in Order of Precedence (From Highest to Lowest)

Operator	Description
()	Parenthetical grouping, function call or constructor
[]	Array subscript, vector or matrix selector
.	Structure field selector, vector component selector
++ --	Prefix or post-fix increment and decrement
+ - !	Unary addition, subtraction, logical NOT
* /	Multiplication and division
+ -	Binary addition and subtraction
< > <= >= == !=	Less than, greater than, less than or equal to, greater than or equal to, equal to, not equal to
&& ^	Logical AND, OR, XOR
? :	Conditional
= += -= *= /=	Assignment, arithmetic assignments
,	Sequence

A few operators are missing from GLSL. Because there are no pointers to worry about, you don't need an address-of operator (&) or a dereference operator (*). A typecast operator is not needed because typecasting is not allowed. Bit-wise operators (&, |, ^, ~, <<, >>, &=, |=, ^=, <<=, >>=) are reserved for future use, as are modulus operators (%, %=).

Array Access

Arrays are indexed using integer expressions, with the first array element at index 0. Shader execution is undefined if an attempt is made to access an array with an index less than zero or greater than or equal to the size of the array:

```
vec4 myFifthColor, ambient, diffuse[6], specular[6];
...
myFifthColor = ambient + diffuse[5] + specular[5];
//Here's how NOT to index an array!
vec4 mySyntaxError=ambient + diffuse[-1] + specular[6];
```

Constructors

Constructors are special functions primarily used to initialize variables, especially of multicomponent data types, but not arrays. They take the form of a function call with the name of the function being the same as the name of the type:

```
vec3 myNormal = vec3(0.0, 1.0, 0.0);
```

Constructors are not limited to declaration initializers; they can be used as expressions anywhere in your shader:

```
greenTint = myColor + vec3(0.0, 1.0, 0.0);
```

A single scalar value is assigned to all elements of a vector:

```
ivec4 myColor = ivec4(255); // all 4 components get 255
```

You can mix and match scalars, vectors, and matrices in your constructor, as long as you end up with enough components to initialize the entire data type. Any extra components are dropped:

```
vec4 myVector1 = vec4(x, vec2(y, z), w);
vec2 myVector2 = vec2(myVector1); // z, w are dropped
float myFloat = float(myVector2); // y dropped
```

Matrices are constructed in column-major order. If you provide a single scalar value, that value is used for the diagonal matrix elements, and all other elements are set to 0:

```
// all of these are same 2x2 identity matrix
mat2 myMatrix1 = mat2(1.0, 0.0, 0.0, 1.0);
mat2 myMatrix2 = mat2(vec2(1.0, 0.0), vec2(0.0, 1.0));
mat2 myMatrix3 = mat2(1.0);
```

You can also use constructors to convert between the different scalar types. This is the only way to perform type conversions. No implicit or explicit type casts or promotions are possible.

The conversion from `int` to `float` is obvious. When you are converting from `float` to `int`, the fractional part is dropped. When you are converting from `int` or `float` to `bool`, values of 0 or 0.0 are converted to `false`, and anything else is converted to `true`. When you are converting from `bool` to `int` or `float`, `true` is converted to 1 or 1.0, and `false` is converted to 0 or 0.0:

```
float myFloat = 4.7;
int myInt = int(myFloat); // myInt = 4
bool myBool = bool(myInt); // myBool = true
myFloat = float(myBool); // myFloat = 1
```

Finally, you can initialize structures by providing arguments in the same order and of the same type as the structure definition:

```
struct surface {
    float indexOfRefraction;
    float reflectivity;
    vec3 color;
    float turbulence;
};
surface mySurf = surface(ior, refl, vec3(red, green, blue), turb);
```

Component Selectors

Individual components of a vector can be accessed by using dot notation along with `{x,y,z,w}`,

{**r,g,b,a**}, or {**s,t,p,q**}. These different notations are useful for positions and normals, colors, and texture coordinates, respectively. Notice the **p** in place of the usual **r** texture coordinate. This component has been renamed to avoid ambiguity with the **r** color component. You cannot mix and match selectors from the different notations:

```
vec3 myVector = {0.25, 0.5, 0.75};
float myR = myVector.r; // 0.25
vec2 myYZ = myVector.yz; // 0.5, 0.75
float myQ = myVector.q; // not allowed, accesses component beyond vec3
float myRY = myVector.ry; // not allowed, mixes two notations
```

You can use the component selectors to rearrange the order of components or replicate them:

```
vec3 myZYX = myVector.zyx; // reverse order
vec4 mySSTT = myVector.ssst; // replicate s and t twice each
```

You can also use them as writemasks on the left side of an assignment to select which components are modified. In this case, you cannot use component selectors more than once:

```
vec4 myColor = vec4(0.0, 1.0, 2.0, 3.0);
myColor.x = -1.0; // -1.0, 1.0, 2.0, 3.0
myColor.yz = vec2(-2.0, -3.0); // -1.0, -2.0, -3.0, 3.0
myColor.wx = vec2(0.0, 1.0); // 1.0, -2.0, -3.0, 0.0
myColor.zz = vec2(2.0, 3.0); // not allowed
```

Another way to get at individual vector components or matrix components is to use array subscript notation. This way, you can use an arbitrarily computed integer index to access your vector or matrix as if it were an array. Shader execution is undefined if an attempt is made to access a component outside the bounds of the vector or matrix:

```
float myY = myVector[1];
float myBug = myVector[-1]; // Don't try this!
```

For matrices, providing a single array index accesses the corresponding matrix column as a vector. Providing a second array index accesses the corresponding vector component:

```
mat3 myMatrix = mat3(1.0);
vec3 myFirstColumn = myMatrix[0]; // first column: 1.0, 0.0, 0.0
float element21 = myMatrix[2][1]; // last column, middle row: 0.0
```

Control Flow

Low-level shaders allow only a single stream of linear execution. GLSL introduces a variety of familiar nonlinear flow mechanisms that reduce code size, make more complex algorithms possible, and make shaders more readable.

Loops

For, **while**, and **do/while** loops are all supported with the same syntax as in C/C++. Loops can be nested. You can use **continue** and **break** to prematurely move on to the next iteration or break out of the loop:

```
for (l = 0; l < numLights; l++)
{
    if (!lightExists[l])
        continue;
    color += light[l];
}
```

```

while (true)
{
    if (lightNum < 0)
        break;
    color += light[lightNum];
    lightNum--;
}
do
{
    color += light[lightNum];
    lightNum--;
} while (lightNum > 0);

```

if/else

You can use `if` and `if/else` clauses to select between multiple blocks of code. These conditionals can also be nested:

```

color = unlitColor;
if (numLights > 0)
{
    color = litColor;
}
if (numLights > 0)
{
    color = litColor;
}
else
{
    color = unlitColor;
}

```

discard

Fragment shaders have a special control flow mechanism called `discard`. It terminates execution of the current fragment's shader. All subsequent per-fragment pipeline stages are skipped, and the fragment is not written to the framebuffer:

```

// e.g. perform an alpha test within your fragment shader
if (color.a < 0.9)
    discard;

```

Functions

Functions are used to modularize shader code. All shaders must define a `main` function, which is the place where execution begins. The `void` parameter list here is optional, but not the `void` return:

```

void main(void)
{
    ...
}

```

Functions must be either defined or declared with a prototype before use. These definitions or declarations should occur globally, outside any function. Return types are required, as are types for each function argument. Also, arguments can have an optional qualifier `in`, `out`, `inout`, or `const` (see [Table 21.2](#)):

```

// function declaration
bool isAnyComponentNegative(const vec4 v);
// function definition
bool isAnyComponentNegative(const vec4 v)
{
    if ((v.x < 0.0) || (v.y < 0.0) ||
        (v.z < 0.0) || (v.w < 0.0))
        return true;
    else
        return false;
}
// function use
void main()
{
    bool someNeg = isAnyComponentNegative(gl_MultiTexCoord0);
    ...
}

```

Structures are allowed as arguments and return types. Arrays are allowed only as arguments, in which case the declaration and definition would include the array name with size, whereas the function call would just use the array name without brackets or size:

```

vec4 sumMyVectors(int howManyToSum, vec4 v[10]);
void main()
{
    vec4 myColors[10];
    ...
    gl_FragColor = sumMyVectors(6, myColors);
}

```

You can give more than one function the same name, as long as the return type or argument types are different. This is called *function name overloading* and is useful if you want to perform the same type of operation on, for example, different sized vectors:

```

float multiplyAccumulate(float a, float b, float c)
{
    return (a * b) + c; // scalar definition
}
vec4 multiplyAccumulate(vec4 a, vec4 b, vec4 c)
{
    return (a * b) + c; // 4-vector definition
}

```

Recursive functions are not allowed. In other words, the same function cannot be present more than once in the current call stack. Some compilers may be able to catch this and throw an error, but in any case, shader execution will be undefined.

Approximately 50 built-in functions provide all sorts of useful calculations, ranging from simple arithmetic to trigonometry. You can consult the GLSL specification for the complete list and descriptions.

Texture Lookup Functions

Texture lookup built-in functions deserve special mention. Whereas some of the other built-in functions are provided as a convenience because you could code your own versions relatively easily, texture lookup built-in functions, listed in [Table 21.6](#), are crucial to perform even the most basic texturing.

Table 21.6. Texture Lookup Built-in Functions

Prototype

```

vec4 texture1D(sampler1D sampler, float coord [,
    ↪ float bias] )
vec4 texture1DProj(sampler1D sampler, vec2 coord [,
    ↪ , float bias] )
vec4 texture1DProj(sampler1D sampler, vec4 coord [,
    ↪ , float bias] )
vec4 texture1DLod(sampler1D sampler, float coord,
    ↪ float lod)
vec4 texture1DProjLod(sampler1D sampler, vec2
    ↪ coord, float lod)
vec4 texture1DProjLod(sampler1D sampler, vec4
    ↪ coord, float lod)
vec4 texture2D(sampler2D sampler, vec2 coord [,
    ↪ float bias] )
vec4 texture2DProj(sampler2D sampler, vec3 coord [,
    ↪ , float bias] )
vec4 texture2DProj(sampler2D sampler, vec4 coord [,
    ↪ , float bias] )
vec4 texture2DLod(sampler2D sampler, vec2 coord,
    ↪ float lod)
vec4 texture2DProjLod(sampler2D sampler, vec3
    ↪ coord, float lod)
vec4 texture2DProjLod(sampler2D sampler, vec4
    ↪ coord, float lod)
vec4 texture3D(sampler3D sampler, vec3 coord [,
    ↪ float bias] )
vec4 texture3DProj(sampler3D sampler, vec4 coord [,
    ↪ , float bias] )
vec4 texture3DLod(sampler3D sampler, vec3 coord,
    ↪ float lod)
vec4 texture3DProjLod(sampler3D sampler, vec4
    ↪ coord, float lod)
vec4 textureCube(samplerCube sampler, vec3 coord [,
    ↪ , float bias] )
vec4 textureCubeLod(samplerCube sampler, vec3
    ↪ coord, float lod)
vec4 shadow1D(sampler1DShadow sampler, vec3 coord
    ↪ [, float bias] )
vec4 shadow2D(sampler2DShadow sampler, vec3 coord
    ↪ [, float bias] )
vec4 shadow1DProj(sampler1DShadow sampler, vec4
    ↪ coord, [, float bias] )
vec4 shadow2DProj(sampler2DShadow sampler, vec4
    ↪ coord, [, float bias] )
vec4 shadow1DLod(sampler1DShadow sampler, vec3
    ↪ coord, float lod)
vec4 shadow2DLod(sampler2DShadow sampler, vec3
    ↪ coord, float lod)
vec4 shadow1DProjLod(sampler1DShadow sampler, vec4
    ↪ coord, float lod)
vec4 shadow2DProjLod(sampler2DShadow sampler, vec4
    ↪ coord, float lod)

```

The lookup is performed on the texture of the type encoded in the function name (1D, 2D, 3D, Cube) currently bound to the sampler represented by the sampler parameter. The "Proj" versions

perform a projective divide on the texture coordinate before lookup. The divisor is the last component of the coordinate vector.

The "Lod" versions, available only in a vertex shader, specify the mipmap level-of-detail (LOD) from which to sample. The non-"Lod" versions sample from the base LOD when used by a vertex shader. Fragment shaders can use only the non-"Lod" versions, where the mipmap LOD is computed as usual based on texture coordinate derivatives. However, fragment shaders can supply an optional bias that will be added to the computed LOD. This bias parameter is not allowed in a vertex shader.

The "shadow" versions perform a depth texture comparison as part of the lookup (see [Chapter 18](#), "Depth Textures and Shadows").

Summary

In this chapter, you learned all the nuts and bolts of the OpenGL Shader Language (GLSL). We discussed all the variable types, operators, and flow control mechanisms. We also described how to use the entrypoints for loading and compiling shader objects and linking and using program objects. There was a lot of ground to cover here, but we made it through at a record pace.

This chapter concludes the boring lecture portion of our shader coverage. You now have a solid conceptual foundation for the remaining two chapters, which will provide practical examples of vertex and fragment shader applications using both low-level and high-level shader languages. The following chapters will prove much more enjoyable with all the textbook learning behind you.

Reference

glAttachObjectARB

Purpose: Attaches an object to another container object.

Include File: `<glext.h>`

Syntax:

```
void glAttachObjectARB(GLhandleARB containerObj,
    GLhandleARB obj);
```

Description: This function attaches an object to a container object. If the first argument is not a container object, if the second argument is already attached to the specified container object, or if the second object is not a type that can be attached to a container, an error is thrown. `glLinkProgramARB` must be called after `glAttachObjectARB` for the newly-attached objects to take effect.

Parameters:

`containerObj` `GLhandleARB`: The container object to which the other object is being attached.

`obj` `GLhandleARB`: The object being attached to the container object.

Returns: None.

See Also: `glDetachObjectARB`, `glLinkProgramARB`, `glGetAttachedObjectsARB`

glBindAttribLocationARB**Purpose:** Sets the location for a generic vertex attribute.**Include File:** `<glext.h>`**Syntax:**

```
void glBindAttribLocationARB(GLhandleARB
    ↪ programObj, GLuint index,
                           const GLcharARB *name);
```

Description: This function explicitly specifies the generic vertex attribute number for use by the specified attribute variable. Any attributes not explicitly bound will be bound automatically during linking. This routine can be called at any time, even before a vertex shader is attached to a program object, to reserve generic attribute locations. Also, `glLinkProgramARB` must be called again after `glBindAttribLocationARB` for the new location(s) to take effect.

Parameters:**programObj** `GLhandleARB`: The program object containing the vertex attribute.**index** `GLuint`: The number of the vertex attribute where this attribute will be located.**name** `const GLcharARB *`: The name of the attribute variable.**Returns:** None.**See Also:** `glGetAttribLocationARB`, `glGetActiveAttribARB`, `glLinkProgramARB`, `glVertexAttrib*ARB`, `glVertexAttribPointerARB`**glCompileShaderARB****Purpose:** Compiles a shader.**Include File:** `<glext.h>`**Syntax:**

```
void glCompileShaderARB(GLhandleARB shaderObj);
```

Description: This function attempts to compile the shader text previously loaded into the shader object. The shader object flag `GL_OBJECT_COMPILE_STATUS_ARB` is set to `GL_TRUE` if the compile is successful and ready for linkage; otherwise, it's set to `GL_FALSE`. The shader object's info log may be updated to include information about the compile.

Parameters:**shaderObj** `GLhandleARB`: The shader object to compile.**Returns:** None.**See Also:** `glGetInfoLogARB`, `glShaderSourceARB`, `glGetObjectParameter*vARB`, `glLinkProgramARB`

glCreateProgramObjectARB**Purpose:** Creates a program object.**Include File:** `<glext.h>`**Syntax:**`GLhandleARB glCreateProgramObjectARB(GLvoid);`**Description:** This function creates a new program object and returns its handle.**Parameters:** None.**Returns:** `GLhandleARB`: The handle to the new program object.**See Also:** `glCreateShaderObjectARB`, `glDeleteObjectARB`, `glUseProgramObjectARB`, `glLinkProgramARB`**glCreateShaderObjectARB****Purpose:** Creates a shader object of the requested type.**Include File:** `<glext.h>`**Syntax:**`GLhandleARB glCreateShaderObjectARB(GLenum
➥ shaderType);`**Description:** This function creates a new shader object of the specified type and returns its handle.**Parameters:****shaderType** `GLenum`: The type of shader object being created. It can be one of the following constants: `GL_VERTEX_SHADER_ARB`: Create a vertex shader object. `GL_FRAGMENT_SHADER_ARB`: Create a fragment shader object.**Returns:** `GLhandleARB`: The handle to the new shader object.**See Also:** `glCreateProgramObjectARB`, `glDeleteObjectARB`, `glCompileShaderARB`**glDeleteObjectARB****Purpose:** Deletes an object.**Include File:** `<glext.h>`

Syntax:

```
void glDeleteObjectARB(GLhandleARB obj);
```

Description: If the specified object is not attached to any container object and is not a part of the current rendering state of any context, the object is deleted immediately. Otherwise, it is flagged for future deletion and won't be deleted until it's no longer attached to any container object and no longer part of the current rendering state of any context. When a container object is deleted, all objects attached to it become detached.

Parameters:

obj **GLhandleARB**: The object to delete.

Returns: None.

See Also: `glCreateProgramObject`, `glCreateShaderObjectARB`,
`glGetObjectParameter*vARB`, `glUseProgramObjectARB`

glDetachObjectARB

Purpose: Detaches an object from a container object.

Include File: `<glext.h>`

Syntax:

```
void glDetachObjectARB(GLhandleARB containerObj,  
                      GLhandleARB attachedObj);
```

Description: This function detaches an object from a container object. If the detached object is flagged for deletion and not attached to any other container objects, it is deleted. `glLinkProgramARB` must be called after `glDetachObjectARB` for the newly-detached objects to take effect.

Parameters:

containerObj **GLhandleARB**: The container object from which the other object is to be detached.

attachedObj **GLhandleARB**: The object being detached.

Returns: None.

See Also: `glAttachObjectARB`, `glLinkProgramARB`, `glGetAttachedObjectsARB`

glGetActiveAttribARB

Purpose: Gets information about active vertex attributes.

Include File: `<glext.h>`

Syntax:

```
void glGetActiveAttribARB(GLhandleARB programObj,
  ➔ GLuint index,
  ➔ GLsizei maxLength,
  ➔ GLsizei *length, GLint *size,
  ➔ GLenum *type, GLcharARB
  ➔ *name);
```

Description: After a program object link has been attempted, this function can be called to find information about a particular vertex attribute variable, including its name, the length of its name, its size, and its type.

Parameters:

programObj `GLhandleARB`: The program object being queried.

index `GLuint`: The vertex attribute being queried.

maxLength `GLsizei`: The maximum number of characters that should be returned for the name.

length `GLsizei *`: A pointer to the location where the name length is returned. The length includes only characters actually returned and does not include the null terminator. If a `NULL` pointer is provided, no length is returned.

size `GLint *`: A pointer to the location where the attribute size is returned. This size, measured in units of the returned type, is currently always 1 but is part of the API in the event that attribute arrays become part of the OpenGL Shading Language at a later date.

type `GLenum *`: A pointer to the location where the attribute type is returned. The type will be one of the following constants:

- `GL_FLOAT`: Scalar floating-point value.
- `GL_FLOAT_VEC2_ARB`: Two-component floating-point vector.
- `GL_FLOAT_VEC3_ARB`: Three-component floating-point vector.
- `GL_FLOAT_VEC4_ARB`: Four-component floating-point vector.
- `GL_FLOAT_MAT2_ARB`: 2x2 floating-point matrix.
- `GL_FLOAT_MAT3_ARB`: 3x3 floating-point matrix.
- `GL_FLOAT_MAT4_ARB`: 4x4 floating-point matrix.

name `GLcharARB *`: A pointer to the location where the attribute name is returned as a null-terminated string.

Returns: None.

See Also: `glGetAttribLocationARB`, `glBindAttribLocationARB`,
`glGetVertexAttrib*vARB`, `glGetVertexAttribPointervARB`

glGetActiveUniformARB**Purpose:** Gets information about active uniforms.**Include File:** `<glext.h>`**Syntax:**

```
void glGetActiveUniformARB(GLhandleARB programObj,
  ➔ GLuint index,
  ➔ GLsizei maxLength,
  ➔ GLsizei *length, GLint *size,
  ➔ GLenum *type, GLcharARB
  ➔ *name);
```

Description: After a program object link has been attempted, this function can be called to find information about a particular uniform variable, including its name, the length of its name, its size, and its type.**Parameters:**

programObj `GLhandleARB`: The program object being queried.

index `GLuint`: The uniform being queried.

maxLength `GLsizei`: The maximum number of characters that should be returned for the name.

length `GLsizei *`: A pointer to the location where the name length is returned. The length includes only characters actually returned and does not include the null terminator. If a `NULL` pointer is provided, no length is returned.

size `GLint *`: A pointer to the location where the uniform array size is returned. This size is in terms of the returned type and is 1 for nonarrays.

type `GLenum *`: A pointer to the location where the uniform type is returned. The type will be one of the following constants:

- `GL_FLOAT`: Scalar floating-point value.
- `GL_FLOAT_VEC2_ARB`: Two-component floating-point vector.
- `GL_FLOAT_VEC3_ARB`: Three-component floating-point vector.
- `GL_FLOAT_VEC4_ARB`: Four-component floating-point vector.
- `GL_INT`: Scalar integer value.
- `GL_INT_VEC2_ARB`: Two-component integer vector.
- `GL_INT_VEC3_ARB`: Three-component integer vector.
- `GL_INT_VEC4_ARB`: Four-component integer vector.
- `GL_BOOL_ARB`: Scalar Boolean value.

`GL_BOOL_VEC2_ARB`: Two-component Boolean vector.

`GL_BOOL_VEC3_ARB`: Three-component Boolean vector.

`GL_BOOL_VEC4_ARB`: Four-component Boolean vector.

`GL_FLOAT_MAT2_ARB`: 2x2 floating-point matrix.

`GL_FLOAT_MAT3_ARB`: 3x3 floating-point matrix.

`GL_FLOAT_MAT4_ARB`: 4x4 floating-point matrix.

`GL_SAMPLER_1D_ARB`: Handle for accessing 1D texture.

`GL_SAMPLER_2D_ARB`: Handle for accessing 2D texture.

`GL_SAMPLER_3D_ARB`: Handle for accessing 3D texture.

`GL_SAMPLER_CUBE_ARB`: Handle for accessing cube-mapped texture.

`GL_SAMPLER_1D_SHADOW_ARB`: Handle for accessing 1D depth texture with depth comparison.

`GL_SAMPLER_2D_SHADOW_ARB`: Handle for accessing 2D depth texture with depth comparison.

name `GLcharARB *`: A pointer to the location where the uniform name is returned as a null-terminated string.

Returns: None.

See Also: `glGetUniformLocationARB`, `glGetUniform*vARB`, `glUniform*ARB`

glGetAttachedObjectsARB

Purpose: Gets a list of objects attached to a container object.

Include File: `<glext.h>`

Syntax:

```
void glGetAttachedObjectsARB(GLhandleARB
  ↪ containerObj, GLsizei maxCount,
                           GLsizei *count,
  ↪ GLhandleARB *obj);
```

Description: This function returns a list of all the objects contained within another object. Specifically, it returns all the shader objects contained within a program object.

Parameters:

`containerObj` `GLhandleARB`: The container object being queried.

`maxCount` `GLsizei`: The maximum number of object handles to return.

count `GLsizei *`: A pointer to the location where the count is returned. The count includes only handles actually returned. If a `NULL` pointer is provided, no count is returned.

obj `GLhandleARB *`: A pointer to an array of returned object handles.

Returns: None.

See Also: `glAttachObjectARB`, `glDetachObjectARB`, `glDeleteObjectARB`

glGetAttribLocationARB

Purpose: Gets the location of a vertex attribute variable.

Include File: `<glext.h>`

Syntax:

```
GLint glGetAttribLocationARB(GLhandleARB
    ↗ programObj, const GLcharARB *name);
```

Description: After a program object has been successfully linked, this function returns the location of the vertex attribute variable with the specified name.

Parameters:

programObj `GLhandleARB`: The program object being queried.

name `const GLcharARB *`: The vertex attribute variable name, null terminated.

Returns: `GLint`: The location of the vertex attribute or `-1` if no such attribute is active or if the name starts with the reserved prefix `"gl_"`. If *name* corresponds to an active matrix attribute, the location of the first column is returned.

See Also: `glGetActiveAttribARB`, `glBindAttribLocationARB`, `glLinkProgramARB`, `glVertexAttrib*ARB`, `glVertexAttribPointerARB`

glGetHandleARB

Purpose: Returns an object handle.

Include File: `<glext.h>`

Syntax:

```
GLhandleARB glGetHandleARB(GLenum pname);
```

Description: This function returns the handle to an object that is part of the current state.

Parameters:

pname `GLenum`: The object handle from current state to return. It must be the constant `GL_PROGRAM_OBJECT_ARB`.

Returns: `GLhandleARB`: The current program object or `0` if no such object is currently used.

See Also: [glCreateProgramObjectARB](#), [glUseProgramObjectARB](#)

glGetInfoLogARB

Purpose: Gets information about the last compile or link.

Include File: `<glext.h>`

Syntax:

```
void glGetInfoLogARB(GLhandleARB obj, GLsizei
➥ maxLength, GLsizei *length,
➥ GLcharARB *infoLog);
```

Description: Each shader object has an info log that contains information about the most recent compile attempt by that shader object. Similarly, each program object has an info log that contains information about the most recent link or validation attempt. This function provides access to these logs.

Parameters:

obj `GLhandleARB`: The object whose information log is being queried.

maxLength `GLsizei`: The maximum number of characters that should be returned in the info log.

length `GLsizei *`: A pointer to the location where the length of the info log is returned. The length includes only characters actually returned and does not include the null terminator. If a `NULL` pointer is provided, no length is returned.

infoLog `GLcharARB *`: A pointer to the location where the info log text is returned as a null-terminated string.

Returns: None.

See Also: [glCompileShaderARB](#), [glLinkProgramARB](#), [glValidateProgramARB](#)

glGetObjectParameterfvARB

Purpose: Gets information about the specified object.

Include File: `<glext.h>`

Syntax:

```
void glGetObjectParameterfvARB(GLhandleARB obj,
➥ GLenum pname, GLfloat *params);
void glGetObjectParameterivARB(GLhandleARB obj,
➥ GLenum pname, GLint *params);
```

Description: This function queries a property of the specified program or shader object.

Parameters:

<i>obj</i>	<code>GLhandleARB</code> : The object being queried.
<i>pname</i>	<code>GLenum</code> : The property being queried. It can be one of the following constants:
	<code>GL_OBJECT_TYPE_ARB</code> : Returns <code>GL_PROGRAM_OBJECT_ARB</code> if the object is a program object and <code>GL_SHADER_OBJECT_ARB</code> if the object is a shader object.
	<code>GL_OBJECT_SUBTYPE_ARB</code> : Returns either <code>GL_VERTEX_SHADER_ARB</code> or <code>GL_FRAGMENT_SHADER_ARB</code> if the object is a shader object.
	<code>GL_OBJECT_DELETE_STATUS_ARB</code> : Returns 1 or 1.0 if the object's delete flag is set; otherwise, 0 or 0.0 is returned.
	<code>GL_OBJECT_COMPILE_STATUS_ARB</code> : Returns 1 or 1.0 if the shader object's most recent compile was successful.
	<code>GL_OBJECT_LINK_STATUS_ARB</code> : Returns 1 or 1.0 if the program object's most recent link was successful.
	<code>GL_OBJECT_VALIDATE_STATUS_ARB</code> : Returns 1 or 1.0 if the program object's most recent validation was successful.
	<code>GL_OBJECT_INFO_LOG_LENGTH_ARB</code> : Returns the length in characters of the object's info log, including the null terminator, or 0 if there is no info log.
	<code>GL_OBJECT_ATTACHED_OBJECTS_ARB</code> : Returns the number of shader objects attached to this program object.
	<code>GL_OBJECT_ACTIVE_UNIFORMS_ARB</code> : Returns the number of uniforms active in this program object.
	<code>GL_OBJECT_ACTIVE_UNIFORM_MAX_LENGTH_ARB</code> : Returns the length of this program object's longest active uniform variable name.
	<code>GL_OBJECT_SHADER_SOURCE_LENGTH_ARB</code> : Returns the length of this shader object's shader text, including the null terminator.
	<code>GL_OBJECT_ACTIVE_ATTRIBUTES_ARB</code> : Returns the number of vertex attributes active in this program object.
	<code>GL_OBJECT_ACTIVE_ATTRIBUTE_MAX_LENGTH_ARB</code> : Returns the length of this program object's longest active vertex attribute variable name.
<i>params</i>	<code>GLfloat */ GLint */</code> : A pointer to the location where the result will be stored.
Returns:	None.
See Also:	<code>glShaderSourceARB</code> , <code>glCompileShaderARB</code> , <code>glLinkProgramARB</code> , <code>glAttachObjectARB</code> , <code>glDetachObjectARB</code> , <code>glCreateShaderObjectARB</code> , <code>glCreateProgramObjectARB</code> , <code>glDeleteObjectARB</code>

glGetShaderSourceARB

Purpose: Gets a shader object's source text.

Include File: `<glext.h>`

Syntax:

```
void glGetShaderSourceARB(GLhandleARB obj, GLsizei
➥ maxLength, GLsizei *length,
                           GLcharARB *source);
```

Description: This function returns a concatenation of all the shader source text previously supplied via `glShaderSourceARB`.

Parameters:

`obj` `GLhandleARB`: The shader object being queried.

`maxLength` `GLsizei`: The maximum number of characters that should be returned.

`length` `GLsizei *: A pointer to the location where the length of the source text is returned. The length includes only characters actually returned and does not include the null terminator. If a NULL pointer is provided, no length is returned.`

`source` `GLcharARB *: A pointer to the location where the shader source text is returned as a null-terminated string.`

Returns: None.

See Also: `glShaderSourceARB`, `glCreateShaderObjectARB`, `glCompileShaderARB`

glGetUniformfvARB

Purpose: Gets the value(s) of a uniform variable.

Include File: `<glext.h>`

Syntax:

```
void glGetUniformfvARB(GLhandleARB programObj,
➥ GLint location,
                           GLfloat *params);
void glGetUniformivARB(GLhandleARB programObj,
➥ GLint location, GLint *params);
```

Description: If the program object has been successfully linked, and a valid uniform location has been provided, as by `glGetUniformLocationARB`, this function returns the value or values of the uniform variable at that location. The number of values returned is based on the type of the uniform. Each element of an array variable must be queried independently. If the variable is a matrix, its data is returned in column-major order.

Parameters:

`programObj` `GLhandleARB`: The program object being queried.

`location` `GLint`: The location of the uniform variable being queried.

params `GLfloat */ GLint *`: A pointer to the location where the uniform value(s) will be stored.

Returns: None.

See Also: `glUniform*ARB`, `glGetUniformLocationARB`, `glGetActiveUniformARB`

glGetUniformLocationARB

Purpose: Gets the location of a uniform variable.

Include File: `<glext.h>`

Syntax:

```
GLint glGetUniformLocationARB(GLhandleARB
  ↪ programObj, const GLcharARB *name);
```

Description: After a program object has been successfully linked, this function returns the location of the uniform variable with the specified name.

Parameters:

`programObj` `GLhandleARB`: The program object being queried.

`name` `const GLcharARB *`: The uniform variable name, null terminated. The name cannot be a structure, array of structures, or a subcomponent of a vector or matrix. The `.` and `[]` operators can be used to identify members of a structure or array. The first element of an array can be queried either using just the name of the array, or with `[0]` appended.

Returns: `GLint`: The location of the uniform or `-1` if no such uniform is active or if the name starts with the reserved prefix `"gl_"`.

See Also: `glGetActiveUniformARB`, `glLinkProgramARB`, `glUniform*ARB`, `glGetUniform*vARB`

glLinkProgramARB

Purpose: Links a program object.

Include File: `<glext.h>`

Syntax:

```
void glLinkProgramARB(GLhandleARB programObj);
```

Description: This function attempts to link the previously compiled shader objects contained within the specified program object. The program object flag `GL_OBJECT_LINK_STATUS_ARB` is set to `GL_TRUE` if the link is successful and ready for use; otherwise, it's set to `GL_FALSE`. The program object's info log will be updated to include information about the link.

Parameters:

programObj GLhandleARB: The program object to link.

Returns: None.

See Also: `glGetInfoLogARB`, `glGetObjectParameter*vARB`, `glCompileShaderARB`, `glUseProgramObjectARB`

glShaderSourceARB

Purpose: Loads source text into a shader object.

Include File: `<glext.h>`

Syntax:

```
void glShaderSourceARB(GLhandleARB shaderObj,
    ↪ GLsizei count,
                const GLcharARB **string,
    ↪ const GLint *length);
```

Description: This function loads a shader object with one or more strings of shader source text. Any pre-existing source text is replaced with the new text. For the new shader source to take effect, the shader object must be freshly compiled, and the program object(s) it is attached to must be freshly linked.

Parameters:

shaderObj GLhandleARB: The shader object whose source text is being loaded.

count GLsizei: The number of strings to load.

string const GLcharARB **: A pointer to an array of one or more strings, each representing part of the shader source text.

length const GLint *: A pointer to an array of lengths, representing the length of each shader source string, excluding the null terminator. If a length is -1 , the corresponding string is guaranteed to be null-terminated. If length is `NULL`, all strings are guaranteed to be null-terminated.

Returns: None.

See Also: `glCompileShaderARB`, `glGetShaderSourceARB`, `glCreateShaderObjectARB`

glUniform*ARB

Purpose: Load uniform variable values.

Include File: `<glext.h>`

Syntax:

```

void glUniform1fARB(GLint location, GLfloat v0);
void glUniform2fARB(GLint location, GLfloat v0,
    GLfloat v1);
void glUniform3fARB(GLint location, GLfloat v0,
    GLfloat v1, GLfloat v2);
void glUniform4fARB(GLint location, GLfloat v0,
    GLfloat v1,
                           GLfloat v2,
    GLfloat v3);
void glUniform1iARB(GLint location, GLint v0);
void glUniform2iARB(GLint location, GLint v0,
    GLint v1);
void glUniform3iARB(GLint location, GLint v0,
    GLint v1, GLint v2);
void glUniform4iARB(GLint location, GLint v0,
    GLint v1, GLint v2, GLint v3);
void glUniform1fvARB(GLint location, GLsizei count
    , const GLfloat *value);
void glUniform2fvARB(GLint location, GLsizei count
    , const GLfloat *value);
void glUniform3fvARB(GLint location, GLsizei count
    , const GLfloat *value);
void glUniform4fvARB(GLint location, GLsizei count
    , const GLfloat *value);
void glUniform1ivARB(GLint location, GLsizei count
    , const GLint *value);
void glUniform2ivARB(GLint location, GLsizei count
    , const GLint *value);
void glUniform3ivARB(GLint location, GLsizei count
    , const GLint *value);
void glUniform4ivARB(GLint location, GLsizei count
    , const GLint *value);
void glUniformMatrix2fvARB(GLint location, GLsizei
    count, GLboolean transpose, const GLfloat *value);
void glUniformMatrix3fvARB(GLint location, GLsizei
    count, GLboolean transpose, const GLfloat *value);
void glUniformMatrix4fvARB(GLint location, GLsizei
    count, GLboolean transpose, const GLfloat *value);

```

Description: This function loads one or more values into the specified uniform variable of the currently used program object. The size and type of the function must match the size and type of the uniform variable, except for Boolean type uniforms. For Boolean uniforms, either the floating-point or integer functions can be used, where 0.0 or 0 is converted to `GL_FALSE` and all other values are converted to `GL_TRUE`. If no program object is in use, this function will fail and generate an error.

Parameters:

`location` `GLint`: The location of the uniform variable to load, as returned from `glGetUniformLocationARB`. If an array, this is the starting location for loading array elements.

`count` `GLsizei`: The number of array elements to load or 1 if not an array.

transpose `GLboolean`: Whether or not to transpose the specified matrix elements before storing them in OpenGL state. If `GL_TRUE`, the matrix is interpreted as originating in row-major order, and is transposed to column-major order before storing. If `GL_FALSE`, the matrix is interpreted as being in column-major order already, which is the order in which GLSL operates on matrices.

v0, v1, v2, v3 `GLfloat / GLint`: The values to load into the uniform variable.

value `GLfloat */ GLint *`: A pointer to the values to load into the uniform variable.

Returns: None.

See Also: `glGetUniform*vARB`, `glGetUniformLocationARB`, `glGetActiveUniformARB`, `glLinkProgramARB`

glUseProgramObjectARB

Purpose: Sets the current program object.

Include File: `<glext.h>`

Syntax:

```
void glUseProgramObjectARB(GLhandleARB programObj);
```

Description: After a program object has been successfully linked, this function installs its executable code as part of the current rendering state. It can also be used to deinstall all executable code and return to a fully fixed functionality pipeline. Once installed, executable code can be changed only by first relinking, at which point calling this function again is unnecessary.

Parameters:

programObj `GLhandleARB`: The program object to be used or 0 to revert to fixed functionality.

Returns: None.

See Also: `glLinkProgramARB`, `glGetHandleARB`, `glUniform*ARB`

glValidateProgramARB

Purpose: Validates a program object against current state.

Include File: `<glext.h>`

Syntax:

```
void glValidateProgramARB(GLhandleARB programObj);
```

Description: Without taking the current state into consideration, it is impossible to know whether a given program object will execute when first used for rendering, even if it linked successfully. This function validates the specified program object against the current OpenGL state to detect any problems or inefficiencies. The program object flag `GL_OBJECT_VALIDATE_STATUS_ARB` is set to `GL_TRUE` if the validation is successful and the program object is guaranteed to work with the current state; otherwise, it's set to `GL_FALSE`. The program object's info log will be updated to include information about the validation.

Parameters:

`programObj` `GLhandleARB`: The program object to validate.

Returns: None.

See Also: `glGetInfoLogARB`, `glGetObjectParameter*vARB`, `glLinkProgramARB`, `glUseProgramObjectARB`

Chapter 22. Vertex Shading: Do-It-Yourself Transform, Lighting, and Texgen

by Benjamin Lipchak

WHAT YOU'LL LEARN IN THIS CHAPTER:

- How to perform per-vertex lighting
- How to generate texture coordinates
- How to calculate per-vertex fog
- How to calculate per-vertex point size
- How to squash and stretch objects
- How to make realistic skin with vertex blending

This chapter is devoted to the application of vertex shaders. We covered the basic mechanics of high-level and low-level vertex shaders in the preceding two chapters, but at some point you have to put the textbook down and start learning by doing. Here, we introduce a handful of shaders that perform various real-world tasks. You are encouraged to use these shaders as a starting point for your own experimentation.

Getting Your Feet Wet

Every shader should at the very least output a clip-space position coordinate. Lighting and texture coordinate generation (texgen), the other operations typically performed in vertex shaders, may not be necessary. For example, if you're creating a depth texture and all you care about are the final depth values, you wouldn't waste instructions in your shader to output a color or texture coordinates. But one way or another, you always need to output a clip-space position for subsequent primitive assembly and rasterization to occur.

For your first sample shader, you'll perform the bare-bones vertex transformation that would occur automatically by fixed functionality if you weren't using a vertex shader. As an added bonus, you'll copy the incoming color into the outgoing color. Remember, anything that isn't output remains undefined. If you want that color to be available later in the pipeline, you have to copy it from input to output, even if the vertex shader doesn't need to change it in any way.

For each example shader, we'll provide both a high-level and a low-level version that perform equivalent operations. This way, you can learn both shader languages by comparison. Also, if only one of the two extensions is available on your OpenGL implementation, you won't miss out completely. [Figure 22.1](#) shows the result of the simple shaders in [Listings 22.1](#) and [22.2](#).

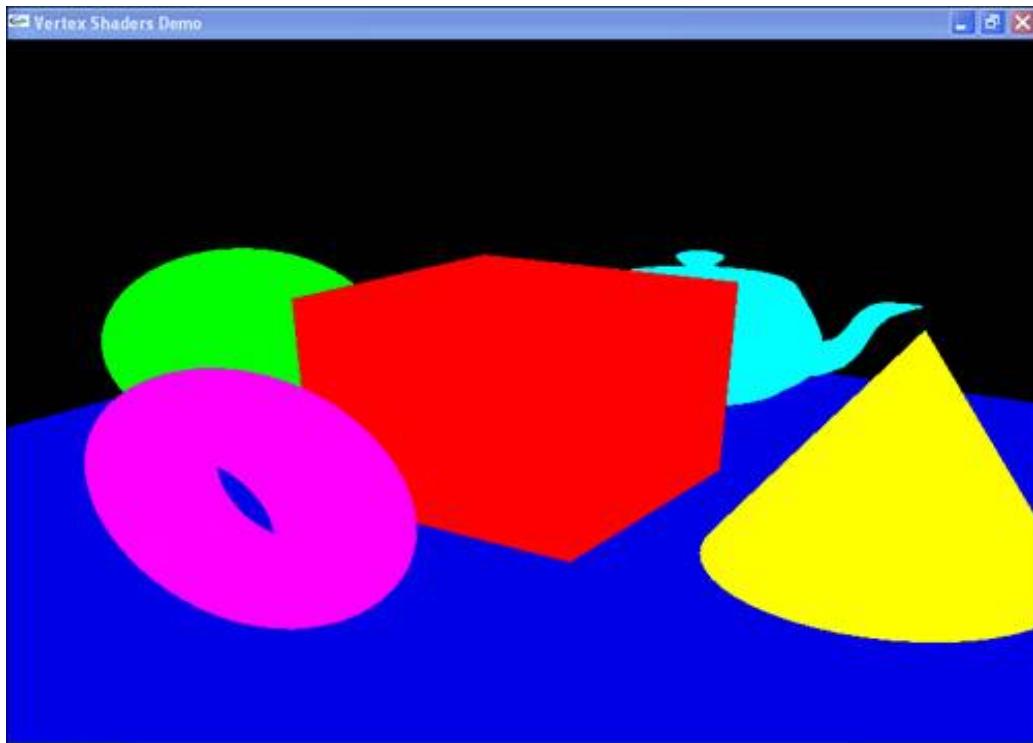
Listing 22.1. Simple High-Level Vertex Shader

```

// simple.vs
//
// Generic vertex transformation,
// copy primary color
void main(void)
{
    // multiply object-space position by MVP matrix
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    // Copy the primary color
    gl_FrontColor = gl_Color;
}

```

Figure 22.1. This vertex shader transforms the position to clip space and copies the vertex's color from input to output.



Listing 22.2. Simple Low-Level Vertex Shader

```

!!ARBvp1.0
# simple.vp
#
# Generic vertex transformation,
# copy primary color
ATTRIB iPos = vertex.position;      # input position
ATTRIB iPrC = vertex.color.primary; # input primary color
OUTPUT oPos = result.position;      # output position
OUTPUT oPrC = result.color.primary; # output primary color
PARAM mvp[4] = { state.matrix.mvp }; # modelview * projection matrix
TEMP clip;                         # temporary register
DP4 clip.x, iPos, mvp[0];          # multiply input position by MVP
DP4 clip.y, iPos, mvp[1];
DP4 clip.z, iPos, mvp[2];
DP4 clip.w, iPos, mvp[3];
MOV oPos, clip;                    # output clip-space coord
MOV oPrC, iPrC;                   # copy primary color in to out

```

END

Notice how much more compact and readable the high-level GLSL shader is versus the low-level `GL_ARB_vertex_program` shader. When explaining the shaders, we'll make reference to the GLSL version. You can compare the two to see how the high-level expressions are decomposed into their low-level counterparts. For example, the multiplication of the object-space vertex position (`gl_Vertex`) by the concatenation of the modelview and projection matrices (`gl_ModelViewProjection`) to get the clip-space vertex position (`gl_Position`) is implemented in the low-level shader by a series of four dot product (`DP4`) instructions.

Diffuse Lighting

Diffuse lighting takes into account the orientation of a surface relative to the direction of incoming light. The following is the equation for diffuse lighting:

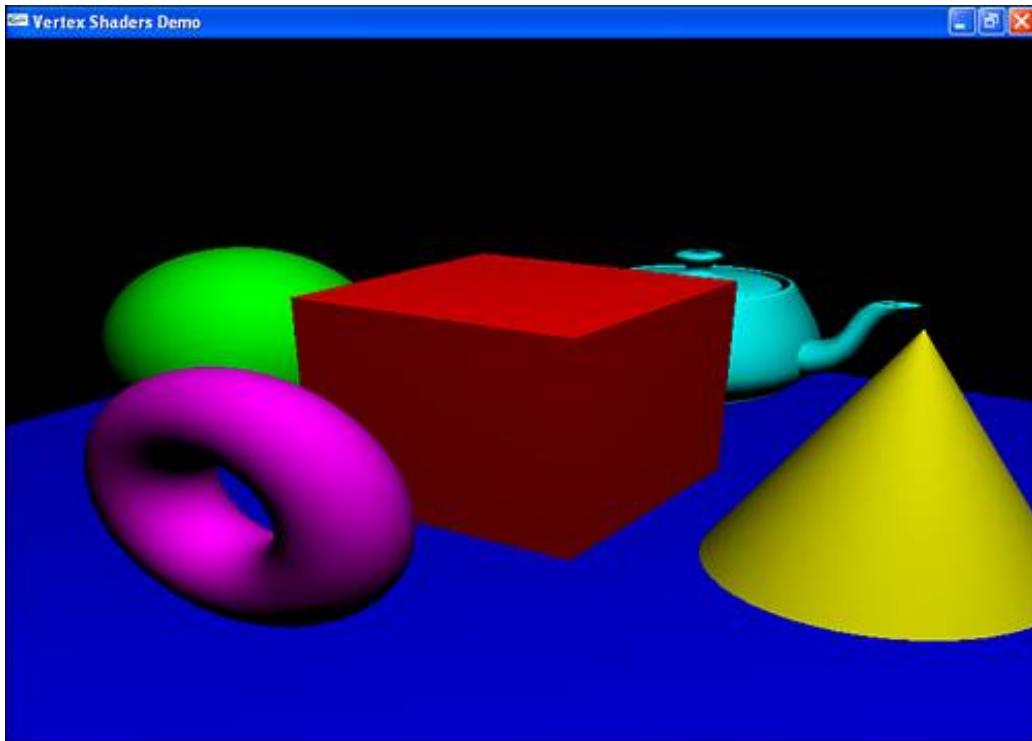
$$C_{\text{diff}} = \max\{N \cdot L, 0\} * C_{\text{mat}} * C_{\text{li}}$$

N is the vertex's unit normal, and L is the unit vector representing the direction from the vertex to the light source. C_{mat} is the color of the surface material, and C_{li} is the color of the light. C_{diff} is the resulting diffuse color. Because the light in the example is white, you can omit that term, as it would be the same as multiplying by `{ 1,1,1,1 }`. [Figure 22.2](#) shows the result from [Listings 22.3](#) and [22.4](#), which implement the diffuse lighting equation.

Listing 22.3. Diffuse Lighting High-Level Vertex Shader

```
// diffuse.vs
//
// Generic vertex transformation,
// diffuse lighting based on one
// white light
uniform vec3 lightPos0;
void main(void)
{
    // normal MVP transform
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    vec3 N = normalize(gl_NormalMatrix * gl_Normal);
    vec4 V = gl_ModelViewMatrix * gl_Vertex;
    vec3 L = normalize(lightPos0 - V.xyz);
    // output the diffuse color
    float NdotL = dot(N, L);
    gl_FrontColor = gl_Color * vec4(max(0.0, NdotL));
}
```

Figure 22.2. This vertex shader computes diffuse lighting.



Listing 22.4. Diffuse Lighting Low-Level Vertex Shader

```
!!ARBvp1.0
# diffuse.vp
#
# Generic vertex transformation,
# diffuse lighting based on one
# white light
ATTRIB iPos = vertex.position;           # input position
ATTRIB iPrC = vertex.color.primary;      # input primary color
ATTRIB iNrm = vertex.normal;             # input normal
OUTPUT oPos = result.position;           # output position
OUTPUT oPrC = result.color.primary;      # output primary color
PARAM mvp[4] = { state.matrix.mvp };      # modelview * proj matrix
PARAM mv[4] = { state.matrix.modelview }; # modelview matrix
# inverse transpose of modelview matrix:
PARAM mvIT[4] = { state.matrix.modelview.invtrans };
PARAM lightPos = program.local[0];         # light pos in eye space
TEMP N, V, L, NdotL;                      # temporary registers
DP4 oPos.x, iPos, mvp[0];                 # xform input pos by MVP
DP4 oPos.y, iPos, mvp[1];
DP4 oPos.z, iPos, mvp[2];
DP4 oPos.w, iPos, mvp[3];
DP4 V.x, iPos, mv[0];                     # xform input pos by MV
DP4 V.y, iPos, mv[1];
DP4 V.z, iPos, mv[2];
DP4 V.w, iPos, mv[3];
SUB L, lightPos, V;                       # vertex to light vector
DP3 N.x, iNrm, mvIT[0];                  # xform norm to eye space
DP3 N.y, iNrm, mvIT[1];
DP3 N.z, iNrm, mvIT[2];
DP3 N.w, N, N;                           # normalize normal
RSQ N.w, N.w;
MUL N, N, N.w;
DP3 L.w, L, L;                           # normalize light vector
RSQ L.w, L.w;
MUL L, L, L.w;
```

```

DP3 NdotL, N, L;                      # N . L
MAX NdotL, NdotL, 0.0;
MUL oPrC, iPrC, NdotL;                 # diffuse color
END

```

After computing the clip-space position as you did in the "simple" shader, the "diffuse" shader proceeds to transform the vertex position to eye space, too. All the lighting calculations are performed in eye space, so you need to transform the normal vector from object space to eye space as well. GLSL provides the `gl_NormalMatrix` built-in uniform matrix as a convenience for this purpose. It is simply the inverse transpose of the modelview matrix's upper-left 3x3 elements. The last vector you need to compute is the light vector, which is the direction from the vertex position to the light position, so you just subtract one from the other.

Both the normal and the light vectors must be unit vectors, so you normalize them before continuing. GLSL supplies a built-in function to perform this common task, but in the low-level shader, you have to do the normalization manually. Turning an arbitrary vector into a unit vector is reasonably simple. You have to scale the vector components by the inverse of the vector's length. A dot product operation (`DP3`) of the vector against itself returns the vector's length squared. A reciprocal square root operation (`RSQ`) turns that length squared into the inverse length scale factor. Then you just multiply (`MUL`) that scale factor by the original vector components to yield the unit vector.

The dot product of the two unit vectors, N and L , will be in the range $[-1, 1]$. But because you're interested in the amount of diffuse lighting bouncing off the surface, having a negative contribution doesn't make sense. This is why you clamp the result of the dot product to the range $[0, 1]$ by using the `max` (GLSL) or `MAX` (low-level) operations. The diffuse lighting contribution can then be multiplied by the vertex's diffuse material color to obtain the final lit color.

Specular Lighting

Specular lighting also takes into account the orientation of a surface relative to the direction of incoming light. The following is the equation for specular lighting:

$$C_{\text{spec}} = \max\{N \cdot H, 0\}^S_{\text{exp}} * C_{\text{mat}} * C_{\text{li}}$$

H is the unit vector representing the direction halfway between the light vector and the view vector, known as the half-angle vector. S_{exp} is the specular exponent, controlling the tightness of the specular highlight. C_{spec} is the resulting specular color. N , C_{mat} , and C_{li} are the same as for diffuse lighting. Because the light in the example is white, you can again omit that term. [Figure 22.3](#) illustrates the output of [Listings 22.5](#) and [22.6](#), which implement both diffuse and specular lighting equations.

Listing 22.5. Diffuse and Specular Lighting High-Level Vertex Shader

```

// specular.vs
//
// Generic vertex transformation,
// diffuse and specular lighting
// based on one white light
uniform vec3 lightPos0;
void main(void)
{
    // normal MVP transform
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    vec3 N = normalize(gl_NormalMatrix * gl_Normal);
    vec4 V = gl_ModelViewMatrix * gl_Vertex;
    vec3 L = normalize(lightPos0 - V.xyz);

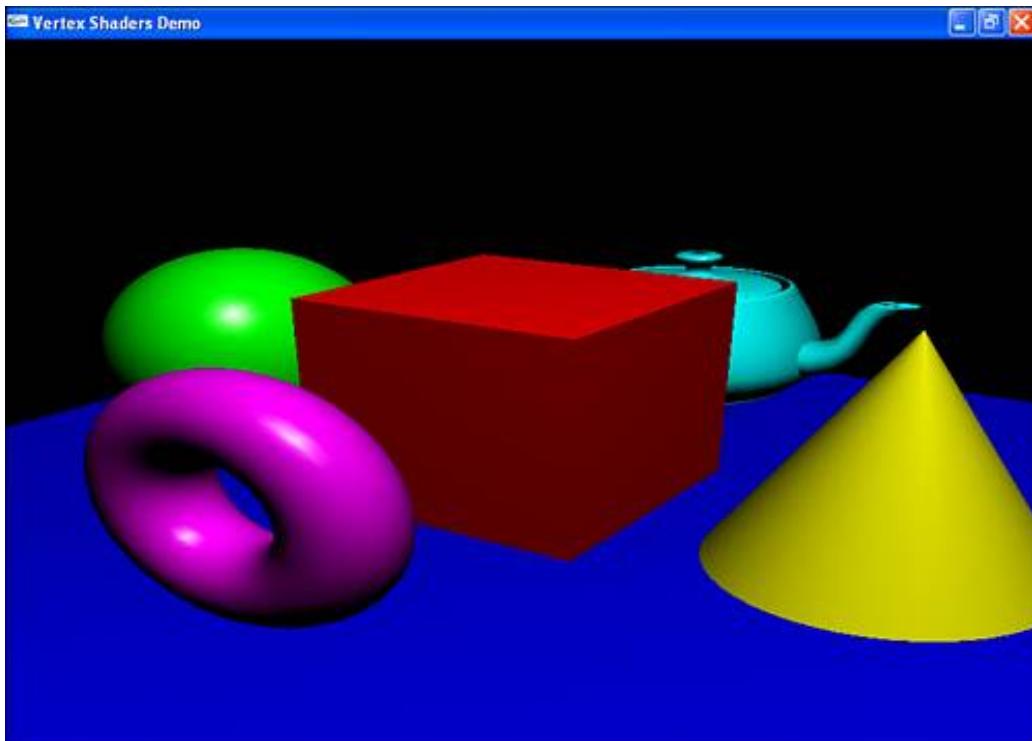
```

```

vec3 H = normalize(L + vec3(0.0, 0.0, 1.0));
const float specularExp = 128.0;
// calculate diffuse lighting
float NdotL = dot(N, L);
vec4 diffuse = gl_Color * vec4(max(0.0, NdotL));
// calculate specular lighting
float NdotH = dot(N, H);
vec4 specular = vec4(pow(max(0.0, NdotH), specularExp));
// sum the diffuse and specular components
gl_FrontColor = diffuse + specular;
}

```

Figure 22.3. This vertex shader computes diffuse and specular lighting.



Listing 22.6. Diffuse and Specular Lighting Low-Level Vertex Shader

```

!!ARBvp1.0
# specular.vp
#
# Generic vertex transformation,
# diffuse and specular lighting
# based on one white light
ATTRIB iPos = vertex.position;           # input position
ATTRIB iPrC = vertex.color.primary;       # input primary color
ATTRIB iNrm = vertex.normal;              # input normal
OUTPUT oPos = result.position;            # output position
OUTPUT oPrC = result.color.primary;        # output primary color
PARAM mvp[4] = { state.matrix.mvp };        # modelview * proj matrix
PARAM mv[4] = { state.matrix.modelview };    # modelview matrix
# inverse transpose of modelview matrix:
PARAM mvIT[4] = { state.matrix.modelview.invtrans };
PARAM lightPos = program.local[0];          # light pos in eye space
TEMP N, V, L, H, NdotL, NdotH;              # temporary registers
TEMP diffuse, specular;
DP4 oPos.x, iPos, mvp[0];                  # xform input pos by MVP
DP4 oPos.y, iPos, mvp[1];

```

```

DP4 oPos.z, iPos,.mvp[2];
DP4 oPos.w, iPos,.mvp[3];
DP4 V.x, iPos, mv[0];
DP4 V.y, iPos, mv[1];
DP4 V.z, iPos, mv[2];
DP4 V.w, iPos, mv[3];
SUB L, lightPos, V;
DP3 N.x, iNrm, mvIT[0];
DP3 N.y, iNrm, mvIT[1];
DP3 N.z, iNrm, mvIT[2];
DP3 N.w, N, N;
RSQ N.w, N.w;
MUL N, N, N.w;
DP3 L.w, L, L;
RSQ L.w, L.w;
MUL L, L, L.w;
ADD H.xyz, L, {0, 0, 1};
DP3 H.w, H, H;
RSQ H.w, H.w;
MUL H, H, H.w;
DP3 NdotL, N, L;
MAX NdotL, NdotL, 0.0;
MUL diffuse, iPrC, NdotL;
DP3 NdotH, N, H;
MAX NdotH, NdotH, 0.0;
POW specular, NdotH.x, 128.0.x;
ADD oPrC, diffuse, specular;
# sum the colors
END

```

The light position is a constant vector passed into the shader from the application. This allows you to easily change the light position interactively without having to alter the shader. You can do this using the left- and right-arrow keys while running the `VertexShaders` example.

We used a hard-coded constant specular exponent of 128, which provides a nice, tight specular highlight. You can experiment with different values to find one you may prefer. Notice in the low-level shader how we supply the exponent as `128.0.x`. We supplied it this way because a scalar value is expected, and the low-level grammar is not smart enough to realize that we're providing a scalar constant, so we have to provide the redundant suffix anyway.

Improved Specular Lighting

Specular highlights change rapidly over the surface of an object. Trying to compute them per-vertex and then interpolating the result across a triangle gives relatively poor results. Instead of a nice circular highlight, you end up with a muddy polygonal-shaped highlight.

One way you can improve the situation is to separate the diffuse lighting result from the specular lighting result, outputting one as the vertex's primary color and the other as the secondary color. By adding the diffuse and specular colors together, you effectively saturate the color (that is, exceed a value of 1.0) wherever a specular highlight appears. If you try to interpolate the sum of these colors, the saturation will more broadly affect the entire triangle. However, if you interpolate the two colors separately and then sum them per fragment, the saturation will occur only where desired, cleaning up some of the muddiness. This sum per fragment is achieved by simply enabling `GL_COLOR_SUM`. Here is the altered GLSL code for separating the two lit colors:

```

// put diffuse into primary color
float NdotL = dot(N, L);
gl_FrontColor = gl_Color * vec4(max(0.0, NdotL));
// put specular into secondary color
float NdotH = dot(N, H);

```

```
gl_FrontSecondaryColor = vec4(pow(max(0.0, NdotH), specularExp));
```

The altered low-level vertex shader code is as follows:

```
OUTPUT oPrC = result.color.primary;           # output primary color
OUTPUT oScC = result.color.secondary;         # output secondary color
...
DP3 NdotL, N, L;                            # N . L
MAX NdotL, NdotL, 0.0;
MUL oPrC, iPrC, NdotL;                      # diffuse goes in primary
DP3 NdotH, N, H;                            # N • H
MAX NdotH, NdotH, 0.0;
POW oScC, NdotH.x, 128.0.x;                 # spec goes in secondary
```

Separating the colors improves things a bit, but the root of the problem is the specular exponent. By raising the specular coefficient to a power, you have a value that wants to change much more rapidly than per-vertex interpolation allows. In fact, if your geometry is not tessellated finely enough, you may lose a specular highlight altogether.

An effective way to avoid this problem is to output just the specular coefficient ($N \cdot H$), but wait and raise it to a power per fragment. This way, you can safely interpolate the more slowly changing ($N \cdot H$). You're not employing fragment shaders yet, so how do you perform this power computation per fragment? All you have to do is set up a 1D texture with a table of s^{128} values and send ($N \cdot H$) out of the vertex shader on a texture coordinate. This is considered custom texgen. Then you will use fixed functionality texture environment to add the specular color from the texture lookup to the interpolated diffuse color from the vertex shader.

The following is the GLSL code, again altered from the original specular lighting shader:

```
// put diffuse lighting result in primary color
float NdotL = dot(N, L);
gl_FrontColor = gl_Color * vec4(max(0.0, NdotL));
// copy (N.H)*8-7 into texcoord
float NdotH = max(0.0, (dot(N, H) * 8.0) - 7.0);
gl_TexCoord[0] = vec4(NdotH, 0.0, 0.0, 1.0);
```

The altered low-level shader code is as follows:

```
OUTPUT oPrC = result.color.primary;           # output primary color
OUTPUT oTxC = result.texcoord[0];             # output texcoord 0
...
DP3 NdotL, N, L;                            # N . L
MAX NdotL, NdotL, 0.0;
MUL oPrC, iPrC, NdotL;                      # output diffuse
DP3 NdotH, N, H;                            # N • H
MAD NdotH.x, NdotH, 8.0, {-7.0};            # (N • H) * 8 - 7
MOV oTxC, {0.0, 0.0, 0.0, 1.0};             # init other components
MAX oTxC.x, NdotH, 0.0;                     # toss into texcoord 0
```

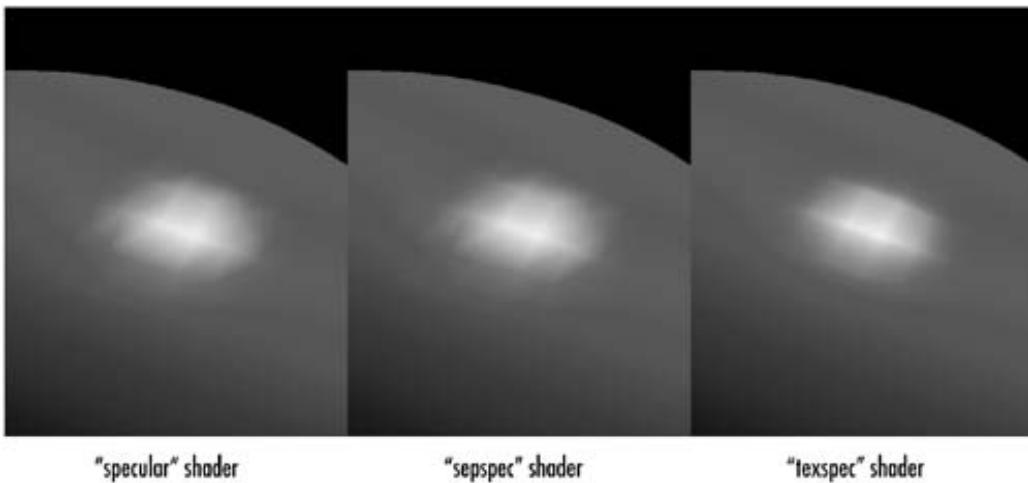
Here, the ($N \cdot H$) has been clamped to the range [0,1]. But if you try raising most of that range to the power of 128, you'll get results so close to zero that they will correspond to texel values of zero. Only the upper 1/8 of ($N \cdot H$) values will begin mapping to measurable texel values. To make economical use of the 1D texture, you can focus in on this upper 1/8 and fill the entire texture with values from this range, improving the resulting precision. This requires that you scale ($N \cdot H$) by 8 and bias by -7 so that [0,1] maps to $[-7,1]$. By using the `GL_CLAMP_TO_EDGE` wrap mode, values in the range $[-7,0]$ will be clamped to 0. Values in the range of interest, [0,1], will receive texel values between $(7/8)^{128}$ and 1.

The specular contribution resulting from the texture lookup is added to the diffuse color output

from the vertex shader using the `GL_ADD` texture environment function.

Figure 22.4 compares the three specular shaders to show the differences in quality. An even more precise method would be to output only the normal vector from the vertex shader and to encode a cube map texture so that at every N coordinate the resulting texel value is $(N \cdot H)^{128}$. We've left this as an exercise for you.

Figure 22.4. The per-vertex specular highlight is improved by using separate specular or a specular exponent texture.



Now that you have a decent specular highlight, you can get a little fancier and take the one white light and replicate it into three colored lights. This activity involves performing the same computations, except now you have three different light positions and you have to take the light color into consideration.

As has been the case with all the lighting shaders, you can change the light positions in the sample by using the left- and right-arrow keys. Figure 22.5 shows the three lights in action, produced by Listings 22.7 and 22.8.

Listing 22.7. Three Colored Lights High-Level Vertex Shader

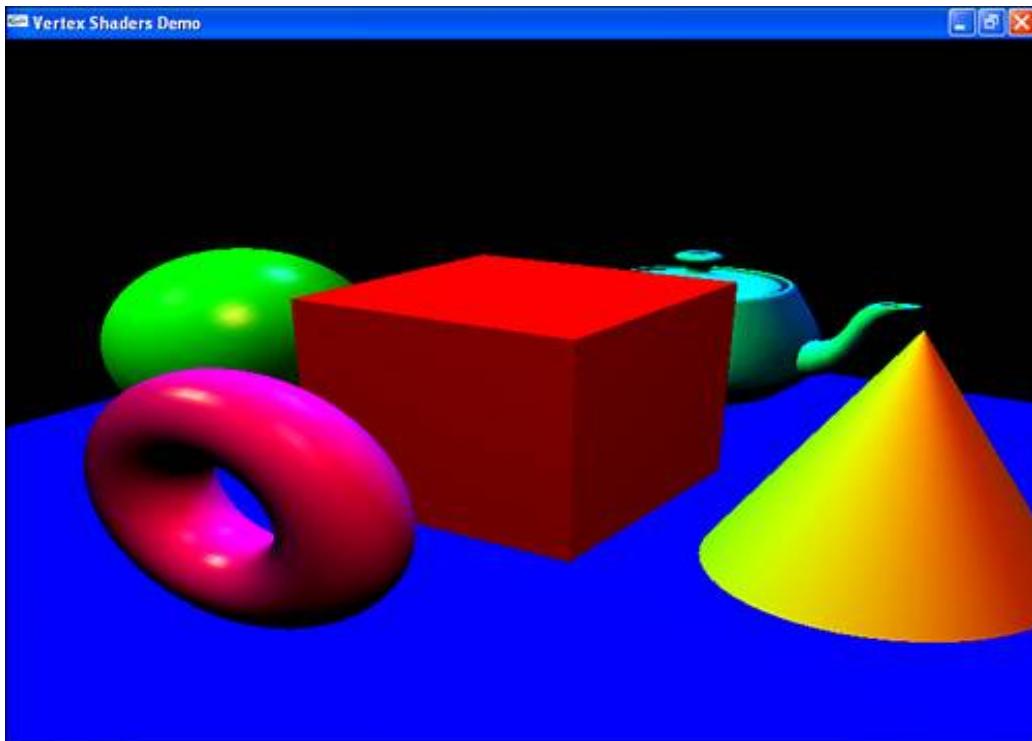
```
// 3lights.vs
//
// Generic vertex transformation,
// 3 colored lights
uniform vec3 lightPos0;
uniform vec3 lightPos1;
uniform vec3 lightPos2;
varying vec4 gl_TexCoord[4];
void main(void)
{
    // normal MVP transform
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    vec3 N = normalize(gl_NormalMatrix * gl_Normal);
    vec4 V = gl_ModelViewMatrix * gl_Vertex;
    // Light colors
    vec4 lightCol[3];
    lightCol[0] = vec4(1.0, 0.25, 0.25, 1.0);
    lightCol[1] = vec4(0.25, 1.0, 0.25, 1.0);
    lightCol[2] = vec4(0.25, 0.25, 1.0, 1.0);
    // Light vectors
    vec3 L[3], H[3];
    L[0] = normalize(lightPos0 - V.xyz);
```

```

L[1] = normalize(lightPos1 - V.xyz);
L[2] = normalize(lightPos2 - V.xyz);
gl_FrontColor = vec4(0.0);
for (int i = 0; i < 3; i++)
{
    // Half-angles
    H[i] = normalize(L[i] + vec3(0.0, 0.0, 1.0));
    // Accumulate the diffuse contributions
    gl_FrontColor += gl_Color * lightCol[i] *
        vec4(max(0.0, dot(N, L[i])));
    // Put N.H specular coefficients into texcoords
    gl_TexCoord[1+i] = vec4(max(0.0, dot(N, H[i])) * 8.0 - 7.0),
        0.0, 0.0, 1.0);
}
}

```

Figure 22.5. Three lights are better than one, though it's hard to tell in black and white.



Listing 22.8. Three Colored Lights Low-Level Vertex Shader

```

!!ARBvp1.0
# 3lights.vp
#
# Generic vertex transformation,
# 3 colored lights
ATTRIB iPos = vertex.position;           # input position
ATTRIB iPrC = vertex.color.primary;      # input primary color
ATTRIB iNrm = vertex.normal;             # input normal
OUTPUT oPos = result.position;          # output position
OUTPUT oPrC = result.color.primary;      # output primary color
OUTPUT oTC0 = result.texcoord[1];        # output texcoord 1
OUTPUT oTC1 = result.texcoord[2];        # output texcoord 2
OUTPUT oTC2 = result.texcoord[3];        # output texcoord 3
PARAM mvp[4] = { state.matrix.mvp };      # modelview * proj mat
PARAM mv[4] = { state.matrix.modelview }; # modelview matrix

```

```

# inverse transpose of modelview matrix:
PARAM mvIT[4] = { state.matrix.modelview.invtrans };
PARAM lightCol0 = { 1.0, 0.25, 0.25, 1.0 }; # light 0 color
PARAM lightCol1 = { 0.25, 1.0, 0.25, 1.0 }; # light 1 color
PARAM lightCol2 = { 0.25, 0.25, 1.0, 1.0 }; # light 2 color
PARAM lightPos0 = program.local[0]; # light pos 0 eye space
PARAM lightPos1 = program.local[1]; # light pos 1 eye space
PARAM lightPos2 = program.local[2]; # light pos 2 eye space
TEMP N, V, L, H, NdotL, NdotH, finalColor; # temporary registers
ALIAS diffuse = NdotL;
ALIAS specular = NdotH;
DP4 oPos.x, iPos, mvp[0]; # xform input pos by MVP
DP4 oPos.y, iPos, mvp[1];
DP4 oPos.z, iPos, mvp[2];
DP4 oPos.w, iPos, mvp[3];
DP4 V.x, iPos, mv[0];
DP4 V.y, iPos, mv[1];
DP4 V.z, iPos, mv[2];
DP4 V.w, iPos, mv[3];
DP3 N.x, iNrm, mvIT[0];
DP3 N.y, iNrm, mvIT[1];
DP3 N.z, iNrm, mvIT[2];
DP3 N.w, N, N;
RSQ N.w, N.w; # normalize normal
MUL N, N, N.w;
# LIGHT 0
SUB L, lightPos0, V; # vertex to light vector
DP3 L.w, L, L;
RSQ L.w, L.w; # normalize light vector
MUL L, L, L.w;
ADD H.xyz, L, {0, 0, 1};
DP3 H.w, H, H; # normalize half-angle
RSQ H.w, H.w;
MUL H, H, H.w;
DP3 NdotL, N, L; # N . L0
MAX NdotL, NdotL, 0.0;
MUL diffuse, iPrC, NdotL; # priCol * lightCol0 * N.L0
MUL finalColor, diffuse, lightCol0;
DP3 NdotH, N, H; # N . H0
MAX NdotH, NdotH, 0.0;
MOV oTC0, {0.0, 0.0, 0.0, 1.0}; # NdotH * 8 - 7 to tc 0
MAD oTC0.x, NdotH, 8, {-7}; # vertex to light vector
# LIGHT 1
SUB L, lightPos1, V; # normalize light vector
DP3 L.w, L, L;
RSQ L.w, L.w; # normalize half-angle
MUL L, L, L.w;
ADD H.xyz, L, {0, 0, 1};
DP3 H.w, H, H; # N . L1
RSQ H.w, H.w;
MUL H, H, H.w;
DP3 NdotL, N, L; # priCol * lightCol0 * N.L1
MAX NdotL, NdotL, 0.0;
MUL diffuse, iPrC, NdotL; # priCol * lightCol0 * N.L1
MAD finalColor, diffuse, lightCol1, finalColor;
DP3 NdotH, N, H; # N . H1
MAX NdotH, NdotH, 0.0;
MOV oTC1, {0.0, 0.0, 0.0, 1.0}; # NdotH * 8 - 7 to tc 1
MAD oTC1.x, NdotH, 8, {-7}; # NdotH * 8 - 7 to tc 1

```

```

# LIGHT 2
SUB L, lightPos2, V;                                # vertex to light vector
DP3 L.w, L, L;                                     # normalize light vector
RSQ L.w, L.w;
MUL L, L, L.w;
ADD H.xyz, L, {0, 0, 1};                            # normalize half-angle
DP3 H.w, H, H;
RSQ H.w, H.w;
MUL H, H, H.w;
DP3 NdotL, N, L;                                     # N . L2
MAX NdotL, NdotL, 0.0;
MUL diffuse, iPrC, NdotL;                            # priCol * N.L2
# priCol * lightCol0 * N.L2
MAD oPrC, diffuse, lightCol2, finalColor;
DP3 NdotH, N, H;                                     # N . H2
MAX NdotH, NdotH, 0.0;
MOV oTC2, {0.0, 0.0, 0.0, 1.0};
MAD oTC2.x, NdotH, 8, {-7};                          # NdotH * 8 - 7 to tc 2
END

```

Interesting to note in this sample is the use of a loop in the GLSL version. Loops are not available in low-level shaders, so we've "unrolled" the loops into a linear sequence; consequently, the code that would be in the loop is replicated three times, once for each light. Even though GLSL permits them, some OpenGL implementations may not support loops in hardware anyway. So if your shader is running really slow, it may be emulating your shader execution in software. Unrolling the loop in your GLSL shader could alleviate the problem, but at the expense of making your code less readable.

Per-Vertex Fog

Though fog is specified as a per-fragment rasterization stage that follows texturing, often implementations perform most of the necessary computation per-vertex and then interpolate the results across the primitive. This shortcut is sanctioned by the OpenGL specification because it improves performance with very little loss of image fidelity. The following is the equation for a second-order exponential fog factor, which controls the blending between the fog color and the unfogged fragment color:

$$ff = e^{-(d*fc)^2}$$

In this equation, ff is the computed fog factor. d is the density constant that controls the "thickness" of the fog. fc is the fog coordinate, which is usually the distance from the vertex to the eye, or is approximated by the absolute value of the vertex position's Z component in eye space. In this chapter's sample shaders, you'll compute the actual distance, not an approximation.

In the first sample fog shader, you'll compute only the fog coordinate and leave it to fixed functionality to compute the fog factor and perform the blend. In the second sample, you'll compute the fog factor yourself within the vertex shader and also perform the blending per-vertex. Performing all these operations per-vertex instead of per-fragment is more efficient and provides acceptable results for most uses. [Figure 22.6](#) illustrates the fogged scene, which is nearly identical for the two sample fog shaders in [Listings 22.9](#) and [22.10](#).

Listing 22.9. Fog Coordinate Generating High-Level Vertex Shader

```

// fogcoord.vs
//
// Generic vertex transformation,
// diffuse and specular lighting,
// per-vertex fogcoord
uniform vec3 lightPos0;

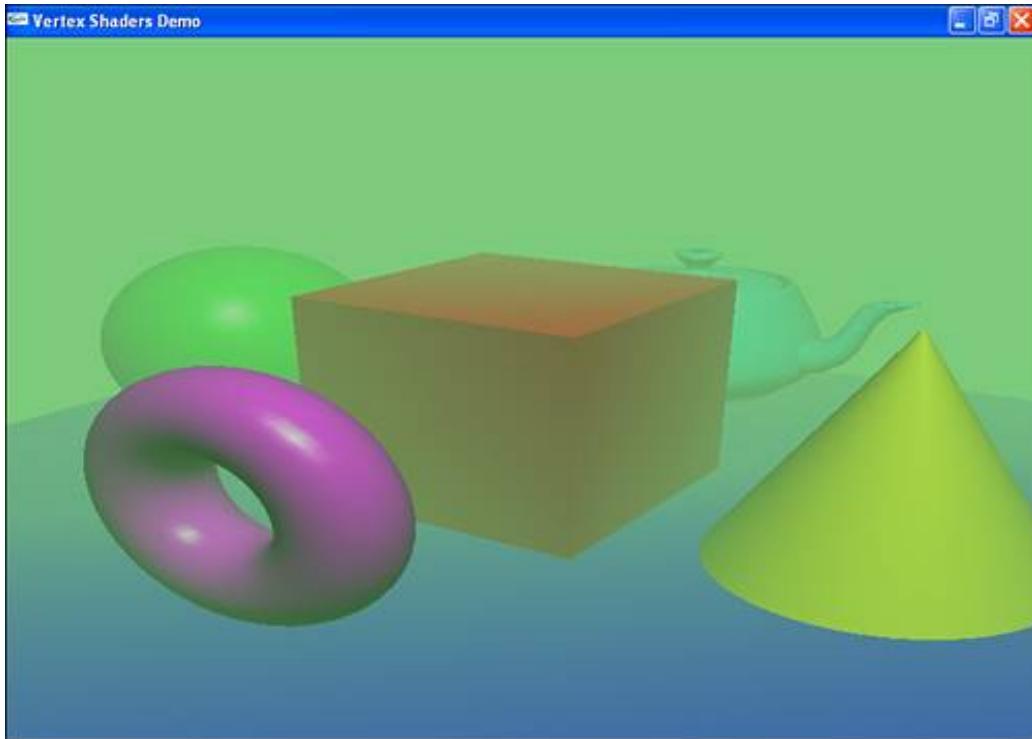
```

```

void main(void)
{
    // normal MVP transform
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    vec3 N = normalize(gl_NormalMatrix * gl_Normal);
    vec4 V = gl_ModelViewMatrix * gl_Vertex;
    vec3 L = normalize(lightPos0 - V.xyz);
    vec3 H = normalize(L + vec3(0.0, 0.0, 1.0));
    const float specularExp = 128.0;
    // calculate diffuse lighting
    float NdotL = dot(N, L);
    vec4 diffuse = gl_Color * vec4(max(0.0, NdotL));
    // calculate specular lighting
    float NdotH = dot(N, H);
    vec4 specular = vec4(pow(max(0.0, NdotH), specularExp));
    // calculate fog coordinate: distance from eye
    gl_FogFragCoord = length(V);
    // sum the diffuse and specular components
    gl_FrontColor = diffuse + specular;
}

```

Figure 22.6. Applying per-vertex fog using a vertex shader.



Listing 22.10. Fog Coordinate Generating Low-Level Vertex Shader

```

!!ARBvp1.0
# fogcoord.vs
#
# Generic vertex transformation,
# diffuse and specular lighting,
# per-vertex fogcoord
ATTRIB iPos = vertex.position;           # input position
ATTRIB iPrC = vertex.color.primary;      # input primary color
ATTRIB iNrm = vertex.normal;             # input normal
OUTPUT oPos = result.position;          # output position
OUTPUT oPrC = result.color.primary;      # output primary color

```

```

OUTPUT oFgC = result.fogcoord;           # output fog coordinate
PARAM mvp[4] = { state.matrix.mvp };      # modelview * proj matrix
PARAM mv[4] = { state.matrix.modelview }; # modelview matrix
# inverse transpose of modelview matrix:
PARAM mvIT[4] = { state.matrix.modelview.invtrans };
PARAM lightPos = program.local[0];        # light pos in eye space
PARAM density = program.local[1];         # fog density
TEMP N, V, L, H, NdotL, NdotH;           # temporary registers
TEMP diffuse, specular, fogCoord;
DP4 oPos.x, iPos, mvp[0];                # xform input pos by MVP
DP4 oPos.y, iPos, mvp[1];
DP4 oPos.z, iPos, mvp[2];
DP4 oPos.w, iPos, mvp[3];
DP4 V.x, iPos, mv[0];                   # xform input pos by MV
DP4 V.y, iPos, mv[1];
DP4 V.z, iPos, mv[2];
DP4 V.w, iPos, mv[3];
SUB L, lightPos, V;                     # vertex to light vector
DP3 N.x, iNrm, mvIT[0];
DP3 N.y, iNrm, mvIT[1];
DP3 N.z, iNrm, mvIT[2];
DP3 N.w, N, N;                         # normalize normal
RSQ N.w, N.w;
MUL N, N, N.w;
DP3 L.w, L, L;                         # normalize light vector
RSQ L.w, L.w;
MUL L, L, L.w;
ADD H.xyz, L, {0, 0, 1};
DP3 H.w, H, H;                         # normalize half-angle
RSQ H.w, H.w;
MUL H, H, H.w;
DP3 NdotL, N, L;                       # N . L
MAX NdotL, NdotL, 0.0;
MUL diffuse, iPrC, NdotL;
DP3 NdotH, N, H;                       # N • H
MAX NdotH, NdotH, 0.0;
POW specular, NdotH.x, 128.0.x;        # 128 is specular exponent
ADD oPrC, diffuse, specular;
DP4 fogCoord.x, V, V;                  # sum the colors
RSQ fogCoord.x, fogCoord.x;
RCP oFgC.x, fogCoord.x;
END

```

The calculation to find the distance from the eye (0,0,0,1) to the vertex in eye space is trivial in GLSL. You need only call the built-in length function, passing in the vertex position vector as an argument. In the low-level shader, you must manually perform the same operation. First, you take the dot product of the vertex position against itself followed by the reciprocal square root, just as if you were normalizing the vector. But instead of multiplying this 1/length scale factor by the vector, you just take its reciprocal so the 1/length becomes the length, which is output directly as the vertex's fog coordinate.

The following is the altered GLSL code for performing the fog blend within the shader instead of in fixed functionality fragment processing:

```

uniform float density;
...
// calculate 2nd order exponential fog factor
const float e = 2.71828;
float fogFactor = (density * length(V));
fogFactor *= fogFactor;
fogFactor = clamp(pow(e, -fogFactor), 0.0, 1.0);

```

```

// sum the diffuse and specular components, then
// blend with the fog color based on fog factor
const vec4 fogColor = vec4(0.5, 0.8, 0.5, 1.0);
gl_FrontColor = mix(fogColor, clamp(diffuse + specular, 0.0, 1.0),
                     fogFactor);

```

The altered low-level shader code is as follows:

```

PARAM density = program.local[1];           # fog density
PARAM fogColor = {0.5, 0.8, 0.5, 1.0};      # fog color
PARAM e = {2.71828, 0, 0, 0};
TEMP diffuse, specular, fogFactor, litColor;
...
ADD litColor, diffuse, specular;           # sum the colors
MAX litColor, litColor, 0.0;                # clamp to [0,1]
MIN litColor, litColor, 1.0;
# fogFactor = clamp(e^(-(d*|Ve|)^2))
DP4 fogFactor.x, V, V;
POW fogFactor.x, fogFactor.x, 0.5.x;
MUL fogFactor.x, fogFactor.x, density.x;
MUL fogFactor.x, fogFactor.x, fogFactor.x;
POW fogFactor.x, e.x, -fogFactor.x;
MAX fogFactor.x, fogFactor.x, 0.0;          # clamp to [0,1]
MIN fogFactor.x, fogFactor.x, 1.0;
SUB litColor, litColor, fogColor;           # blend lit and fog colors
MAD oPrC, fogFactor.x, litColor, fogColor;
END

```

Per-Vertex Point Size

Applying fog attenuates object colors the farther away they are from the viewpoint. Similarly, you can attenuate point sizes so that points rendered close to the viewpoint are relatively large and points farther away diminish into nothing. Like fog, point attenuation is a useful visual cue for conveying perspective. The computation required is similar as well.

You compute the distance from the vertex to the eye exactly the same as you did for the fog coordinate. Then, to get a point size that falls off exponentially with distance, you square the distance, take its reciprocal, and multiply it by the constant 100,000. This constant is chosen specifically for this scene's geometry so that objects toward the back of the scene, as rendered from the initial camera position, are assigned point sizes of approximately 1, whereas points near the front are assigned point sizes of approximately 10.

In this sample application, you'll set the polygon mode for front- and back-facing polygons to `GL_POINT` so that all the objects in the scene are drawn with points. Also, you must enable `GL_VERTEX_PROGRAM_POINT_SIZE_ARB` so that the point sizes output from the vertex shader are substituted in place of the usual OpenGL point size. [Figure 22.7](#) shows the result of [Listings 22.11](#) and [22.12](#).

Listing 22.11. Point Size Generating High-Level Vertex Shader

```

// ptsize.vs
//
// Generic vertex transformation,
// attenuated point size
void main(void)
{
    // normal MVP transform
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    vec4 V = gl_ModelViewMatrix * gl_Vertex;

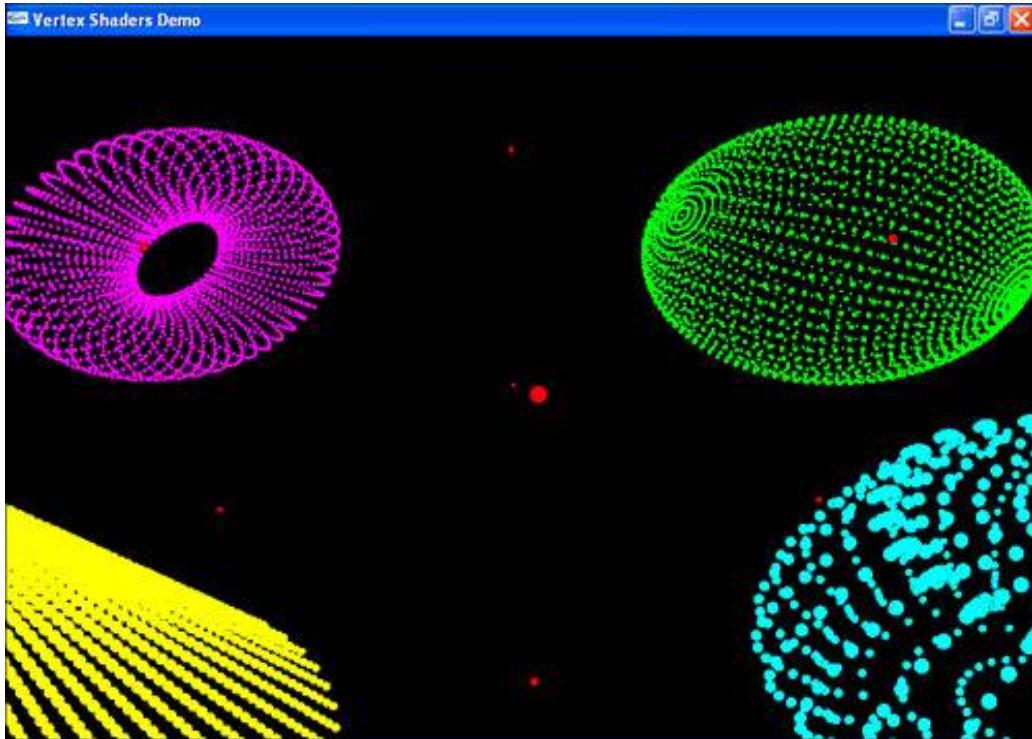
```

```

gl_FrontColor = gl_Color;
// calculate point size based on distance from eye
float ptSize = length(V);
ptSize *= ptSize;
gl_PointSize = 100000.0 / ptSize;
}

```

Figure 22.7. Per-vertex point size makes distant points smaller.



Listing 22.12. Point Size Generating Low-Level Vertex Shader

```

!!ARBvp1.0
# ptsize.vs
#
# Generic vertex transformation,
# attenuated point size
ATTRIB iPos = vertex.position;           # input position
ATTRIB iPrC = vertex.color.primary;       # input primary color
OUTPUT oPos = result.position;            # output position
OUTPUT oPrC = result.color.primary;        # output primary color
OUTPUT oPtS = result.pointsize;            # output point size
PARAM mvp[4] = { state.matrix.mvp };       # modelview * proj matrix
PARAM mv[4] = { state.matrix.modelview };  # modelview matrix
TEMP V, ptSize;                          # temporary registers
DP4 oPos.x, iPos, mvp[0];                # xform input pos by MVP
DP4 oPos.y, iPos, mvp[1];
DP4 oPos.z, iPos, mvp[2];
DP4 oPos.w, iPos, mvp[3];
DP4 V.x, iPos, mv[0];
DP4 V.y, iPos, mv[1];
DP4 V.z, iPos, mv[2];
DP4 V.w, iPos, mv[3];
MOV oPrC, iPrC;                          # copy color
DP4 ptSize.x, V, V;                      # ptSize = 100000 / |Ve|^2
RSQ ptSize.x, ptSize.x;
MUL ptSize.x, ptSize.x, ptSize.x;

```

```
MUL oPtS.x, ptSize.x, 100000;
END
```

Customized Vertex Transformation

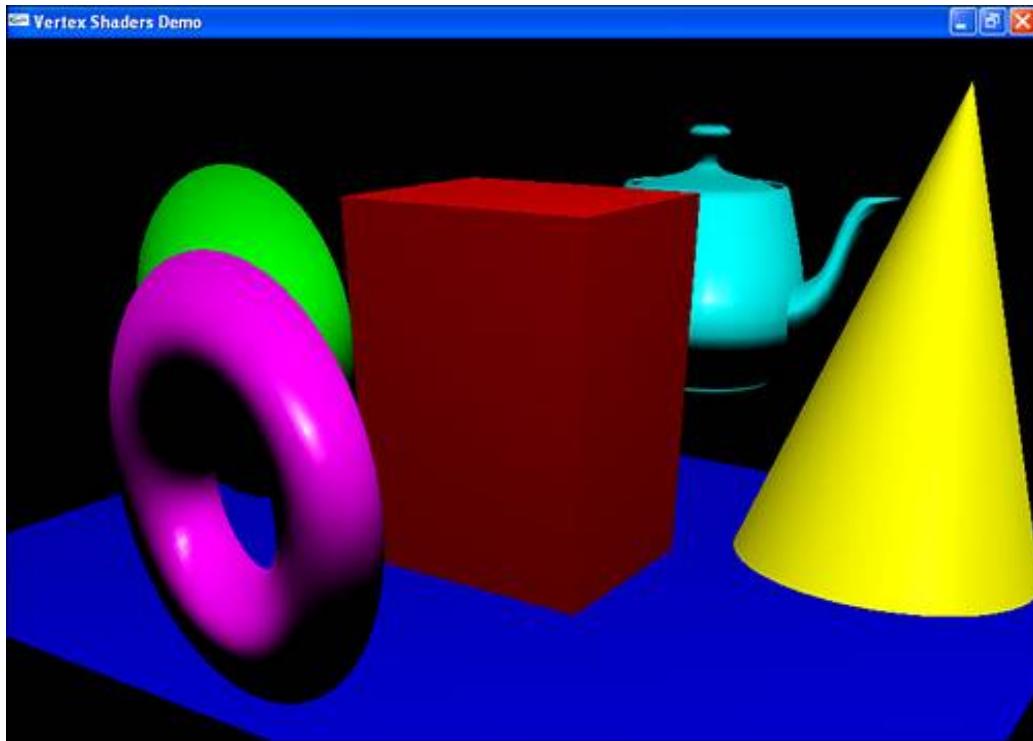
You've already customized lighting, texture coordinate generation, and fog coordinate generation. But what about the vertex positions themselves? The next sample shader applies an additional transformation before transforming by the usual modelview/projection matrix.

[Figure 22.8](#) shows the effects of scaling the object-space vertex position by a squash and stretch factor, which can be set independently for each axis, as in [Listings 22.13](#) and [22.14](#).

Listing 22.13. Squash and Stretch High-Level Vertex Shader

```
// stretch.vs
//
// Generic vertex transformation,
// followed by squash/stretch
uniform vec3 lightPos0;
uniform vec3 squashStretch;
void main(void)
{
    // normal MVP transform, followed by squash/stretch
    vec4 stretchedCoord = gl_Vertex;
    stretchedCoord.xyz *= squashStretch;
    gl_Position = gl_ModelViewProjectionMatrix * stretchedCoord;
    vec3 stretchedNormal = gl_Normal;
    stretchedNormal *= squashStretch;
    vec3 N = normalize(gl_NormalMatrix * stretchedNormal);
    vec4 V = gl_ModelViewMatrix * stretchedCoord;
    vec3 L = normalize(lightPos0 - V.xyz);
    vec3 H = normalize(L + vec3(0.0, 0.0, 1.0));
    // put diffuse lighting result in primary color
    float NdotL = dot(N, L);
    gl_FrontColor = gl_Color * vec4(max(0.0, NdotL));
    // copy (N.H)*8-7 into texcoord
    float NdotH = max(0.0, dot(N, H) * 8.0 - 7.0);
    gl_TexCoord[0] = vec4(NdotH, 0.0, 0.0, 1.0);
}
```

Figure 22.8. Squash and stretch effects customize the vertex transformation.



Listing 22.14. Squash and Stretch Low-Level Vertex Shader

```
!!ARBvp1.0
# stretch.vs
#
# Generic vertex transformation,
# followed by squash/stretch
ATTRIB iPos = vertex.position;           # input position
ATTRIB iPrC = vertex.color.primary;       # input primary color
ATTRIB iNrm = vertex.normal;              # input normal
OUTPUT oPos = result.position;           # output position
OUTPUT oPrC = result.color.primary;       # output primary color
OUTPUT oTxC = result.texcoord[0];         # output texcoord 0
PARAM mvp[4] = { state.matrix.mvp };      # modelview * proj matrix
PARAM mv[4] = { state.matrix.modelview }; # modelview matrix
# inverse transpose of modelview matrix:
PARAM mvIT[4] = { state.matrix.modelview.invtrans };
PARAM lightPos = program.local[0];         # light pos in eye space
PARAM squashStretch = program.local[1];     # stretch scale factors
TEMP N, V, L, H, NdotL, NdotH, ssV, ssN; # temporary registers
MUL ssV, iPos, squashStretch;              # stretch obj-space vertex
MUL ssN, iNrm, squashStretch;              # stretch obj-space normal
DP4 oPos.x, ssV, mvp[0];                  # xform stretch pos by MVP
DP4 oPos.y, ssV, mvp[1];
DP4 oPos.z, ssV, mvp[2];
DP4 oPos.w, ssV, mvp[3];
DP4 V.x, ssV, mv[0];                      # xform stretch pos by MV
DP4 V.y, ssV, mv[1];
DP4 V.z, ssV, mv[2];
DP4 V.w, ssV, mv[3];
SUB L, lightPos, V;                        # vertex to light vector
DP3 N.x, ssN, mvIT[0];                    # xform stretched normal
DP3 N.y, ssN, mvIT[1];
DP3 N.z, ssN, mvIT[2];
DP3 N.w, N, N;                           # normalize normal
RSQ N.w, N.w;
MUL N, N, N.w;
```

```

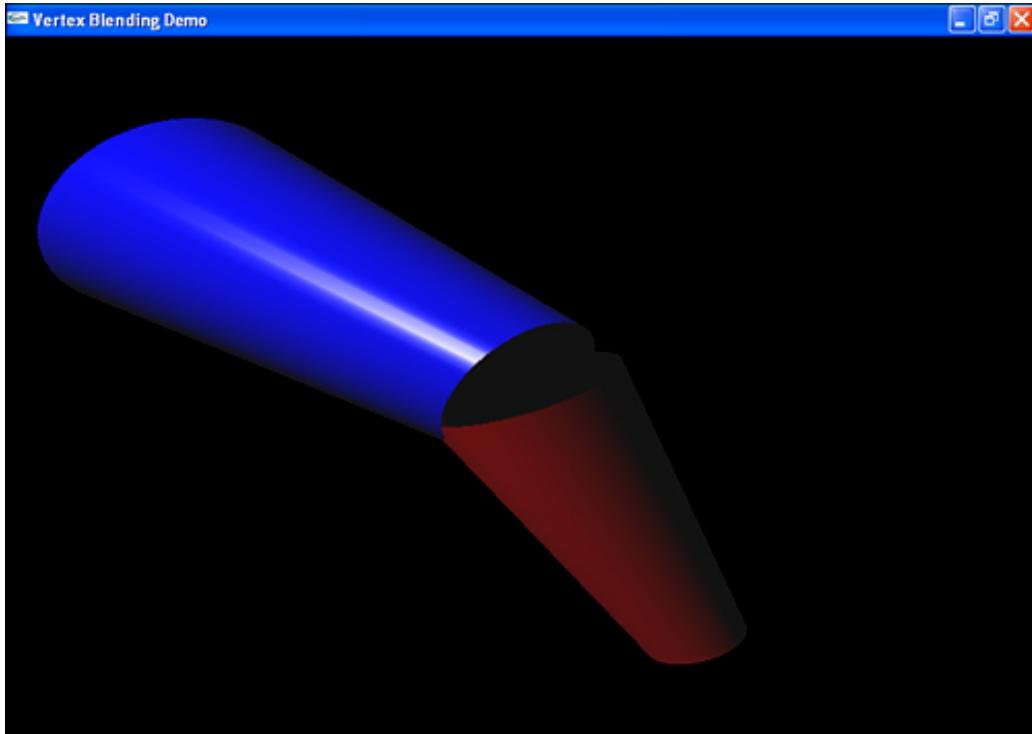
DP3 L.w, L, L;                                # normalize light vector
RSQ L.w, L.w;
MUL L, L, L.w;
ADD H.xyz, L, {0, 0, 1};
DP3 H.w, H, H;                                # normalize half-angle
RSQ H.w, H.w;
MUL H, H, H.w;
DP3 NdotL, N, L;                                # N . L
MAX NdotL, NdotL, 0.0;
MUL oPrC, iPrC, NdotL;                         # output diffuse
DP3 NdotH, N, H;                                # N • H
MAD NdotH.x, NdotH, 8.0, {-7.0};                # (N • H) * 8 - 7
MOV oTxC, {0.0, 0.0, 0.0, 1.0};
MAX oTxC.x, NdotH, 0.0;                         # toss into texcoord 0
END

```

Vertex Blending

Vertex blending is an interesting technique used for skeletal animation. Consider a simple model of an arm with an elbow joint. The forearm and bicep are each represented by a cylinder. When the arm is completely straight, all the "skin" is nicely connected together. But as soon as you bend the arm, as in [Figure 22.9](#), the skin is disconnected and the realism is gone.

Figure 22.9. This simple elbow joint without vertex blending just begs for skin.



The way to fix this problem is to employ multiple modelview matrices when transforming each vertex. Both the forearm and bicep have their own modelview matrix already. The bicep's matrix would orient it relative to the torso if it were attached to a body, or in this case relative to the origin in object-space. The forearm's matrix orients it relative to the bicep. The key to vertex blending is to use a little of each matrix when transforming vertices close to a joint.

You can choose how close to the joint you want the multiple modelview matrices to have influence. We call this the *region of influence*. Vertices outside the region of influence do not require blending. For such a vertex, only the original modelview matrix associated with the object is used. However, vertices that do fall within the region of influence must transform the vertex

twice: once with its own modelview matrix and once with the matrix belonging to the object on the other side of the joint. For this sample, you blend these two eye-space positions together to achieve the final eye-space position.

The amount of one eye-space position going into the mix versus the other is based on the vertex's blend weight. When drawing the `glBegin/glEnd` primitives, in addition to the usual normals, colors, and positions, you also specify a weight for each vertex. You use the `glVertexAttrib1fARB` function for specifying the weight. Vertices right at the edge of the joint receive weights of 0.5, effectively resulting in a 50% influence by each matrix. On the other extreme, vertices on the edge of the region of influence receive weights of 1.0, whereby the object's own matrix has 100% influence. Within the region of influence, weights vary from 1.0 to 0.5, and they can be assigned linearly with respect to the distance from the joint, or based on some higher-order function.

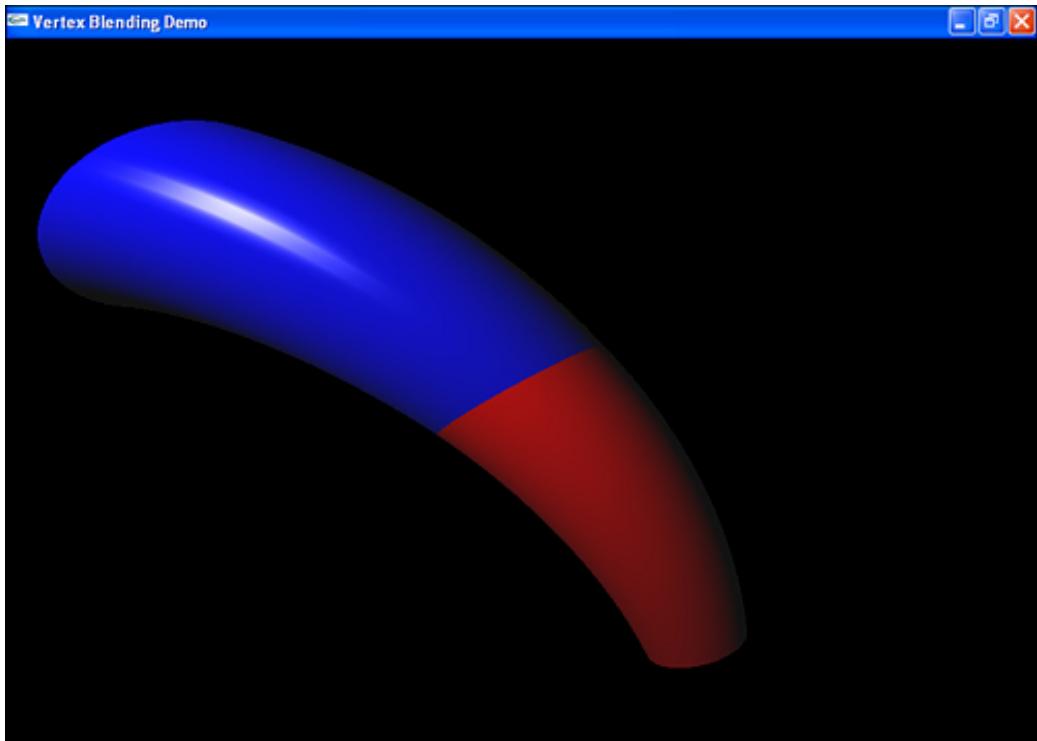
Any other computations dependent on the modelview matrix must also be blended. In the case of the sample shader, you also perform diffuse and specular lighting. This means the normal vector, which usually is transformed by the inverse transpose of the modelview matrix, now must also be transformed twice just like the vertex position. The two results are blended based on the same weights used for vertex position blending.

By using vertex blending, you can create life-like flexible skin on a skeleton structure that is easy to animate. [Figure 22.10](#) shows the arm in its new Elastic Man form, thanks to a region of influence covering the entire arm. [Listings 22.15](#) and [22.16](#) contain the vertex blending shader source.

Listing 22.15. Vertex Blending High-Level Vertex Shader

```
// skinning.vs
//
// Perform vertex skinning by
// blending between two MV
// matrices
uniform vec3 lightPos;
uniform mat4 mv2;
uniform mat3 mv2IT;
attribute float weight;
void main(void)
{
    // compute each vertex influence
    vec4 V1 = gl_ModelViewMatrix * gl_Vertex;
    vec4 V2 = mv2 * gl_Vertex;
    vec4 V = (V1 * weight) + (V2 * (1.0 - weight));
    gl_Position = gl_ProjectionMatrix * V;
    // compute each normal influence
    vec3 N1 = gl_NormalMatrix * gl_Normal;
    vec3 N2 = mv2IT * gl_Normal;
    vec3 N = normalize((N1 * weight) + (N2 * (1.0 - weight)));
    vec3 L = normalize(lightPos - V.xyz);
    vec3 H = normalize(L + vec3(0.0, 0.0, 1.0));
    // put diffuse lighting result in primary color
    float NdotL = dot(N, L);
    gl_FrontColor = 0.1 + gl_Color * vec4(max(0.0, NdotL));
    // copy (N.H)*8-7 into texcoord
    float NdotH = max(0.0, dot(N, H) * 8.0 - 7.0);
    gl_TexCoord[0] = vec4(NdotH, 0.0, 0.0, 1.0);
}
```

Figure 22.10. The stiff two-cylinder arm is now a fun, curvy, flexible object.



Listing 22.16. Vertex Blending Low-Level Vertex Shader

```

!!ARBvp1.0
# skinning.vp
#
# Perform vertex skinning by
# blending between two MV
# matrices
ATTRIB iPos = vertex.position;           # input position
ATTRIB iPrC = vertex.color.primary;      # input primary color
ATTRIB iNrm = vertex.normal;             # input normal
ATTRIB iWeight = vertex.attrib[1];        # input weight
OUTPUT oPos = result.position;           # output position
OUTPUT oPrC = result.color.primary;      # output primary color
OUTPUT oTxC = result.texcoord[0];         # output texcoord 0
PARAM prj[4] = { state.matrix.projection }; # projection matrix
PARAM mv1[4] = { state.matrix.modelview }; # modelview matrix 1
PARAM mv2[4] = { state.matrix.program[0] }; # modelview matrix 2
# inverse transpose of modelview matrix:
PARAM mv1IT[4] = { state.matrix.modelview.invtrans };
PARAM mv2IT[4] = { state.matrix.program[0].invtrans };
PARAM lightPos = program.local[0];        # light pos in eye space
TEMP N1, N2, N, V1, V2, V;                # temporary registers
TEMP L, H, NdotL, NdotH;                  # temporary registers
DP4 V1.x, iPos, mv1[0];                  # xform input pos by MV1
DP4 V1.y, iPos, mv1[1];
DP4 V1.z, iPos, mv1[2];
DP4 V1.w, iPos, mv1[3];
DP4 V2.x, iPos, mv2[0];                  # xform input pos by MV2
DP4 V2.y, iPos, mv2[1];
DP4 V2.z, iPos, mv2[2];
DP4 V2.w, iPos, mv2[3];
SUB V, V1, V2;                           # blend verts w/ weight
MAD V, V, iWeight.x, V2;
DP4 oPos.x, V, prj[0];                  # xform to clip space
DP4 oPos.y, V, prj[1];
DP4 oPos.z, V, prj[2];

```

```

DP4 oPos.w, V, prj[3];
SUB L, lightPos, V;
DP3 N1.x, iNrm, mv1IT[0];
DP3 N1.y, iNrm, mv1IT[1];
DP3 N1.z, iNrm, mv1IT[2];
DP3 N2.x, iNrm, mv2IT[0];
DP3 N2.y, iNrm, mv2IT[1];
DP3 N2.z, iNrm, mv2IT[2];
SUB N, N1, N2;
MAD N, N, iWeight.x, N2;
DP3 N.w, N, N;
RSQ N.w, N.w;
MUL N, N, N.w;
DP3 L.w, L, L;
RSQ L.w, L.w;
MUL L, L, L.w;
ADD H.xyz, L, {0, 0, 1};
DP3 H.w, H, H;
RSQ H.w, H.w;
MUL H, H, H.w;
DP3 NdotL, N, L;
MAX NdotL, NdotL, 0.0;
MAD oPrC, iPrC, NdotL, 0.1;
DP3 NdotH, N, H;
MAD NdotH.x, NdotH, 8.0, {-7.0};
MOV oTxC, {0.0, 0.0, 0.0, 1.0};
MAX oTxC.x, NdotH, 0.0;
# toss into texcoord 0
END

```

In this sample, you use built-in modelview matrix uniforms (GLSL) or parameters (low-level) to access the primary blend matrix. For the secondary matrix, you employ a user-defined uniform matrix (GLSL) or program matrix (low-level).

For normal transformation, you need the inverse transpose of each blend matrix. Although low-level shaders provide a simple way to access the inverse transpose of a matrix, high-level shaders do not. You continue to use the built-in `gl_NormalMatrix` for accessing the primary matrix's inverse transpose, but for the secondary matrix's inverse transpose, there is no shortcut. Instead, you manually compute the inverse of the second modelview matrix within the application and transpose it on the way into OpenGL when calling `glUniformMatrix3fvARB`.

Summary

This chapter provided various sample shaders as a jumping-off point for your own exploration of high- and low-level vertex shaders. Specifically, we provided examples of customized lighting, texture coordinate generation, fog, point size, and vertex transformation.

It is refreshing to give vertex shaders their moment in the spotlight. In reality, vertex shaders often play only supporting roles to their fragment shader counterparts, performing menial tasks such as preparing texture coordinates. Fragment shaders end up stealing the show. In the next chapter, we'll start by focusing solely on fragment shaders. Then in the stunning conclusion, we will see our vertex shader friends once again when we combine the two shaders and say goodbye to fixed functionality once and for all.

Chapter 23. Fragment Shading: Empower Your Pixel Processing

by Benjamin Lipchak

WHAT YOU'LL LEARN IN THIS CHAPTER:

- How to alter colors
- How to post-process images
- How to light an object per-fragment
- How to perform procedural texture mapping

As you may recall from [Chapter 19](#), "Programmable Pipeline: This Isn't Your Father's OpenGL," fragment shaders replace the texturing, color sum, and fog stages of the fixed functionality pipeline. This is the section of the pipeline where the party is happening. Instead of marching along like a mindless herd of cattle, applying each enabled texture based on its pre-ordained texture coordinate, your fragments are free to choose their own adventure. Mix and match textures and texture coordinates. Or calculate your own texture coordinates. Or don't do any texturing, and just compute your own colors. It's all good.

In their natural habitat, vertex shaders and fragment shaders are most often mated for life. Fragment shaders are the dominant partner, directly producing the eye candy you see displayed on the screen, and thus they receive the most attention. However, vertex shaders play an important supporting role. In the name of performance, as much of the grunt work as possible is pushed into vertex shaders because they tend to be executed much less frequently (except for the smallest of triangles). The results are then placed into interpolants for use as input by the fragment shader. The vertex shader is a selfless producer; the fragment shader a greedy consumer.

In this chapter, we continue the learning by example we began in the preceding chapter. We present many fragment shaders, both as further exposure to the low-level and high-level shading languages, and as a launch pad for your own future dabbling. Because you rarely see fragment shaders alone, once you get the hang of fragment shaders in isolation, we will move on to discuss several examples of vertex shaders and fragment shaders working together in peaceful harmony.

Color Conversion

We almost have to contrive some examples illustrating where fragment shaders are used without vertex shader assistance. But we can easily separate them where we simply want to alter the existing color. For these examples, we use fixed functionality lighting to provide a starting color. Then we go to town on it.

Grayscale

One thing you might want to do in your own work is simulate black-and-white film. Given the incoming red, green, and blue color channel intensities, we would like to calculate a single grayscale intensity to output to all three channels. Red, green, and blue each reflect light differently, which we represent by their different contributions to the final intensity.

[Listings 23.1](#) and [23.2](#) show the GLSL and [ARB_fragment_program](#) versions of the shader. Though you won't be able to distinguish between several of the shader results due to the black-and-white limitations of the figures, [Figure 23.1](#) is provided as a reference for some of the other shader results that are distinguishable.

Listing 23.1. Grayscale Conversion High-Level Fragment Shader

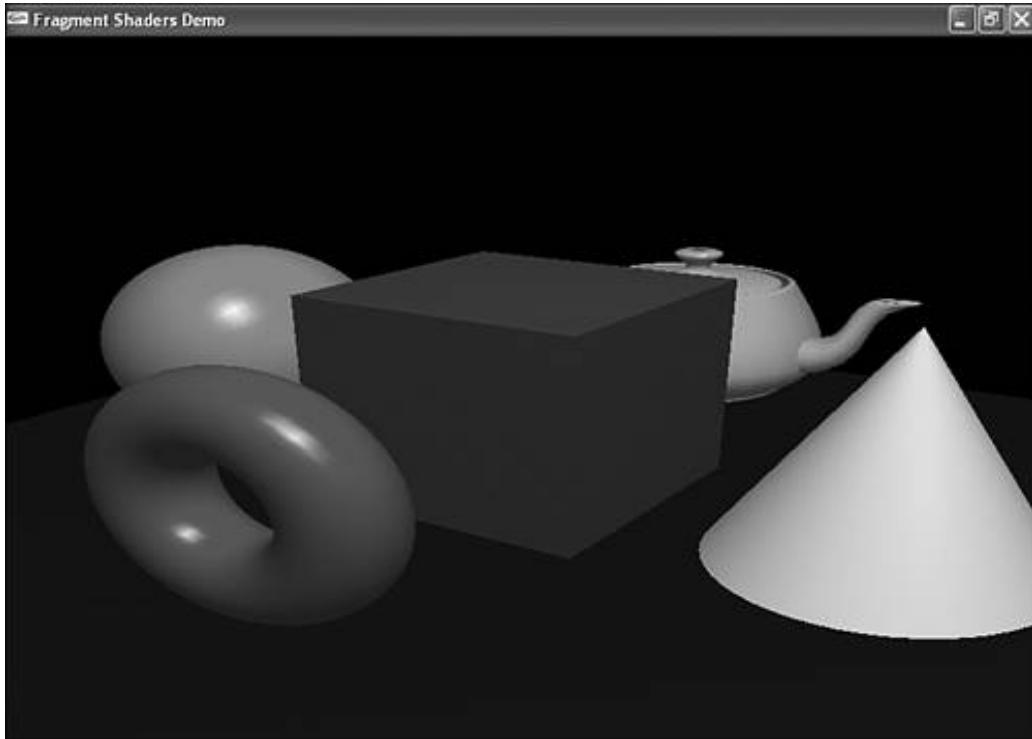
```
// grayscale.fs
//
// convert RGB to grayscale
void main(void)
```

```

{
    // Convert to grayscale
    float gray = dot(gl_Color.rgb, vec3(0.3, 0.59, 0.11));
    // replicate grayscale to RGB components
    gl_FragColor = vec4(gray, gray, gray, 1.0);
}

```

Figure 23.1. This fragment shader converts the RGB color into a single grayscale value.



Listing 23.2. Grayscale Conversion Low-Level Fragment Shader

```

!!ARBfp1.0
# grayscale.fp
#
# convert RGB to grayscale
ATTRIB iPrC = fragment.color.primary; # input primary color
OUTPUT oPrC = result.color;           # output color
DP3 oPrC.rgb, iPrC, {0.3, 0.59, 0.11};# R,G,B each contribute diff.
MOV oPrC.a, 1.0;                     # init alpha to 1
END

```

The key to all these fragment shaders is that what you write to the color output (`gl_FragColor` or `result.color`) is what is passed along down the rest of the OpenGL pipeline, eventually to the framebuffer. The primary color inputs are `gl_Color` and `fragment.color.primary`, respectively.

Try playing with the contributions of each color channel. Notice how they add up to 1. You can simulate overexposure by making them add up to more than 1, while less than 1 will simulate underexposure.

Sepia Tone

In this next example, we recolorize the grayscale picture with a sepia tone. This tone gives the picture the tint of an old Western photograph. To do this, we first convert to grayscale as before.

Then we multiply the gray value by a color vector, which accentuates some color channels and reduces others. [Listings 23.3](#) and [23.4](#) illustrate this sepia-tone conversion.

Listing 23.3. Sepia-Tone Conversion High-Level Fragment Shader

```
// sepia.fs
//
// convert RGB to sepia tone
void main(void)
{
    // Convert RGB to grayscale
    float gray = dot(gl_Color.rgb, vec3(0.3, 0.59, 0.11));
    // convert grayscale to sepia
    gl_FragColor = vec4(gray * vec3(1.2, 1.0, 0.8), 1.0);
}
```

Listing 23.4. Sepia-Tone Conversion Low-Level Fragment Shader

```
!!ARBfp1.0
# sepia.fp
#
# convert RGB to sepia tone
ATTRIB iPrC = fragment.color.primary; # input primary color
OUTPUT oPrC = result.color;           # output color
TEMP gray;
DP3 gray, iPrC, {0.3, 0.59, 0.11};    # convert to grayscale
MUL oPrC.rgb, gray, {1.2, 1.0, 0.8};  # convert to sepia
MOV oPrC.a, 1.0;                      # init alpha to 1
END
```

You can choose to colorize with any tint you like. Go ahead and play with the tint factors. Here, we've hard-coded one for sepia. If you're truly ambitious, you could substitute external application-defined constants (uniforms in GLSL, program parameters in [ARB_fragment_program](#)) to make the tint color user-selectable so you don't have to write a different shader for every tint color.

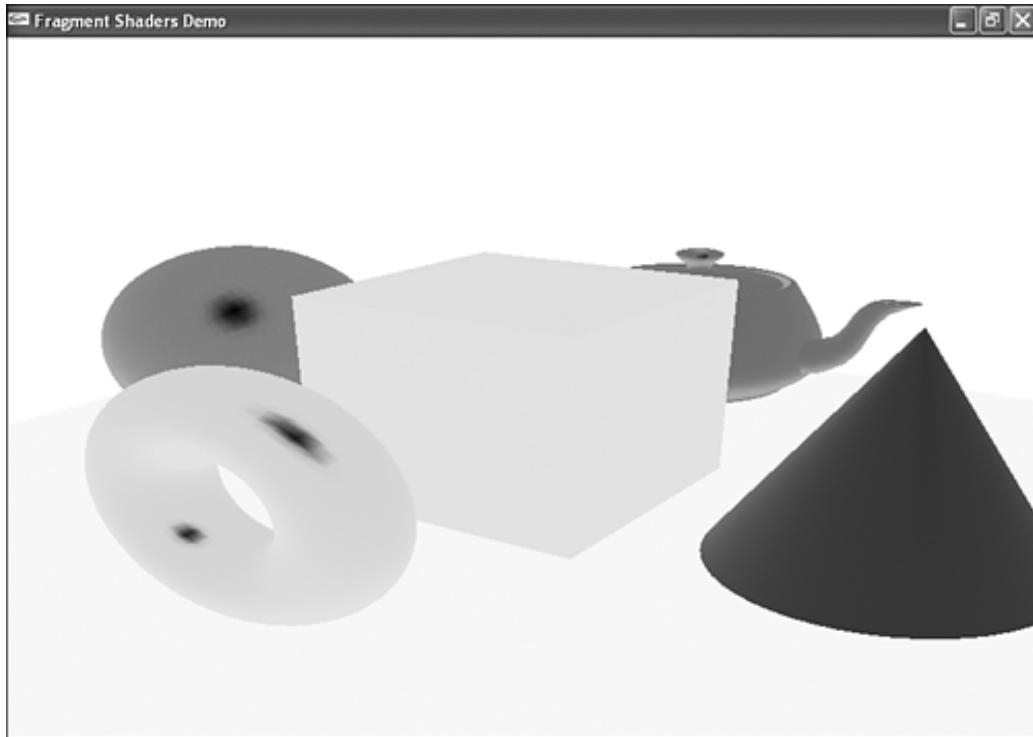
Inversion

For this next example, we're going for the film negative effect. These shaders are almost too simple to mention. All you have to do is take whatever color you were otherwise going to draw and subtract that color from 1. Black becomes white, and white becomes black. Red becomes cyan. Purple becomes chartreuse. You get the picture.

[Figure 23.2](#) illustrates the color inversion performed in [Listings 23.5](#) and [23.6](#). Use your imagination or consult the sample code for the grayscale inversion, which is just as straightforward.

Listing 23.5. Color Inversion High-Level Fragment Shader

```
// colorinvert.fs
//
// invert like a color negative
void main(void)
{
    // invert color components
    gl_FragColor.rgb = 1.0 - gl_Color.rgb;
    gl_FragColor.a = 1.0;
}
```

Figure 23.2. This fragment shader inverts the RGB color, yielding a film negative effect.**Listing 23.6. Color Inversion Low-Level Fragment Shader**

```
!!ARBfp1.0
# colorinvert.fp
#
# invert like a color negative
ATTRIB iPrC = fragment.color.primary; # input primary color
OUTPUT oPrC = result.color;           # output color
SUB oPrC.rgb, 1.0, iPrC;             # invert RGB colors
MOV oPrC.a, 1.0;                   # init alpha to 1
END
```

Heat Signature

Now, we attempt our first texture lookup. In this sample shader, we simulate a heat signature effect like the one in the movie *Predator*. Heat is represented by a color spectrum ranging from black to blue to green to yellow to red.

We again use the grayscale conversion, this time as our scalar heat value. We use this value as a texture coordinate to index into a 1D texture populated with the color gradients from black to red. [Figure 23.3](#) shows the results of the heat signature shaders in [Listings 23.7](#) and [23.8](#).

Listing 23.7. Heat Signature High-Level Fragment Shader

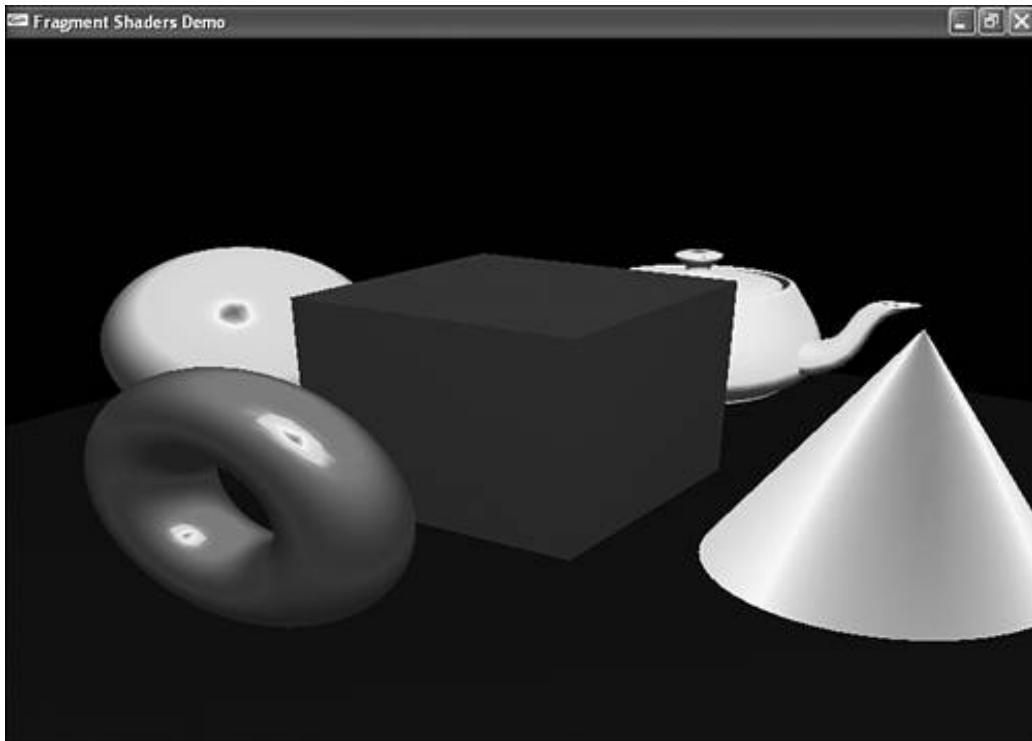
```
// heatsig.fs
//
// map grayscale to heat signature
uniform sampler1D sampler0;
void main(void)
{
    // Convert to grayscale
```

```

float gray = dot(gl_Color.rgb, vec3(0.3, 0.59, 0.11));
// lookup heatsig value
gl_FragColor = texture1D(sampler0, gray);
}

```

Figure 23.3. This fragment shader simulates a heat signature by looking up a color from a 1D texture.



Listing 23.8. Heat Signature Low-Level Fragment Shader

```

!!ARBfp1.0
# heatsig.fp
#
# map grayscale to heat signature
ATTRIB iPrC = fragment.color.primary; # input primary color
OUTPUT oPrC = result.color;           # output color
TEMP gray;
DP3 gray, iPrC, {0.3, 0.59, 0.11};   # R,G,B -> gray
TEX oPrC, gray, texture[0], 1D;       # lookup heatsig value
END

```

In the low-level shader, notice how we reference the texture unit from which we want to look up as `texture[0]`. The texture unit is effectively hard-coded into the shader. This is in contrast to the GLSL shader, which uses a special sampler uniform that can be set within the application.

Dependent Texture Lookups

Fixed functionality texture mapping was very strict, requiring all texture lookups to use an interpolated per-vertex texture coordinate. One of the powerful new capabilities made possible by fragment shaders is that you can calculate your own texture coordinates per-fragment. You can even use the result of one texture lookup as the coordinate for another lookup. All these cases are considered dependent texture lookups. They're named that because the lookups are dependent on other preceding operations in the fragment shader.

You may not have noticed, but we just performed a dependent texture lookup in the heat signature shader. First, we had to compute our texture coordinate by doing the grayscale conversion. Then we used that value as a texture coordinate to perform a dependent texture lookup into the 1D heat signature texture.

The dependency chain can continue: You could, for example, take the color from the heat texture and use that as a texture coordinate to perform a lookup from a cube map texture, perhaps to gamma-correct your color. Beware, however, that some OpenGL implementations have a hardware limit as to the length of dependency chains, so keep this point in mind if you want to avoid falling into a non-hardware-accelerated driver path!

Per-Fragment Fog

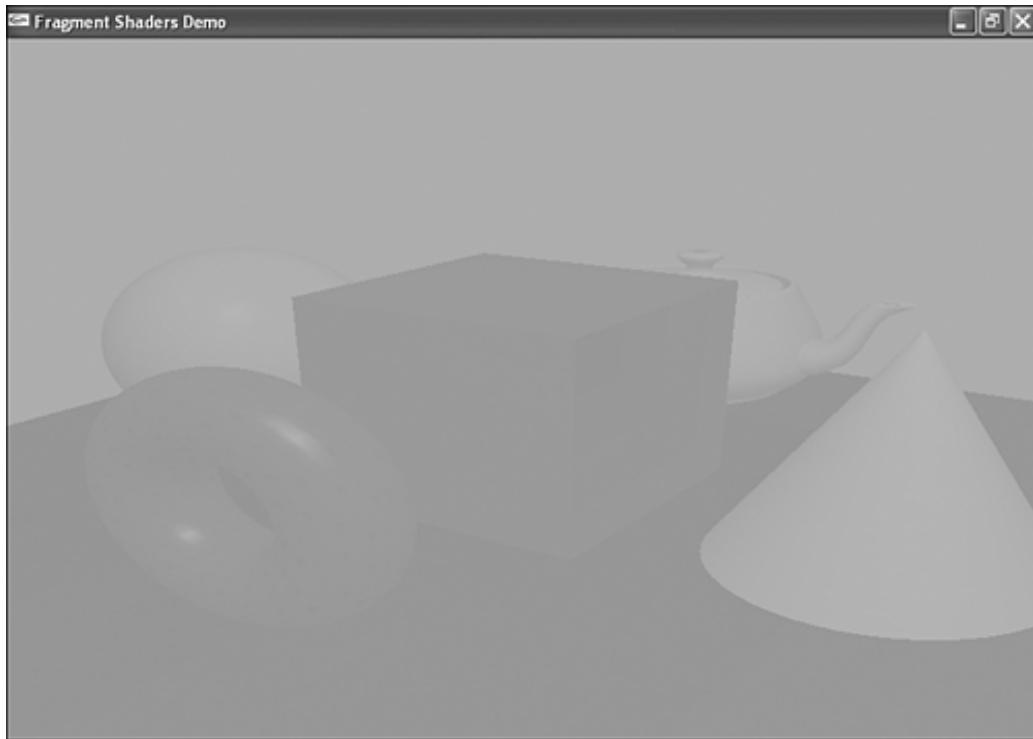
Instead of performing fog blending per-vertex, or calculating the fog factor per-vertex and using fixed functionality fog blending, we compute the fog factor and perform the blend ourselves within the fragment shader in the following example. This example emulates `GL_EXP2` fog mode except that it will be more accurate than most fixed functionality implementations, which apply the exponentiation per-vertex instead of per-fragment. This is most noticeable on low-tesselation geometry that extends from the foreground to the background, such as the floor upon which all the objects in the scene rest. Compare the results of this shader with the fog shaders in the preceding chapter, and you can readily see the difference.

[Figure 23.4](#) illustrates the output of the fog shader in [Listings 23.9](#) and [23.10](#).

Listing 23.9. Per-Fragment Fog High-Level Fragment Shader

```
// fog.fs
//
// per-pixel fog
uniform float density;
void main(void)
{
    const vec4 fogColor = vec4(0.5, 0.8, 0.5, 1.0);
    // calculate 2nd order exponential fog factor
    // based on fragment's Z distance
    const float e = 2.71828;
    float fogFactor = (density * gl_FragCoord.z);
    fogFactor *= fogFactor;
    fogFactor = clamp(pow(e, -fogFactor), 0.0, 1.0);
    // Blend fog color with incoming color
    gl_FragColor = mix(fogColor, gl_Color, fogFactor);
}
```

Figure 23.4. This fragment shader performs per-fragment fog computation.



Listing 23.10. Per-Fragment Fog Low-Level Fragment Shader

```
!!ARBfp1.0
# fog.fp
#
# per-pixel fog
ATTRIB iPrC = fragment.color.primary;    # input primary color
ATTRIB iFrP = fragment.position;          # input fragment position
OUTPUT oPrC = result.color;               # output color
PARAM density = program.local[0];         # fog density
PARAM fogColor = {0.5, 0.8, 0.5, 1.0};    # fog color
PARAM e = {2.71828, 0, 0, 0};
TEMP fogFactor;
# fogFactor = clamp(e^(-(d*Zw)^2))
MUL fogFactor.x, iFrP.z, density.x;
MUL fogFactor.x, fogFactor.x, fogFactor.x;
POW fogFactor.x, e.x, -fogFactor.x;
MAX fogFactor.x, fogFactor.x, 0.0;        # clamp to [0,1]
MIN fogFactor.x, fogFactor.x, 1.0;
LRP oPrC, fogFactor.x, iPrC, fogColor; # blend lit and fog colors
END
```

We need to comment on a few things here. One is the instructions used to blend. On the one hand, we have GLSL's built-in `mix` function. On the other hand, we have the `LRP` instruction in the low-level shader, which is specific to `ARB_fragment_program`. `LRP` is not available in `ARB_vertex_program`, where you have to perform blending with a combination of instructions, such as `SUB` and `MAD`.

Another thing to notice is how we have chosen to make the density an externally set constant rather than a hard-coded one. This way, we can tie the density to keystrokes. When the user hits the left or right arrows, we update the density shader constant with a new value without having to change the shader text at all. As a general rule, constant values that you may want to change at some point should not be hard-coded, but all others should be. By hard-coding a value, you give the OpenGL implementation's optimizing compiler an early opportunity to use this information to possibly make your shader run even faster.

Now that we've already created fog the hard way, we can use a low-level `ARB_fragment_program` extension shortcut that actually lets us request fog with a single line:

```
!!ARBfp1.0
OPTION ARB_fog_exp2;
...
```

Also available are `ARB_fog_exp` and `ARB_fog_linear`. These shortcuts were made available to ease the transition for application developers who were accustomed to the convenience of just setting `glEnable(GL_FOG)`.

Image Processing

Image processing is another application of fragment shaders that doesn't depend on vertex shader assistance. After drawing the scene without fragment shaders, we can apply convolution kernels to post-process the image in a variety of ways.

To keep the shaders concise and improve the probability of their being hardware-accelerated on a wider range of hardware, we've limited the kernel size to 3x3. Feel free to experiment with larger kernel sizes.

Within the sample application, `glCopyTexImage2D` is called to copy the contents of the framebuffer into a texture. The texture size is chosen to be the largest power-of-2 size smaller than the window. A fragment-shaded quad is then drawn centered within the window with the same dimensions as the texture, with a base texture coordinate ranging from (0,0) in the lower left to (1,1) in the upper right.

The fragment shader takes its base texture coordinate and performs a texture lookup to obtain the center sample of the 3x3 kernel neighborhood. It then proceeds to apply eight different offsets to lookup samples for the other eight spots in the neighborhood. Finally, the shader applies some filter to the neighborhood to yield a new color for the center of the neighborhood. Each sample shader provides a different filter commonly used for image-processing tasks.

Blur

Blurring may be the most commonly applied filter in everyday use. It smoothes out high-frequency features, such as the jaggies along object edges. It is also called a low-pass filter because it lets low-frequency features pass through while filtering out high-frequency features.

Because we're using only a 3x3 kernel, the blur is not overly dramatic in a single pass. We could make it more blurry by using a larger kernel or, as we do here, by applying the blur filter multiple times in successive passes. [Figure 23.5](#) shows the results of the blur filter in [Listings 23.11](#) and [23.12](#) after five passes.

Listing 23.11. Post-Process Blur High-Level Fragment Shader

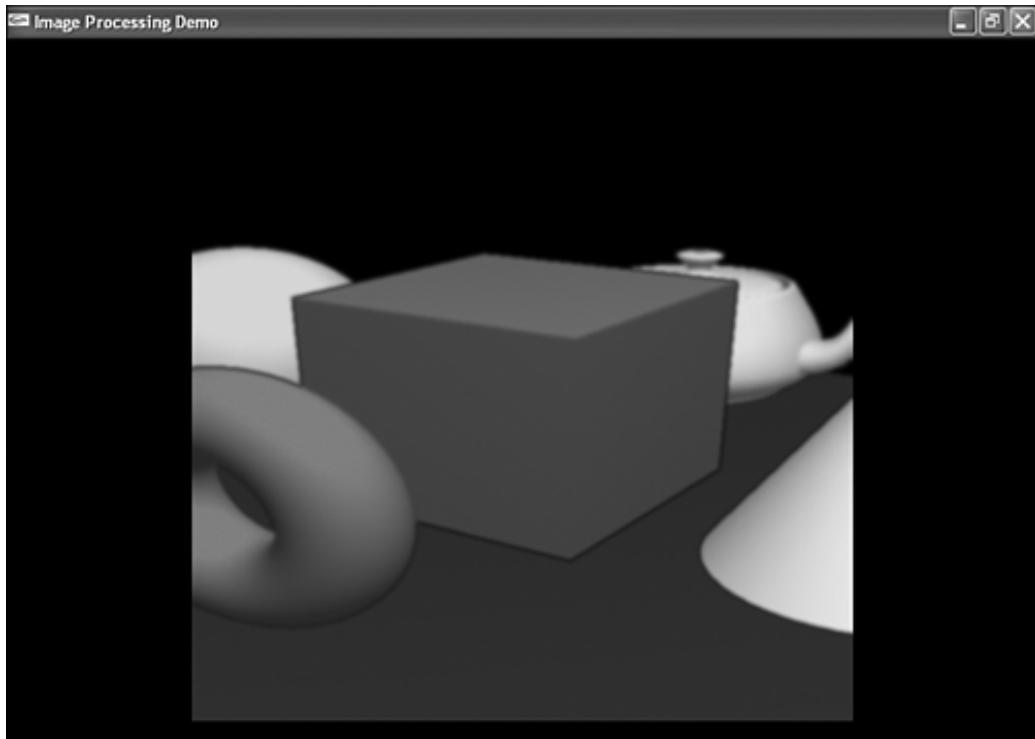
```
// blur.fs
//
// blur (low-pass) 3x3 kernel
uniform sampler2D sampler0;
uniform vec2 tc_offset[9];
void main(void)
{
    vec4 sample[9];
    for (int i = 0; i < 9; i++)
    {
        sample[i] = texture2D(sampler0,
```

```

        gl_TexCoord[0].st + tc_offset[i]);
}
// 1 2 1
// 2 1 2 / 13
// 1 2 1
gl_FragColor = (sample[0] + (2.0*sample[1]) + sample[2] +
                 (2.0*sample[3]) + sample[4] + (2.0*sample[5]) +
                 sample[6] + (2.0*sample[7]) + sample[8]) / 13.0;
}

```

Figure 23.5. This fragment shader blurs the scene.



Listing 23.12. Post-Process Blur Low-Level Fragment Shader

```

!!ARBfp1.0
# blur.fp
#
# blur (low-pass) 3x3 kernel
ATTRIB iTC0 = fragment.texcoord[0]; # input texcoord
OUTPUT oPrC = result.color; # output color
TEMP tc0, tc1, tc2, tc3, tc4, tc5, tc6, tc7, tc8;
ADD tc0, iTC0, program.local[0];
ADD tc1, iTC0, program.local[1];
ADD tc2, iTC0, program.local[2];
ADD tc3, iTC0, program.local[3];
ADD tc4, iTC0, program.local[4];
ADD tc5, iTC0, program.local[5];
ADD tc6, iTC0, program.local[6];
ADD tc7, iTC0, program.local[7];
ADD tc8, iTC0, program.local[8];
TEX tc0, tc0, texture[0], 2D;
TEX tc1, tc1, texture[0], 2D;
TEX tc2, tc2, texture[0], 2D;
TEX tc3, tc3, texture[0], 2D;
TEX tc4, tc4, texture[0], 2D;
TEX tc5, tc5, texture[0], 2D;

```

```

TEX tc6, tc6, texture[0], 2D;
TEX tc7, tc7, texture[0], 2D;
TEX tc8, tc8, texture[0], 2D;
# 1 2 1
# 2 1 2 / 13
# 1 2 1
ADD tc0, tc0, tc2;
ADD tc2, tc4, tc6;
ADD tc0, tc0, tc2;
ADD tc0, tc0, tc8;
ADD tc1, tc1, tc3;
ADD tc3, tc5, tc7;
ADD tc1, tc1, tc3;
MAD tc0, tc1, 2.0, tc0;
MUL oPrC, tc0, 0.076923; # 1/13
END

```

The first thing we do in the blur shaders is generate our nine texture coordinates. This is accomplished by adding precomputed constant offsets to the interpolated base texture coordinate. The offsets were computed taking into account the size of the texture such that the neighboring texels to the north, south, east, west, northeast, southeast, northwest, and southwest could be obtained by a simple 2D texture lookup. In the low-level version, `texture[0]` indicates that the texture lookup takes place on texture unit 0. In the GLSL version, you have to use a special-purpose uniform called a *sampler*, just as we did in the heat sampler. The sampler is loaded outside the shader to reflect which texture unit is in play.

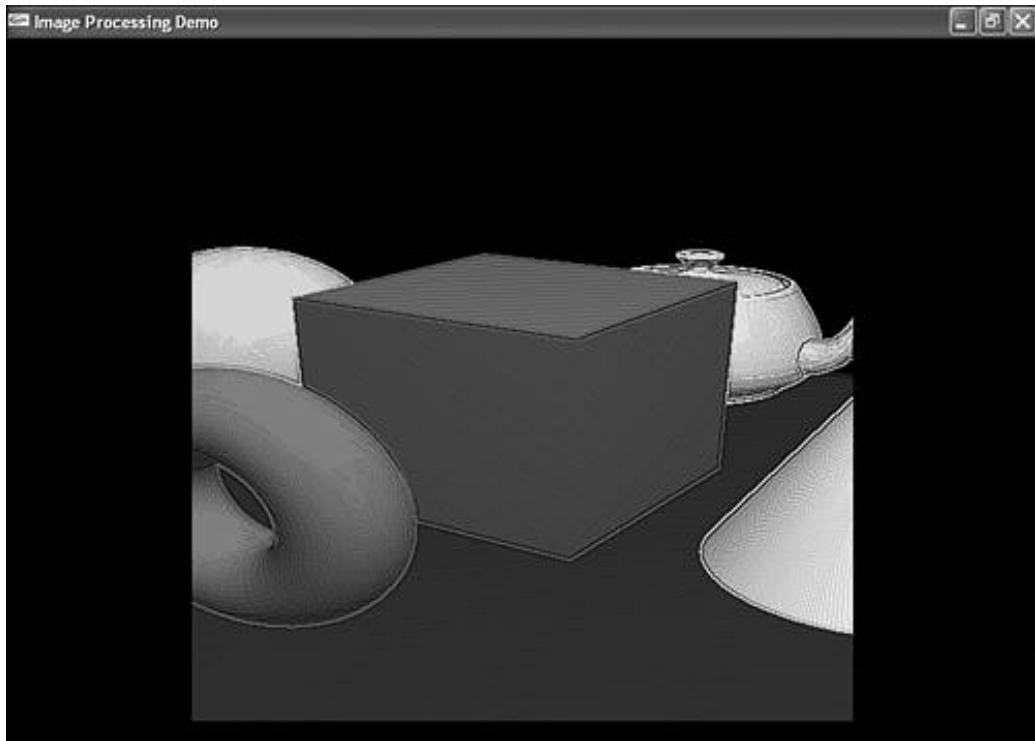
This neighborhood is obtained the same way in all our image processing shaders. It is the filter applied to the neighborhood that differs in each shader. In the case of the blur filter, the texel neighborhood is multiplied by a 3x3 kernel of coefficients (1s and 2s), which add up to 13. The resulting values are all summed and averaged by dividing by 13, resulting in the new color for the texel. Note that we could have made the kernel coefficient values 1/13 and 2/13 instead of 1 and 2, but that would have required many extra multiplies. It is simpler and cheaper for us to factor out the 1/13 and just apply it at the end.

Try experimenting with the filter coefficients. What if, for example, you put a weight of 1 at each corner and then divide by 4? Notice what happens when you divide by more or less than the sum of the coefficients: The scene grows darker or lighter. That makes sense. If your scene were all white, you would be effectively multiplying the filter coefficients by 1 and adding them up. If you don't divide by the sum of the coefficients, you'll end up with a color other than white.

Sharpen

Sharpening is the opposite of blurring. Some examples of its use include making edges more pronounced and making text more readable. [Figure 23.6](#) illustrates the use of sharpening, applying the filter in two passes.

Figure 23.6. This fragment shader sharpens the scene.



Here is the GLSL code for applying the sharpen filter:

```
// sharpen.fs
//
// 3x3 sharpen kernel
uniform sampler2D sampler0;
uniform vec2 tc_offset[9];
void main(void)
{
    vec4 sample[9];
    for (int i = 0; i < 9; i++)
    {
        sample[i] = texture2D(sampler0,
                               gl_TexCoord[0].st + tc_offset[i]);
    }
    // -1 -1 -1
    // -1  9 -1
    // -1 -1 -1
    gl_FragColor = (sample[4] * 9.0) -
                   (sample[0] + sample[1] + sample[2] +
                    sample[3] + sample[5] +
                    sample[6] + sample[7] + sample[8]);
}
```

All our image processing low-level shaders look the same up to and including the part where they perform the texture lookups, so we'll concentrate on the parts that are different: the application of the convolution kernel. The low-level code for sharpening is as follows:

```
!!ARBfp1.0
# sharpen.fp
#
# 3x3 sharpen kernel
...
# -1 -1 -1
# -1  9 -1
# -1 -1 -1
```

```
ADD tc0, -tc0, -tc1;
ADD tc0, tc0, -tc2;
ADD tc0, tc0, -tc3;
ADD tc0, tc0, -tc5;
ADD tc0, tc0, -tc6;
ADD tc0, tc0, -tc7;
ADD tc0, tc0, -tc8;
MAD oPrC, tc4, 9.0, tc0;
END
```

Notice how this kernel also sums to 1, as did the blur filter. This operation guarantees that, on average, the filter is not increasing or decreasing the brightness. It's just sharpening the brightness, as desired.

Dilation and Erosion

Dilation and erosion are morphological filters, meaning they alter the shape of objects. Dilation grows the size of bright objects, whereas erosion shrinks the size of bright objects. (They each have the reverse effect on dark objects.) [Figures 23.7](#) and [23.8](#) show the effects of three passes of dilation and erosion, respectively.

Figure 23.7. This fragment shader dilates objects.

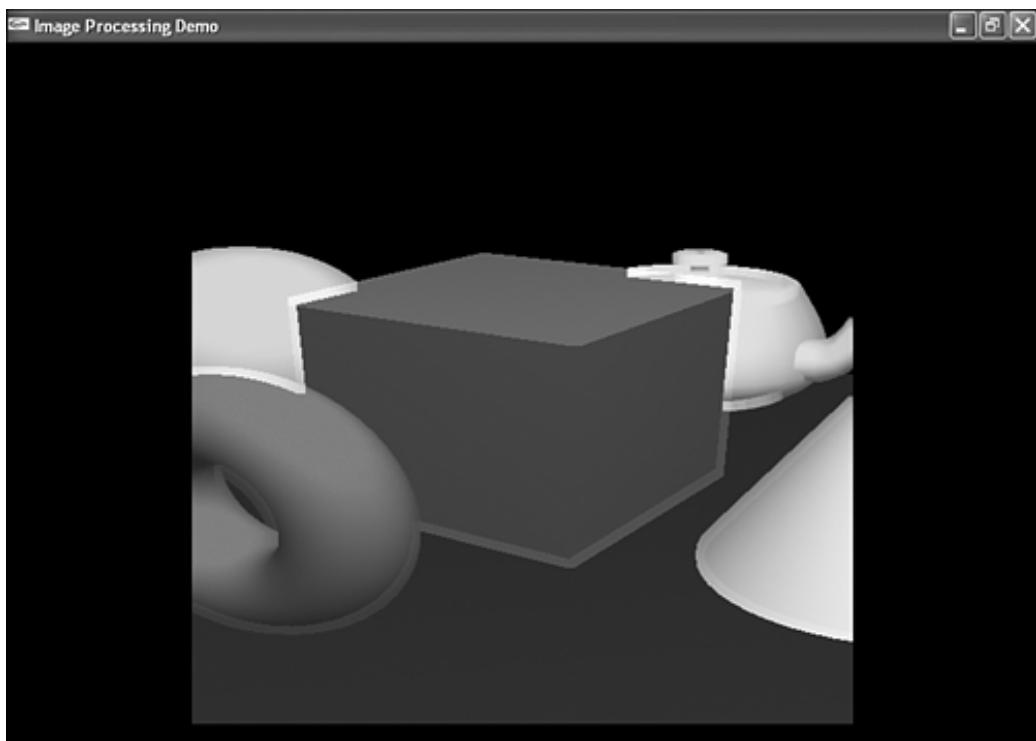
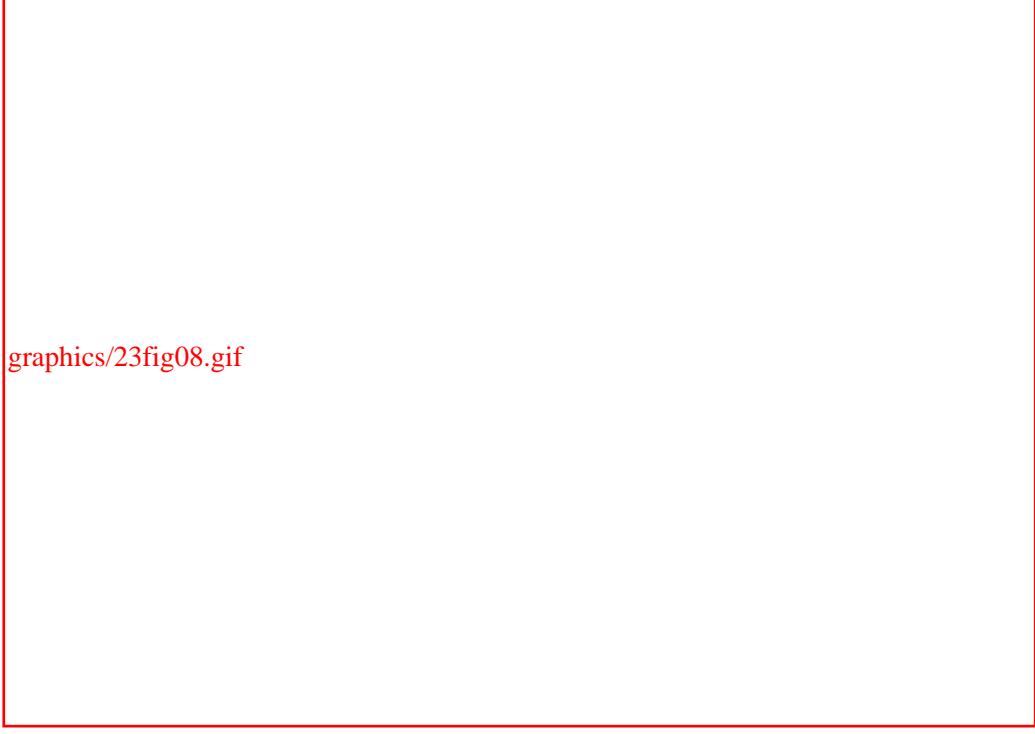


Figure 23.8. This fragment shader erodes objects.



graphics/23fig08.gif

Dilation simply finds the maximum value in the neighborhood:

```
// dilation.fs
//
// maximum of 3x3 kernel
uniform sampler2D sampler0;
uniform vec2 tc_offset[9];
void main(void)
{
    vec4 sample[9];
    vec4 maxValue = vec4(0.0);
    for (int i = 0; i < 9; i++)
    {
        sample[i] = texture2D(sampler0,
                               gl_TexCoord[0].st + tc_offset[i]);
        maxValue = max(sample[i], maxValue);
    }
    gl_FragColor = maxValue;
}
!!ARBfp1.0
# dilation.fp
#
# maximum of 3x3 kernel
...
MAX tc0, tc0, tc1;
MAX tc0, tc0, tc2;
MAX tc0, tc0, tc3;
MAX tc0, tc0, tc4;
MAX tc0, tc0, tc5;
MAX tc0, tc0, tc6;
MAX tc0, tc0, tc7;
MAX oPrC, tc0, tc8;
END
```

Erosion conversely finds the minimum value in the neighborhood:

```
// erosion.fs
```

```

//  

// minimum of 3x3 kernel  

uniform sampler2D sampler0;  

uniform vec2 tc_offset[9];  

void main(void)  

{  

    vec4 sample[9];  

    vec4 minValue = vec4(1.0);  

    for (int i = 0; i < 9; i++)  

    {  

        sample[i] = texture2D(sampler0,  

                               gl_TexCoord[0].st + tc_offset[i]);  

        minValue = min(sample[i], minValue);  

    }  

    gl_FragColor = minValue;  

}  

!!ARBfp1.0  

# erosion.fp  

#  

# minimum of 3x3 kernel  

...  

MIN tc0, tc0, tc1;  

MIN tc0, tc0, tc2;  

MIN tc0, tc0, tc3;  

MIN tc0, tc0, tc4;  

MIN tc0, tc0, tc5;  

MIN tc0, tc0, tc6;  

MIN tc0, tc0, tc7;  

MIN oPrC, tc0, tc8;  

END

```

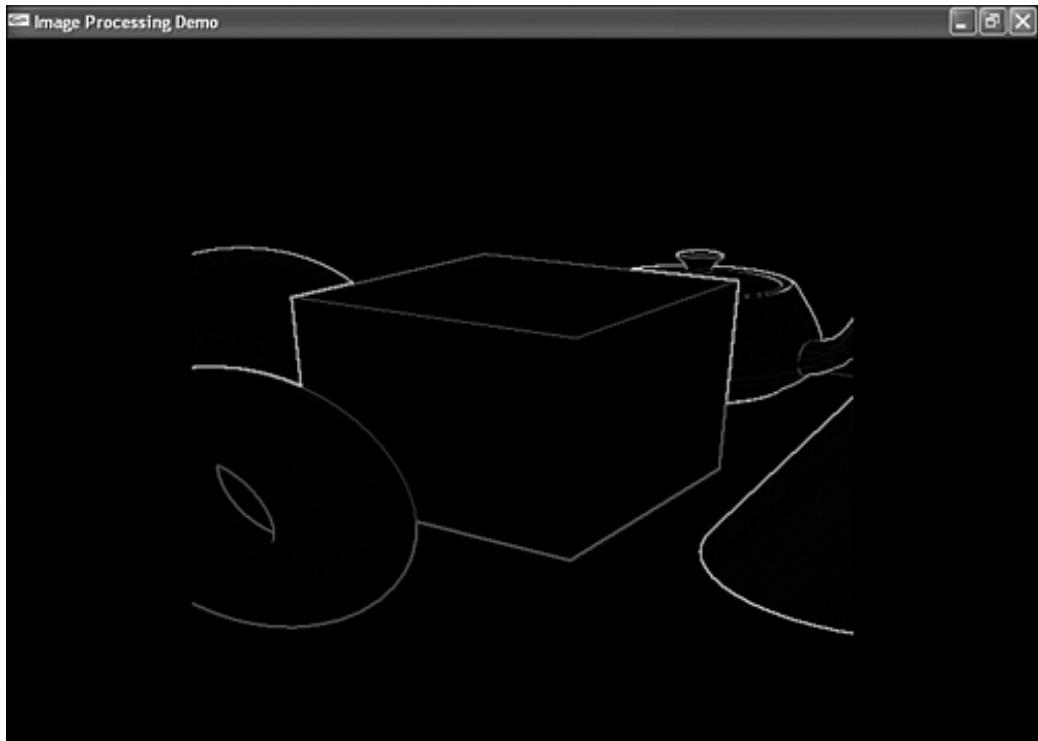
Edge Detection

One last filter class worthy of mention here is edge detectors. They do just what you would expect—detect edges. Edges are simply places in an image where the color changes rapidly, and edge detection filters pick up on these rapid changes and highlight them.

Three widely used edge detectors are Laplacian, Sobel, and Prewitt. Sobel and Prewitt are gradient filters that detect changes in the first derivative of each color channel's intensity, but only in a single direction. Laplacian, on the other hand, detects zero-crossings of the second derivative, where the intensity gradient suddenly changes from getting darker to getting lighter, or vice versa. It works for edges of any orientation.

Because the differences in their results are subtle, [Figure 23.9](#) shows the results from only one of them, the Laplacian filter. Try out the others and examine their shaders at your leisure in the accompanying sample code.

Figure 23.9. This fragment shader implements Laplacian edge detection.



The Laplacian's filter code is almost identical to the sharpen code we just looked at:

```

// laplacian.fs
//
// Laplacian edge detection
uniform sampler2D sampler0;
uniform vec2 tc_offset[9];
void main(void)
{
    vec4 sample[9];
    for (int i = 0; i < 9; i++)
    {
        sample[i] = texture2D(sampler0,
                               gl_TexCoord[0].st + tc_offset[i]);
    }
    // -1 -1 -1
    // -1  8 -1
    // -1 -1 -1
    gl_FragColor = (sample[4] * 8.0) -
                   (sample[0] + sample[1] + sample[2] +
                    sample[3] + sample[5] +
                    sample[6] + sample[7] + sample[8]);
}
!!ARBfp1.0
# laplacian.fp
#
# Laplacian edge detection
...
# -1 -1 -1
# -1  8 -1
# -1 -1 -1
ADD tc0, -tc0, -tc1;
ADD tc0, tc0, -tc2;
ADD tc0, tc0, -tc3;
ADD tc0, tc0, -tc5;
ADD tc0, tc0, -tc6;
ADD tc0, tc0, -tc7;

```

```

ADD tc0, tc0, -tc8;
MAD oPrC, tc4, 8.0, tc0;
END

```

The difference, of course, is that the center kernel value is 8 rather than the 9 present in the sharpen kernel. The coefficients sum up to 0 rather than 1. This explains the blackness of the image. Instead of, on average, retaining its original brightness, the edge detection kernel will produce 0 in areas of the image with no color change.

Lighting

Welcome back to another discussion of lighting shaders. In the preceding chapter, we covered per-vertex lighting. We also described a couple of per-fragment fixed functionality tricks to improve the per-vertex results: separate specular with color sum and power function texture for specular exponent. In this chapter, we perform all our lighting calculations in the fragment shader to obtain the greatest accuracy.

The shaders here will look very familiar. The same lighting equations are implemented, so the code is virtually identical. One new thing is the use of vertex shaders and fragment shaders together. The vertex shader sets up the data that needs to be interpolated across the line or triangle, such as normals and light vectors. The fragment shader then proceeds to do most of the work, resulting in a final color.

Diffuse Lighting

As a refresher, the equation for diffuse lighting follows:

$$C_{\text{diff}} = \max\{N \cdot L, 0\} * C_{\text{mat}} * C_{\text{li}}$$

You need a vertex shader that generates both normal and light vectors. [Listings 23.13](#) and [23.14](#) contain the high-level and low-level vertex shader source to generate these necessary interpolants for diffuse lighting.

Listing 23.13. Diffuse Lighting Interpolant Generating High-Level Vertex Shader

```

// diffuse.vs
//
// setup interpolants for diffuse lighting
uniform vec3 lightPos0;
varying vec3 N, L;
void main(void)
{
    // vertex MVP transform
    // eye-space normal
    N = gl_NormalMatrix * gl_Normal;
    // eye-space light vector
    vec4 V = gl_ModelViewMatrix * gl_Vertex;
    L = lightPos0 - V.xyz;
    // Copy the primary color
    gl_FrontColor = gl_Color;
}

```

Listing 23.14. Diffuse Lighting Interpolant Generating Low-Level Vertex Shader

```

!ARBvp1.0
# diffuse.vp
#
# setup interpolants for diffuse lighting

```

```

ATTRIB iPos = vertex.position;      # input position
ATTRIB iPrC = vertex.color.primary; # input primary color
ATTRIB iNrm = vertex.normal;        # input normal
OUTPUT oPos = result.position;     # output position
OUTPUT oPrC = result.color.primary; # output primary color
OUTPUT oTC0 = result.texcoord[0];   # output texcoord 0
OUTPUT oTC1 = result.texcoord[1];   # output texcoord 1
PARAM mvp[4] = { state.matrix.mvp }; # model-view * projection matrix
PARAM mv[4] = { state.matrix.modelview }; # model-view matrix
# inverse transpose of model-view matrix:
PARAM mvIT[4] = { state.matrix.modelview.invtrans };
PARAM lightPos = program.local[0];   # light pos in eye space
TEMP V;
DP4 oPos.x, iPos, mvp[0];          # temporary register
DP4 oPos.y, iPos, mvp[1];          # xform input pos by MVP
DP4 oPos.z, iPos, mvp[2];
DP4 oPos.w, iPos, mvp[3];
DP4 V.x, iPos, mv[0];              # xform input pos by MV
DP4 V.y, iPos, mv[1];
DP4 V.z, iPos, mv[2];
DP4 V.w, iPos, mv[3];
DP3 oTC0.x, iNrm, mvIT[0];        # xform norm to eye space
DP3 oTC0.y, iNrm, mvIT[1];
DP3 oTC0.z, iNrm, mvIT[2];
SUB oTC1, lightPos, V;             # put N in texcoord 0
MOV oPrC, iPrC;                  # light vector in texcoord 1
# copy primary color in to out
END

```

When using low-level shaders, we're stuck tossing the normal and light vector into a standard texture coordinate for interpolation. However, notice how we are able to give descriptive names `N` and `L` to our interpolants, known as *varyings* in GLSL. They have to match the names used in the fragment shader. All in all, this feature makes the high-level shaders much more readable and less error-prone. For example, if we're not careful in the low-level shaders, we might accidentally output `L` into texture coordinate 0, whereas the fragment shader is expecting it in texture coordinate 1. No compile error would be thrown. GLSL, on the other hand, matches them up automatically by name, keeping us out of trouble and at the same time avoiding the need for tedious comments in code explaining the contents of each interpolant.

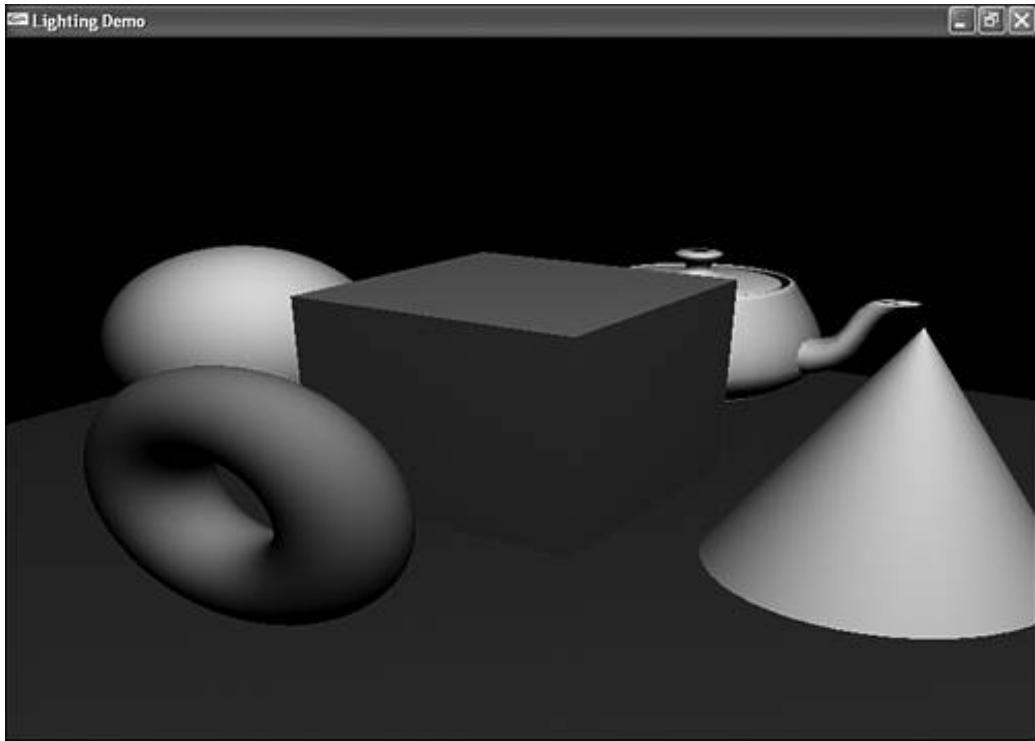
The diffuse lighting fragment shaders resulting in [Figure 23.10](#) follow in [Listings 23.15](#) and [23.16](#). Unlike colors produced by specular lighting, diffuse lit colors do not change rapidly across a line or triangle, so you will probably not be able to distinguish between per-vertex and per-fragment diffuse lighting. For this reason, in general, it would be more efficient to perform diffuse lighting in the vertex shader, as we did in the preceding chapter. We perform it here per-fragment simply as a learning exercise.

Listing 23.15. Diffuse Lighting High-Level Fragment Shader

```

// diffuse.fs
//
// per-pixel diffuse lighting
varying vec3 N, L;
void main(void)
{
    // output the diffuse color
    float intensity = max(0.0,
        dot(normalize(N), normalize(L)));
    gl_FragColor = gl_Color;
    gl_FragColor.rgb *= intensity;
}

```

Figure 23.10. Per-fragment diffuse lighting.**Listing 23.16. Diffuse Lighting Low-Level Fragment Shader**

```
!!ARBfp1.0
# diffuse.fp
#
# per-pixel diffuse lighting
ATTRIB iPrC = fragment.color.primary; # input primary color
ATTRIB iTC0 = fragment.texcoord[0]; # normal (N)
ATTRIB iTC1 = fragment.texcoord[1]; # light vector (L)
OUTPUT oPrC = result.color; # output color
TEMP N, L, NdotL;
DP3 N.w, iTC0, iTC0; # normalize normal
RSQ N.w, N.w;
MUL N, iTC0, N.w;
DP3 L.w, iTC1, iTC1; # normalize light vec
RSQ L.w, L.w;
MUL L, iTC1, L.w;
DP3 NdotL, N, L; # N . L
MAX NdotL, NdotL, 0.0; # max(N . L, 0)
MUL oPrC.rgb, iPrC, NdotL; # diffuse color
MOV oPrC.a, iPrC.a; # preserve alpha
END
```

First, we normalize the interpolated normal and light vectors. Then one more dot product, a maximum, and a multiply, and we're finished. Because we want a white light, we can save ourselves the additional multiply by $C_{li} = \{1, 1, 1, 1\}$.

Multiple Specular Lights

Rather than cover specular lighting and multiple light samples independently, we'll cover both at the same time. As a refresher, the specular lighting equation is

$$C_{\text{spec}} = \max\{N \cdot H, 0\}^{\alpha} S_{\text{exp}} \cdot C_{\text{mat}} \cdot C_{\text{li}}$$

The vertex shaders need to generate light vector interpolants for all three lights, in addition to the normal vector. We'll calculate the half-angle vector in the fragment shader. [Listings 23.17](#) and [23.18](#) show the vertex shaders for the three diffuse and specular lights.

Listing 23.17. Three Lights High-Level Vertex Shader

```
// 3lights.vs
//
// setup interpolants for 3 specular lights
uniform vec3 lightPos0;
uniform vec3 lightPos1;
uniform vec3 lightPos2;
varying vec3 N, L[3];
void main(void)
{
    // vertex MVP transform
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    vec4 V = gl_ModelViewMatrix * gl_Vertex;
    // eye-space normal
    N = gl_NormalMatrix * gl_Normal;
    // Light vectors
    L[0] = lightPos0 - V.xyz;
    L[1] = lightPos1 - V.xyz;
    L[2] = lightPos2 - V.xyz;
    // Copy the primary color
    gl_FrontColor = gl_Color;
}
```

Listing 23.18. Three Lights Low-Level Vertex Shader

```
!ARBvp1.0
# 3lights.vp
#
# setup interpolants for 3 specular lights
ATTRIB iPos = vertex.position;      # input position
ATTRIB iPrC = vertex.color.primary; # input primary color
ATTRIB iNrm = vertex.normal;        # input normal
OUTPUT oPos = result.position;      # output position
OUTPUT oPrC = result.color.primary; # output primary color
OUTPUT oTC0 = result.texcoord[0];    # output texcoord 0
OUTPUT oTC1 = result.texcoord[1];    # output texcoord 1
OUTPUT oTC2 = result.texcoord[2];    # output texcoord 2
OUTPUT oTC3 = result.texcoord[3];    # output texcoord 3
PARAM mvp[4] = { state.matrix.mvp }; # model-view * projection matrix
PARAM mv[4] = { state.matrix.modelview }; # model-view matrix
# inverse transpose of model-view matrix:
PARAM mvIT[4] = { state.matrix.modelview.invtrans };
PARAM lightPos0 = program.local[0]; # light pos 0 in eye space
PARAM lightPos1 = program.local[1]; # light pos 1 in eye space
PARAM lightPos2 = program.local[2]; # light pos 2 in eye space
TEMP V;                            # temporary register
DP4 oPos.x, iPos, mvp[0];          # xform input pos by MVP
DP4 oPos.y, iPos, mvp[1];
DP4 oPos.z, iPos, mvp[2];
DP4 oPos.w, iPos, mvp[3];
DP4 V.x, iPos, mv[0];              # xform input pos by MV
DP4 V.y, iPos, mv[1];
```

```

DP4 V.z, iPos, mv[2];
DP4 V.w, iPos, mv[3];
DP3 oTC0.x, iNrm, mvIT[0];           # xform norm to eye space
DP3 oTC0.y, iNrm, mvIT[1];
DP3 oTC0.z, iNrm, mvIT[2];           # put N in texcoord 0
SUB oTC1, lightPos0, V;              # light vector 0 in texcoord 1
SUB oTC2, lightPos1, V;              # light vector 1 in texcoord 2
SUB oTC3, lightPos2, V;              # light vector 2 in texcoord 3
MOV oPrC, iPrC;                     # copy primary color in to out
END

```

The fragment shaders will be doing most of the heavy lifting. [Figure 23.11](#) shows the result of [Listings 23.19](#) and [23.20](#).

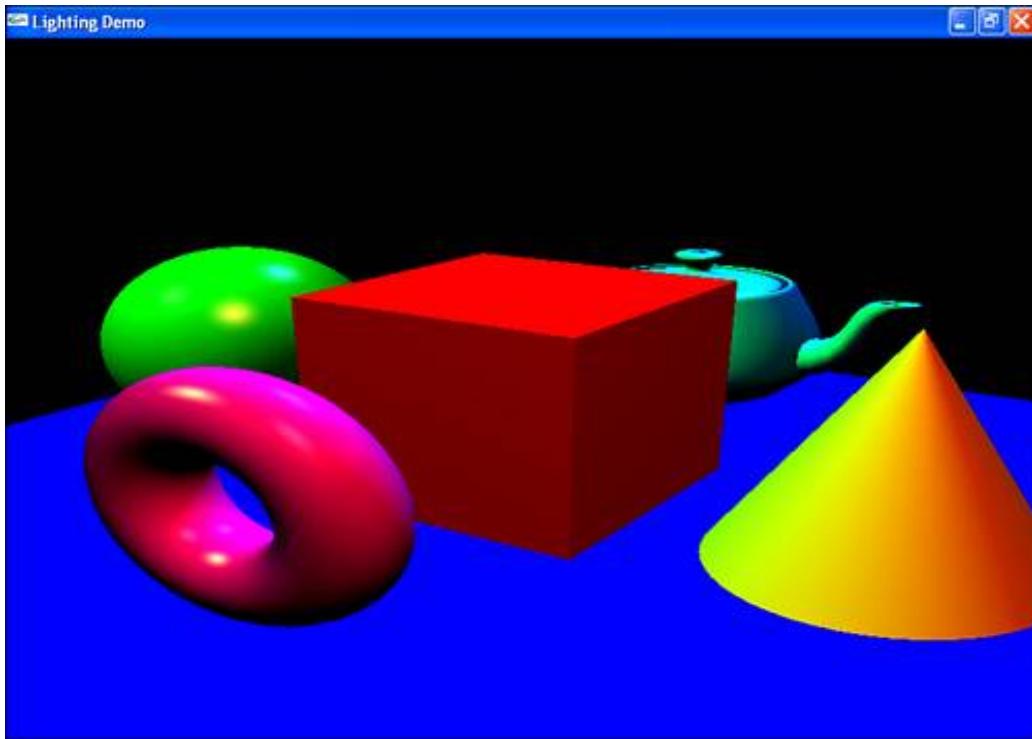
Listing 23.19. Three Diffuse and Specular Lights High-Level Fragment Shader

```

// 3lights.fs
//
// 3 specular lights
varying vec3 N, L[3];
void main(void)
{
    const float specularExp = 128.0;
    vec3 NN = normalize(N);
    // Light colors
    vec3 lightCol[3];
    lightCol[0] = vec3(1.0, 0.25, 0.25);
    lightCol[1] = vec3(0.25, 1.0, 0.25);
    lightCol[2] = vec3(0.25, 0.25, 1.0);
    gl_FragColor = vec4(0.0);
    for (int i = 0; i < 3; i++)
    {
        vec3 NL = normalize(L[i]);
        vec3 NH = normalize(NL + vec3(0.0, 0.0, 1.0));
        // Accumulate the diffuse contributions
        gl_FragColor.rgb += gl_Color.rgb * lightCol[i] *
            max(0.0, dot(NN, NL));
        // Accumulate the specular contributions
        gl_FragColor.rgb += lightCol[i] *
            pow(max(0.0, dot(NN, NH)), specularExp);
    }
    gl_FragColor.a = gl_Color.a;
}

```

Figure 23.11. Per-fragment diffuse and specular lighting with three lights.



Listing 23.20. Three Diffuse and Specular Lights Low-Level Fragment Shader

```
!!ARBfp1.0
# 3lights.fp
#
# 3 specular lights
ATTRIB iPrC = fragment.color.primary;# input primary color
ATTRIB iTC0 = fragment.texcoord[0]; # normal (N)
ATTRIB iTC1 = fragment.texcoord[1]; # light vector (L) 0
ATTRIB iTC2 = fragment.texcoord[2]; # light vector (L) 1
ATTRIB iTC3 = fragment.texcoord[3]; # light vector (L) 2
OUTPUT oPrC = result.color; # output color
PARAM lightCol0 = { 1.0, 0.25, 0.25, 1.0 } ; # light 0 color
PARAM lightCol1 = { 0.25, 1.0, 0.25, 1.0 } ; # light 1 color
PARAM lightCol2 = { 0.25, 0.25, 1.0, 1.0 } ; # light 2 color
TEMP N, L, H, NdotL, NdotH, finalColor;
ALIAS diffuse = NdotL;
ALIAS specular = NdotH;
DP3 N.w, iTC0, iTC0; # normalize normal
RSQ N.w, N.w;
MUL N, iTC0, N.w;
DP3 L.w, iTC1, iTC1; # normalize light vec 0
RSQ L.w, L.w;
MUL L, iTC1, L.w;
ADD H, L, {0, 0, 1}; # half-angle vector 0
DP3 H.w, H, H; # normalize it
RSQ H.w, H.w;
MUL H, H, H.w;
DP3 NdotL, N, L; # N . L0
MAX NdotL, NdotL, 0.0; # max(N . L, 0)
MUL diffuse, iPrC, NdotL; # diffuse color
MUL finalColor, diffuse, lightCol0;
DP3 NdotH, N, H; # N . H0
MAX NdotH, NdotH, 0.0; # max(N . H, 0)
POW specular, NdotH.x, 128.0.x; # NdotH^128
MAD finalColor, specular, lightCol0, finalColor;
DP3 L.w, iTC2, iTC2; # normalize light vec 1
```

```

RSQ L.w, L.w;
MUL L, iTC2, L.w;
ADD H, L, {0, 0, 1};           # half-angle vector 1
DP3 H.w, H, H;                # normalize it
RSQ H.w, H.w;
MUL H, H, H.w;
DP3 NdotL, N, L;              # N . L1
MAX NdotL, NdotL, 0.0;         # max(N . L, 0)
MUL diffuse, iPrC, NdotL;     # diffuse color
MAD finalColor, diffuse, lightCol1, finalColor;
DP3 NdotH, N, H;              # N . H1
MAX NdotH, NdotH, 0.0;         # max(N . H, 0)
POW specular, NdotH.x, 128.0.x; # NdotH^128
MAD finalColor, specular, lightCol1, finalColor;
DP3 L.w, iTC3, iTC3;          # normalize light vec 2
RSQ L.w, L.w;
MUL L, iTC3, L.w;
ADD H, L, {0, 0, 1};           # half-angle vector 2
DP3 H.w, H, H;                # normalize it
RSQ H.w, H.w;
MUL H, H, H.w;
DP3 NdotL, N, L;              # N . L2
MAX NdotL, NdotL, 0.0;         # max(N . L, 0)
MUL diffuse, iPrC, NdotL;     # diffuse color
MAD finalColor, diffuse, lightCol2, finalColor;
DP3 NdotH, N, H;              # N . H2
MAX NdotH, NdotH, 0.0;         # max(N . H, 0)
POW specular, NdotH.x, 128.0.x; # NdotH^128
MAD oPrC.rgb, specular, lightCol2, finalColor;
MOV oPrC.a, iPrC.a;           # preserve alpha
END

```

This time, we made each of the three lights a different color instead of white, necessitating an additional multiply by `lightColn` (C_{li}). The lack of loops really makes itself obvious here in the low-level shader, which is more than three times as long.

Procedural Texture Mapping

When can you texture map an object without using any textures? When you're using procedural texture maps. This technique enables you to apply colors or other surface properties to an object, just like using conventional texture maps. With conventional texture maps, you load a texture image into OpenGL with `glTexImage`; then you perform a texture lookup within your fragment shader. However, with procedural texture mapping, you skip the texture loading and texture lookup and instead describe algorithmically what the texture looks like.

Procedural texture mapping has advantages and disadvantages. One advantage is that its storage requirements are measured in terms of a few shader instructions rather than megabytes of texture cache and/or system memory consumed by conventional textures. This frees your storage for other uses, such as the vertex buffer objects discussed in [Chapter 16](#), "Buffer Objects: It's Your Video Memory; You Manage It!"

Another benefit is its virtually limitless resolution. Like vector drawings versus raster drawings, procedural textures scale to any size without loss of quality. Conventional textures require you to increase texture image sizes to improve quality when greatly magnified. Eventually, you'll hit a hardware limit. The only hardware limit affecting procedural texture quality is the floating-point precision of the shader processors, which are required to be at least 24-bit for OpenGL.

A disadvantage of procedural texture maps, and the reason they're not used more frequently, is that the complexity of the texture you want to represent requires an equally complex fragment

shader. Everything from simple shapes and colors all the way to complex plasma, fire, smoke, marble, or wood grain can be achieved with procedural textures, given enough shader instructions to work with. But sometimes you just want the company logo or a satellite map or someone's face textured onto your scene. Certainly, conventional textures will always serve a purpose!

Checkerboard Texture

Enough discussion. Let's warm up with our first procedural texture: a 3D checkerboard. Our object will appear to be cut out of a block of alternating white and black cubes. Sounds simple enough, right?

We'll use the object-space position at each fragment to decide what color to make that fragment. So we need a vertex shader that, in addition to transforming the object-space position to clip-space as usual, also copies that object-space position into an interpolant so it becomes available to the fragment shader. While we're at it, we might as well add diffuse and specular lighting, so our vertex shader needs to output the normal and light vector as well.

[Listings 23.21](#) and [23.22](#) show the high-level and low-level versions of this vertex shader. We'll use it for all three of our procedural texture mapping samples.

Listing 23.21. Procedural Texture Mapping High-Level Vertex Shader

```
// checkerboard.vs
//
// Generic vertex transformation,
// copy object-space position and
// lighting vectors out to interpolants
uniform vec3 lightPos;
varying vec3 N, L, V;
void main(void)
{
    // normal MVP transform
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    // map object-space position onto unit sphere
    V = gl_Vertex.xyz;
    // eye-space normal
    N = gl_NormalMatrix * gl_Normal;
    // eye-space light vector
    vec4 Veye = gl_ModelViewMatrix * gl_Vertex;
    L = lightPos - Veye.xyz;
}
```

Listing 23.22. Procedural Texture Mapping Low-Level Vertex Shader

```
!ARBvp1.0
# checkerboard.vp
#
# Generic vertex transformation,
# copy object-space position and
# light vectors out to interpolants
ATTRIB iPos = vertex.position;           # input position
ATTRIB iNrm = vertex.normal;             # input normal
OUTPUT oPos = result.position;           # output position
OUTPUT oTC0 = result.texcoord[0];         # output texcoord 0: N
OUTPUT oTC1 = result.texcoord[1];         # output texcoord 1: L
OUTPUT oTC2 = result.texcoord[2];         # output texcoord 2: V
PARAM mvp[4] = { state.matrix.mvp }; # model-view * proj matrix
PARAM mv[4] = { state.matrix.modelview }; # model-view matrix
# inverse transpose of model-view matrix:
```

```

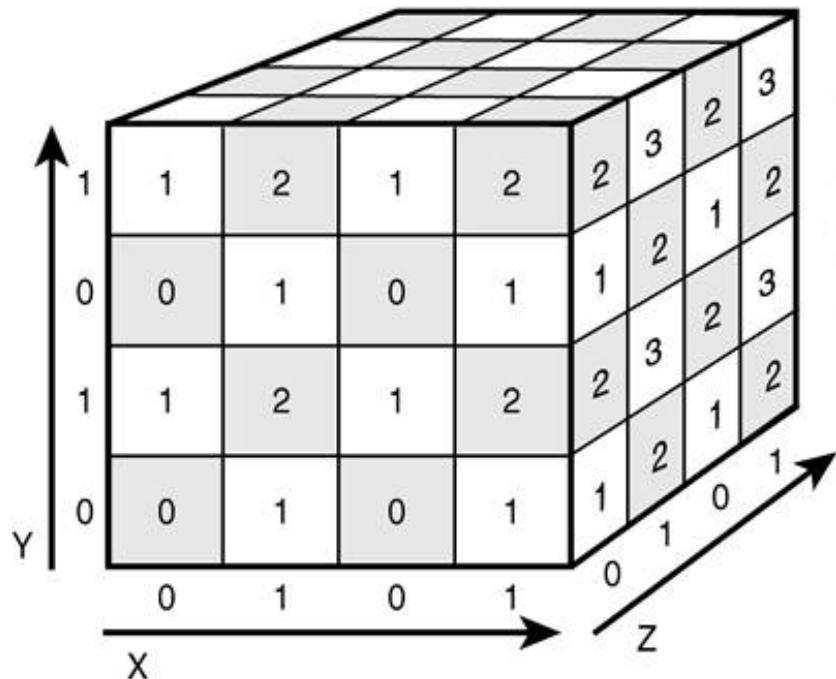
PARAM mvIT[4] = { state.matrix.modelview.invtrans };
PARAM lightPos = program.local[0];    # light pos in eye space
TEMP V;                                # temporary register
DP4 oPos.x, iPos, mvp[0];                # xform input pos by MVP
DP4 oPos.y, iPos, mvp[1];
DP4 oPos.z, iPos, mvp[2];
DP4 oPos.w, iPos, mvp[3];
DP4 V.x, iPos, mv[0];                   # xform input pos by MV
DP4 V.y, iPos, mv[1];
DP4 V.z, iPos, mv[2];
DP4 V.w, iPos, mv[3];
DP3 oTC0.x, iNrm, mvIT[0];              # xform norm to eye space
DP3 oTC0.y, iNrm, mvIT[1];
DP3 oTC0.z, iNrm, mvIT[2];              # put N in texcoord 0
SUB oTC1, lightPos, V;                  # light vector in texcoord 1
MOV oTC2, iPos;                         # put objPos in texcoord 2
END

```

The object we're using for our samples is a sphere. The size of the sphere doesn't matter because we normalize the object-space position at the beginning of the fragment shader. This means that all the positions we deal with in the fragment shader will be in the range $[-1,1]$.

Our strategy for the fragment shader will be to break up the range $[-1,1]$ into eight alternating blocks along each axis. Each block will be assigned an alternating value of 0 or 1 for each axis, as illustrated in [Figure 23.12](#). If the total of the three values is even, we paint it black; otherwise, we paint it white.

Figure 23.12. This diagram illustrates how we assign alternating colors to blocks of fragments.



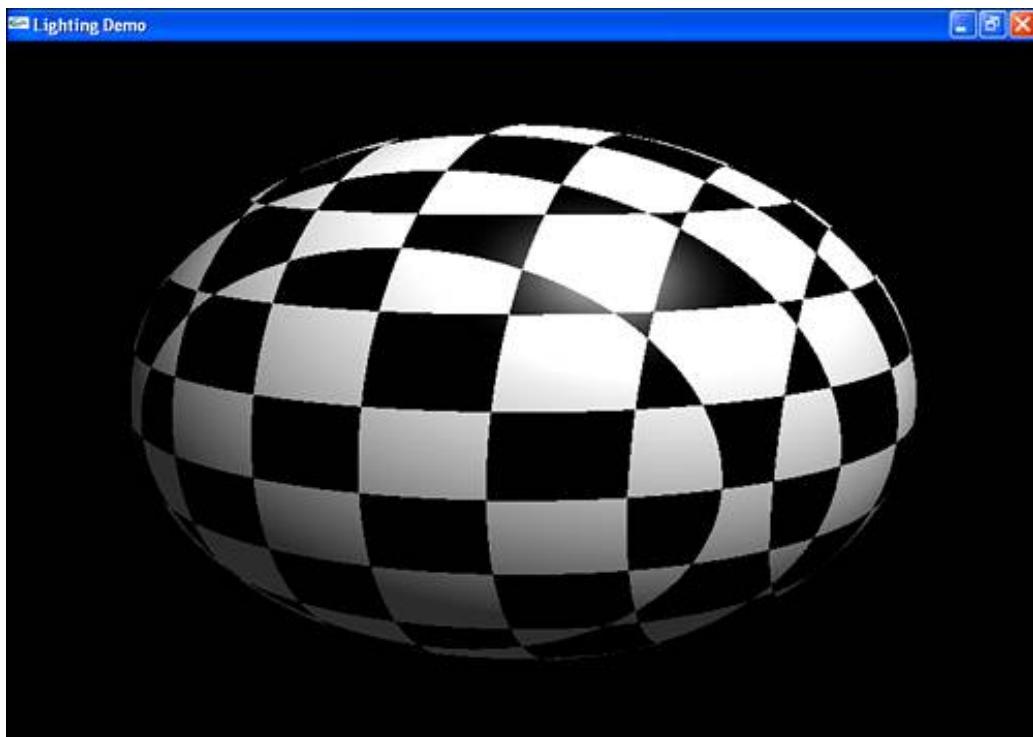
[Figure 23.13](#) shows the result of [Listings 23.23](#) and [23.24](#), which implement our checkerboard procedural texture mapping algorithm.

Listing 23.23. Checkerboard High-Level Fragment Shader

```
// checkerboard.fs
```

```
//  
// 3D solid checker grid  
varying vec3 V; // object-space position  
varying vec3 N; // eye-space normal  
varying vec3 L; // eye-space light vector  
const vec3 onColor = vec3(1.0, 1.0, 1.0);  
const vec3 offColor = vec3(0.0, 0.0, 0.0);  
const float ambientLighting = 0.2;  
const float specularExp = 60.0;  
const float specularIntensity = 0.75;  
const int numSquaresPerSide = 8;  
void main (void)  
{  
    // Normalize vectors  
    vec3 NN = normalize(N);  
    vec3 NL = normalize(L);  
    vec3 NV = normalize(V);  
    vec3 NH = normalize(NL + vec3(0.0, 0.0, 1.0));  
    // Map -1,1 to 0,numSquaresPerSide  
    vec3 onOrOff = ((NV + 1.0) * float(numSquaresPerSide)) / 2.0;  
    // mod 2 >= 1  
    onOrOff = step(1.0, mod(onOrOff, 2.0));  
    // 3-way xor  
    onOrOff.x = step(0.5,  
        mod(onOrOff.x + onOrOff.y + onOrOff.z, 2.0));  
    // checkerboard grid  
    vec3 surfColor = mix(offColor, onColor, onOrOff.x);  
    // calculate diffuse lighting + 20% ambient  
    surfColor *= (ambientLighting + vec3(max(0.0, dot(NN, NL))));  
    // calculate specular lighting w/ 75% intensity  
    surfColor += (specularIntensity *  
        vec3(pow(max(0.0, dot(NN, NH)), specularExp)));  
    gl_FragColor = vec4(surfColor, 1.0);  
}
```

Figure 23.13. This 3D checkerboard is generated without using any texture images.



Listing 23.24. Checkerboard Low-Level Fragment Shader

```
!ARBfp1.0
# checkerboard.fp
#
# 3D solid checker grid
ATTRIB N = fragment.texcoord[0];
ATTRIB L = fragment.texcoord[1];
ATTRIB V = fragment.texcoord[2];      # obj-space position
OUTPUT oPrC = result.color;          # output color
PARAM onColor = {1.0, 1.0, 1.0, 1.0};
PARAM offColor = {0.0, 0.0, 0.0, 1.0};
# 0.25 * squares per side, ambient lighting,
# specular exponent, specular intensity
PARAM misc = {2.0, 0.2, 60.0, 0.75};
TEMP NV, NN, NL, NH, NdotL, NdotH, surfColor, onOrOff;
ALIAS specular = NdotH;
DP3 NV.w, V, V;                      # normalize vertex pos
RSQ NV.w, NV.w;
MUL NV, V, NV.w;
# Map position from -1,1 to 0,numSquaresPerSide/2
MAD onOrOff, NV, misc.x, misc.x;
# mod2 by doubling FRC, then subtract 1 for >= 1 compare
FRC onOrOff, onOrOff;
MAD onOrOff, onOrOff, 2.0, -1.0;
CMP onOrOff, onOrOff, 0.0, 1.0;
# perform xor by adding all 3 axes' onoroff values,
# then mod2 again
DP3 onOrOff, onOrOff, 1.0;
MUL onOrOff, onOrOff, 0.5;
FRC onOrOff, onOrOff;
MAD onOrOff, onOrOff, 2.0, -1.0;
CMP onOrOff, onOrOff, 0.0, 1.0;
# checkerboard grid
LRP surfColor, onOrOff, onColor, offColor;
DP3 NN.w, N, N;                      # normalize normal
RSQ NN.w, NN.w;
MUL NN, N, NN.w;
DP3 NL.w, L, L;                      # normalize light vec
RSQ NL.w, NL.w;
MUL NL, L, NL.w;
ADD NH, NL, {0, 0, 1};                # half-angle vector
DP3 NH.w, NH, NH;                   # normalize it
RSQ NH.w, NH.w;
MUL NH, NH, NH.w;
# diffuse lighting
DP3 NdotL, NN, NL;                  # N . L
MAX NdotL, NdotL, 0.0;                # max(N . L, 0)
ADD NdotL, NdotL, misc.y;            # 20% ambient
MUL surfColor, surfColor, NdotL;     # factor in diffuse color
# specular lighting
DP3 NdotH, NN, NH;                  # N . H
MAX NdotH, NdotH, 0.0;                # max(N . H, 0)
POW specular, NdotH.x, misc.z;       # NdotH^60
MAD oPrC, misc.w, specular, surfColor; # 75% specular intensity
END
```

GLSL has a built-in modulo function (`mod`), which is used to achieve the alternating blocks. However, we have to work a little harder to perform the modulo 2 operation in our low-level fragment shader. We take the value, divide it by 2, use the `FRC` instruction to get the fractional

part, and then multiply that by 2. What we're left with is a value in the range [0,2].

Next, we must determine whether the value is within [0,1] or [1,2]. We do this in GLSL using the `step` function, which returns 1 if the second argument is greater than or equal to the first, and 0 otherwise. In the low-level shader, we can do this using the `CMP` instruction, but because it compares the first argument to 0, not 1, we first subtract 1 from the argument.

Now that we have a value of 0 or 1 on each axis, we sum those three values and again perform modulo 2 and a greater than or equal to comparison. That way, we can assign colors of black or white based on whether the final sum is even or odd. We accomplish this with `mix` in the high-level shader or `LRP` in the low-level version.

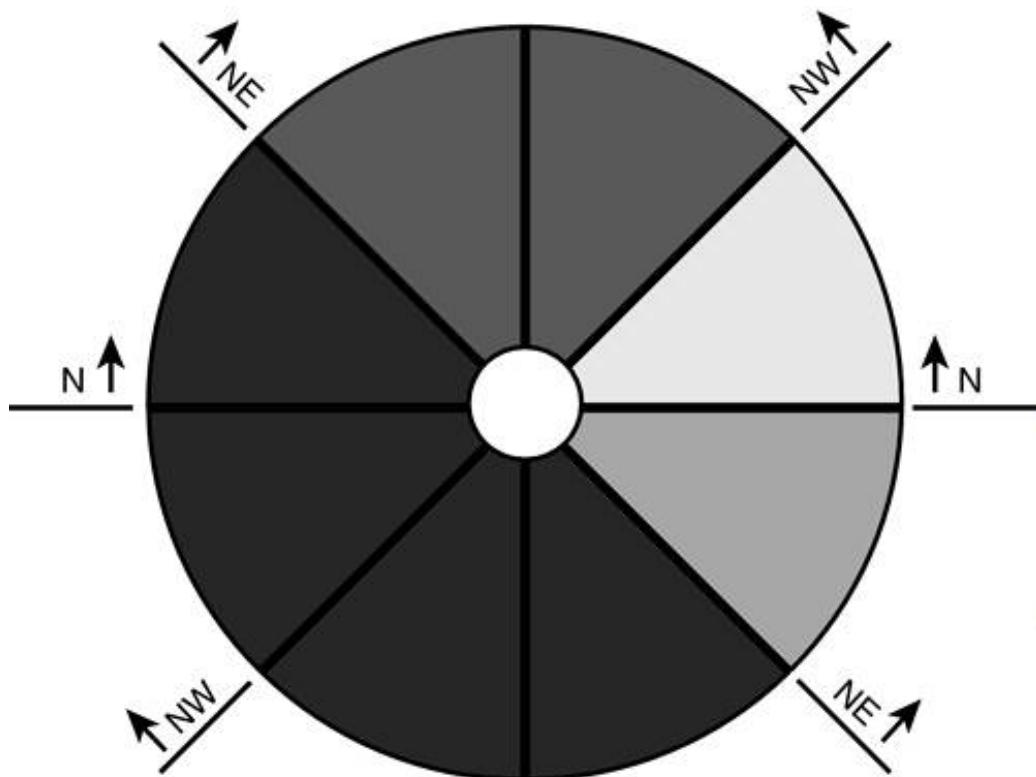
You can very easily alter the shaders to change the checkerboard colors or to adjust the number of blocks per row. Give it a try!

Beach Ball Texture

In this next sample, we're going to turn our sphere into a beach ball. The ball will have eight longitudinal stripes with alternating primary colors. The north and south poles of the ball will be painted white. Let's get started!

Look at the ball from above. We'll be slicing it up into three half spaces: north-south, northeast-southwest, and northwest-southeast. See [Figure 23.14](#) for a visual depiction. The north slices are assigned full red values, while south slices are assigned no red. The two slices that are both in the southeast half space and the northeast half space are assigned full green, while all other slices receive no green. Notice how the overlapping red and green slice becomes yellow. Finally, all slices in the southwest half space are assigned the color blue.

Figure 23.14. An overhead view showing how the beachball colors are assigned.



The east slices nicely alternate from red to yellow to green to blue. But what about the west slices? The easiest way to address them is to effectively copy the east slices and rotate them 180 degrees. We're looking down at the ball from the positive y-axis. If the object-space position's x

coordinate is greater than or equal to 0, the position is used as-is. However, if the coordinate is less than 0, we negate both the x-axis and z-axis position, which maps the original position to its mirror on the opposite side of the beach ball.

The white caps at the poles are simple to add in. After coloring the rest of the ball with stripes, we replace that color with white whenever the absolute value of the y-axis position is close to 1. Figure 23.15 shows the result of the beach ball shaders in [Listings 23.25](#) and [23.26](#).

Listing 23.25. Beach Ball High-Level Fragment Shader

```
// beachball.fs
//
// Longitudinal stripes, end caps
varying vec3 V; // object-space position
varying vec3 N; // eye-space normal
varying vec3 L; // eye-space light vector
const vec3 myRed = vec3(1.0, 0.0, 0.0);
const vec3 myYellow = vec3(1.0, 1.0, 0.0);
const vec3 myGreen = vec3(0.0, 1.0, 0.0);
const vec3 myBlue = vec3(0.0, 0.0, 1.0);
const vec3 myWhite = vec3(1.0, 1.0, 1.0);
const vec3 myBlack = vec3(0.0, 0.0, 0.0);
const vec3 northHalfSpace = vec3(0.0, 0.0, 1.0);
const vec3 northeastHalfSpace = vec3(0.707, 0.0, 0.707);
const vec3 northwestHalfSpace = vec3(-0.707, 0.0, 0.707);
const float capSize = 0.03; // 0 to 1
const float smoothEdgeTol = 0.005;
const float ambientLighting = 0.2;
const float specularExp = 60.0;
const float specularIntensity = 0.75;

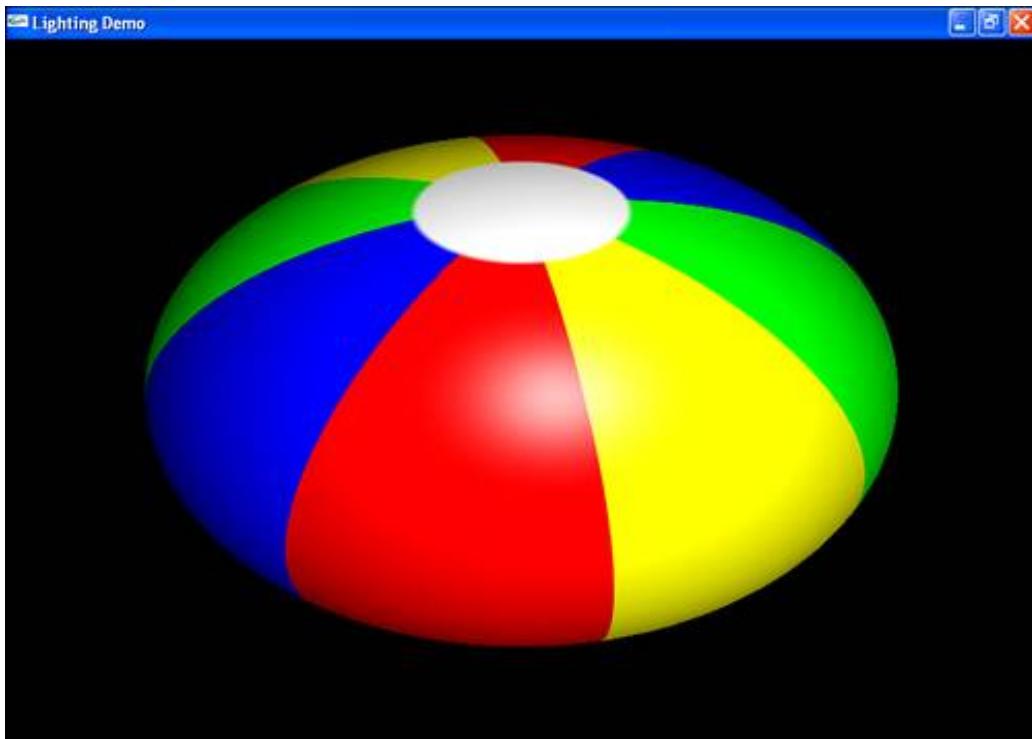
void main (void)
{
    // Normalize vectors
    vec3 NN = normalize(N);
    vec3 NL = normalize(L);
    vec3 NH = normalize(NL + vec3(0.0, 0.0, 1.0));
    vec3 NV = normalize(V);
    // Mirror half of ball across X and Z axes
    float mirror = (NV.x >= 0.0) ? 1.0 : -1.0;
    NV.xz *= mirror;
    // Check for north/south, east/west,
    // northeast/southwest, northwest/southeast
    vec4 distance;
    distance.x = dot(NV, northHalfSpace);
    distance.y = dot(NV, northeastHalfSpace);
    distance.z = dot(NV, northwestHalfSpace);
    // setup for white caps on top and bottom
    distance.w = abs(NV.y) - 1.0 + capSize;
    distance = smoothstep(vec4(0.0), vec4(smoothEdgeTol), distance);
    // red, green, red+green=yellow, and blue stripes
    vec3 surfColor = mix(myBlack, myRed, distance.x);
    surfColor += mix(myBlack, myGreen, distance.y*(1.0-distance.z));
    surfColor = mix(surfColor, myBlue, 1.0-distance.y);
    // white caps on top and bottom
    surfColor = mix(surfColor, myWhite, distance.w);
    // calculate diffuse lighting + 20% ambient
    surfColor *= (ambientLighting + vec3(max(0.0, dot(NN, NL))));
    // calculate specular lighting w/ 75% intensity
    surfColor += (specularIntensity *
        vec3(pow(max(0.0, dot(NN, NH)), specularExp))));
```

```

    gl_FragColor = vec4(surfColor, 1.0);
}

```

Figure 23.15. You have built your own beach ball from scratch!



Listing 23.26. Beach Ball Low-Level Fragment Shader

```

!!ARBfp1.0
# beachball.fp
#
# Longitudinal stripes, end caps
ATTRIB N = fragment.texcoord[0];
ATTRIB L = fragment.texcoord[1];
ATTRIB V = fragment.texcoord[2];      # obj-space position
OUTPUT oPrC = result.color;          # output color
PARAM myRed = {1.0, 0.0, 0.0, 1.0};
PARAM myYellow = {1.0, 1.0, 0.0, 1.0};
PARAM myGreen = {0.0, 1.0, 0.0, 1.0};
PARAM myBlue = {0.0, 0.0, 1.0, 1.0};
PARAM myWhite = {1.0, 1.0, 1.0, 1.0};
PARAM myBlack = {0.0, 0.0, 0.0, 1.0};
PARAM northHalfSpace = {0.0, 0.0, 1.0};
PARAM northeastHalfSpace = {0.707, 0.0, 0.707};
PARAM northwestHalfSpace = {-0.707, 0.0, 0.707};
# cap size minus one, ambient lighting,
# specular exponent, specular intensity
PARAM misc = {-0.97, 0.2, 60.0, 0.75};
TEMP NV, NN, NL, NH, NdotL, NdotH, surfColor, distance, mirror;
ALIAS specular = NdotH;
ALIAS redColor = NV;
DP3 NV.w, V, V;                      # normalize vertex pos
RSQ NV.w, NV.w;
MUL NV, V, NV.w;

# Mirror half of ball across X and Z axes
CMP mirror, NV.x, -1.0, 1.0;

```

```

MUL NV.xz, NV, mirror;
# Check for north/south, east/west,
# northeast/southwest, northwest/southeast
DP3 distance.x, NV, northHalfSpace;
DP3 distance.y, NV, northeastHalfSpace;
DP3 distance.z, NV, northwestHalfSpace;
# setup for white caps on top and bottom
ABS distance.w, NV.y;
ADD distance.w, distance.w, misc.x;
CMP distance, distance, 0.0, 1.0;
# red, green, red+green=yellow, and blue stripes
LRP redColor, distance.x, myRed, myBlack;
MAD distance.z, -distance.y, distance.z, distance.y;
LRP surfColor, distance.z, myGreen, myBlack;
ADD surfColor, surfColor, redColor;
SUB distance.y, 1.0, distance.y;
LRP surfColor, distance.y, myBlue, surfColor;
# white caps on top and bottom
LRP surfColor, distance.w, myWhite, surfColor;
DP3 NN.w, N, N;                                # normalize normal
RSQ NN.w, NN.w;
MUL NN, N, NN.w;
DP3 NL.w, L, L;                                # normalize light vec
RSQ NL.w, NL.w;
MUL NL, L, NL.w;
ADD NH, NL, {0, 0, 1};                          # half-angle vector
DP3 NH.w, NH, NH;                            # normalize it
RSQ NH.w, NH.w;
MUL NH, NH, NH.w;
# diffuse lighting
DP3 NdotL, NN, NL;                            # N . L
MAX NdotL, NdotL, 0.0;                          # max(N . L, 0)
ADD NdotL, NdotL, misc.y;                      # 20% ambient
MUL surfColor, surfColor, NdotL;                # factor in diffuse color
# specular lighting
DP3 NdotH, NN, NH;                            # N . H
MAX NdotH, NdotH, 0.0;                          # max(N . H, 0)
POW specular, NdotH.x, misc.z;                # NdotH^60
MAD oPrC, misc.w, specular, surfColor; # 75% specular intensity
END

```

After remapping all negative x positions as described earlier, we use dot products to determine on which side of each half space the current object-space coordinate falls. The sign of the dot product tells us which side of the half space is in play. In the low-level shader, we use the `CMP` instruction to perform this greater than or equal to 0 comparison. However, in the GLSL shader, we don't use the built-in `step` function this time. Instead, we introduce a new and improved version:

`smoothstep`.

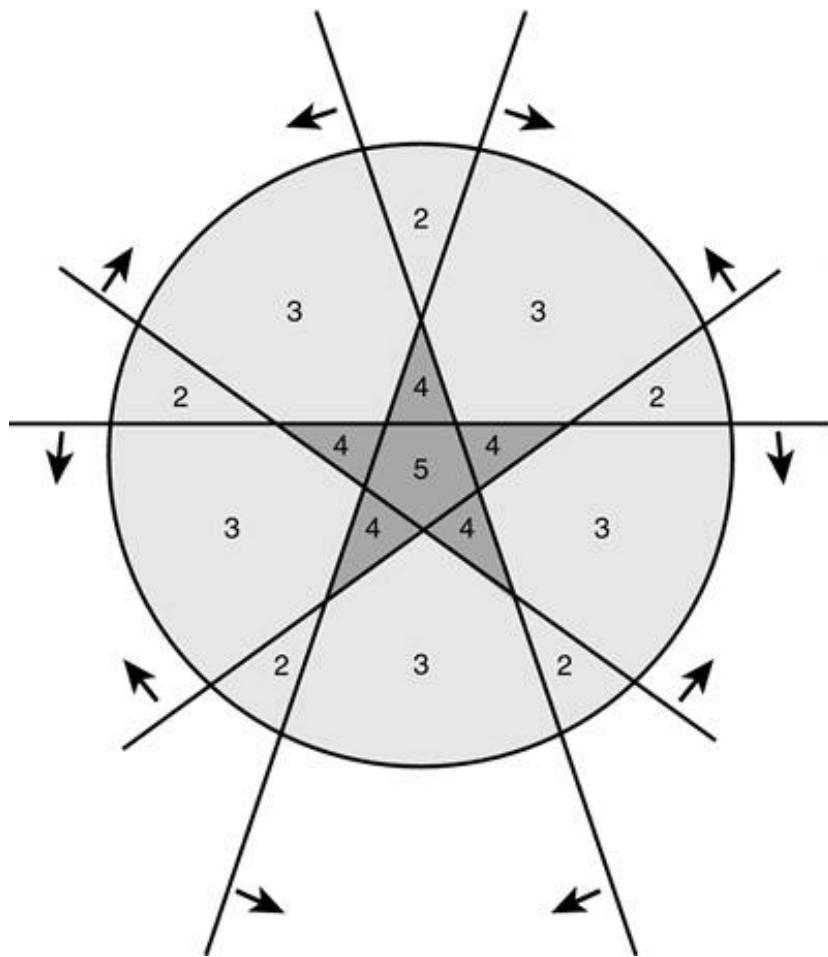
Instead of transitioning directly from 0 to 1 at the edge of a half space, `smoothstep` allows for a smooth transition near the edge where values between 0 and 1 are returned. Switch back and forth between the high-level and low-level versions and you'll see how `smoothstep` helps reduce the aliasing jaggies.

Toy Ball Texture

For our final procedural texture mapping feat, we'll transform our sphere into a familiar toy ball, again using no conventional texture images. This ball will have a red star on a yellow background circumscribed by a blue stripe. We will describe all this inside a fragment shader.

The tricky part is obviously the star shape. For each fragment, the shader must determine whether the fragment is within the star, in which case it's painted red, or whether it remains outside the star, in which case it's painted yellow. To make this determination, we first detect whether the fragment is inside or outside five different half spaces, as shown in [Figure 23.16](#).

Figure 23.16. This diagram illustrates the determination of whether a fragment is inside or outside the star as described by 5 half spaces.



Any fragment that is inside at least four of the five half spaces is inside the star. We'll start a counter at -3 and increment it for every half space that the fragment falls within. Then we'll clamp it to the range $[0,1]$. A 0 indicates that we're outside the star and should paint the fragment yellow. A 1 indicates that we're inside the star and should paint the fragment red.

Adding the blue stripe, like the white caps on the beach ball, is an easy last step. Instead of repainting fragments close to the ends of the ball, we repaint them close to the center, this time along the z -axis. [Figure 23.17](#) illustrates the result of the toy ball shaders in [Listings 23.27](#) and [23.28](#).

Listing 23.27. Toy Ball High-Level Fragment Shader

```
// toyball.fs
//
// Based on shader by Bill Licea-Kane
varying vec3 V; // object-space position
varying vec3 N; // eye-space normal
varying vec3 L; // eye-space light vector
const vec3 myRed = vec3(0.6, 0.0, 0.0);
const vec3 myYellow = vec3(0.6, 0.5, 0.0);
const vec3 myBlue = vec3(0.0, 0.3, 0.6);
```

```

const vec3 myHalfSpace0 = vec3(0.31, 0.95, 0.0);
const vec3 myHalfSpace1 = vec3(-0.81, 0.59, 0.0);
const vec3 myHalfSpace2 = vec3(-0.81, -0.59, 0.0);
const vec3 myHalfSpace3 = vec3(0.31, -0.95, 0.0);
const vec3 myHalfSpace4 = vec3(1.0, 0.0, 0.0);
const float stripeThickness = 0.4; // 0 to 1
const float starSize = 0.2; // 0 to ~0.3
const float smoothEdgeTol = 0.005;
const float ambientLighting = 0.2;
const float specularExp = 60.0;
const float specularIntensity = 0.5;
void main (void)
{
    vec4 distVector;
    float distScalar;
    // Normalize vectors
    vec3 NN = normalize(N);
    vec3 NL = normalize(L);
    vec3 NH = normalize(NL + vec3(0.0, 0.0, 1.0));
    vec3 NV = normalize(V);
    // Each flat edge of the star defines a half-space. The interior
    // of the star is any point within at least 4 out of 5 of them.
    // Start with -3 so that it takes adding 4 ins to equal 1.
    float myInOut = -3.0;
    // We need to perform 5 dot products, one for each edge of
    // the star. Perform first 4 in vector, 5th in scalar.
    distVector.x = dot(NV, myHalfSpace0);
    distVector.y = dot(NV, myHalfSpace1);
    distVector.z = dot(NV, myHalfSpace2);
    distVector.w = dot(NV, myHalfSpace3);
    distScalar = dot(NV, myHalfSpace4);
    // The half-space planes all intersect the origin. We must
    // offset them in order to give the star some size.
    distVector += starSize;
    distScalar += starSize;
    distVector = smoothstep(0.0, smoothEdgeTol, distVector);
    distScalar = smoothstep(0.0, smoothEdgeTol, distScalar);
    myInOut += dot(distVector, vec4(1.0));
    myInOut += distScalar;
    myInOut = clamp(myInOut, 0.0, 1.0);
    // red star on yellow background
    vec3 surfColor = mix(myYellow, myRed, myInOut);
    // blue stripe down middle
    myInOut = smoothstep(0.0, smoothEdgeTol,
                        abs(NV.z) - stripeThickness);
    surfColor = mix(myBlue, surfColor, myInOut);
    // calculate diffuse lighting + 20% ambient
    surfColor *= (ambientLighting + vec3(max(0.0, dot(NN, NL))));
    // calculate specular lighting w/ 50% intensity
    surfColor += (specularIntensity *
                  vec3(pow(max(0.0, dot(NN, NH)), specularExp)));
    gl_FragColor = vec4(surfColor, 1.0);
}

```

Figure 23.17. The toy ball shader describes a relatively complex shape.



Listing 23.28. Toy Ball Low-Level Fragment Shader

```
!!ARBfp1.0
# toyball.fp
#
# Based on shader by Bill Licea-Kane
ATTRIB N = fragment.texcoord[0];
ATTRIB L = fragment.texcoord[1];
ATTRIB V = fragment.texcoord[2];           # obj-space position
OUTPUT oPrC = result.color;                # output color
PARAM myRed = {0.6, 0.0, 0.0, 1.0};
PARAM myYellow = {0.6, 0.5, 0.0, 1.0};
PARAM myBlue = {0.0, 0.3, 0.6, 1.0};
PARAM myHalfSpace0 = {0.31, 0.95, 0.0};
PARAM myHalfSpace1 = {-0.81, 0.59, 0.0};
PARAM myHalfSpace2 = {-0.81, -0.59, 0.0};
PARAM myHalfSpace3 = {0.31, -0.95, 0.0};
PARAM myHalfSpace4 = {1.0, 0.0, 0.0};
# stripe thickness, star size & ambient lighting,
# specular exponent, specular intensity
PARAM misc = {0.4, 0.2, 60.0, 0.5};
TEMP NV, NN, NL, NH, NdotL, NdotH, surfColor, distance, myInOut;
ALIAS specular = NdotH;
DP3 NV.w, V, V;                           # normalize vertex pos
RSQ NV.w, NV.w;
MUL NV, V, NV.w;
# Each flat edge of the star defines a half-space. The interior
# of the star is any point within at least 4 out of 5 of them.
# Start with -3 so that it takes adding 4 ins to equal 1.
MOV myInOut, -3.0;

# We need to perform 5 dot products, one for each edge of
# the star. Perform first 4 in vector, 5th in a second
# vector along with the blue stripe.
DP3 distance.x, NV, myHalfSpace0;
DP3 distance.y, NV, myHalfSpace1;
DP3 distance.z, NV, myHalfSpace2;
```

```

DP3 distance.w, NV, myHalfSpace3;
# The half-space planes all intersect the origin.  We must
# offset them in order to give the star some size.
ADD distance, distance, misc.y;
CMP distance, distance, 0.0, 1.0;
DP4 distance, distance, 1.0;
ADD myInOut, myInOut, distance;
# set up last star edge and blue stripe
DP3 distance.x, NV, myHalfSpace4;
ADD distance.x, distance.x, misc.y;
ABS distance.y, NV.z;
SUB distance.y, distance.y, misc.x;
CMP distance, distance, 0.0, 1.0;
ADD_SAT myInOut, myInOut, distance.x;
# red star on yellow background
LRP surfColor, myInOut, myRed, myYellow;
# blue stripe down middle
LRP surfColor, distance.y, surfColor, myBlue;
DP3 NN.w, N, N;                                # normalize normal
RSQ NN.w, NN.w;
MUL NN, N, NN.w;
DP3 NL.w, L, L;                                # normalize light vec
RSQ NL.w, NL.w;
MUL NL, L, NL.w;
ADD NH, NL, {0, 0, 1};                          # half-angle vector
DP3 NH.w, NH, NH;                            # normalize it
RSQ NH.w, NH.w;
MUL NH, NH, NH.w;
# diffuse lighting
DP3 NdotL, NN, NL;                            # N . L
MAX NdotL, NdotL, 0.0;                          # max(N . L, 0)
ADD NdotL, NdotL, misc.y;                      # 20% ambient
MUL surfColor, surfColor, NdotL;                # factor in diffuse color
# specular lighting
DP3 NdotH, NN, NH;                            # N . H
MAX NdotH, NdotH, 0.0;                          # max(N . H, 0)
POW specular, NdotH.x, misc.z;                # NdotH^60
MAD oPrC, misc.w, specular, surfColor; # 50% specular intensity
END

```

The half spaces cut through the center of the sphere. This is what we wanted for the beach ball, but for the star we need them offset from the center slightly. This is why we add an extra constant distance to the result of the half space dot products. The larger you make this constant, the larger your star will be.

Again, we use `smoothstep` in the GLSL shader and `CMP` in the low-level shader when picking between inside and outside. For efficiency, we put the inside/outside results of the first four half spaces into a four-component vector. This way, we can sum the four components with a single four-component dot product against the vector $\{1, 1, 1, 1\}$. The fifth half space's inside/outside value goes into a lonely float and is added to the other four separately because no five-component vector type is available. You could create such a type yourself out of a structure, but you would likely sacrifice performance on most implementations, which natively favor four-component vectors.

If you want to toy with this shader, try this exercise: Convert the star into a six-pointed star by adding another half space and adjusting the existing half space planes. Prove to yourself how many half spaces your fragments must fall within now to fall within the star, and adjust the `myInOut` counter's initial value accordingly.

Summary

The possible applications of vertex and fragment shaders are limited only by your imagination. We've introduced a few just to spark your creativity and to provide you with some basic building blocks so you can easily jump right in and start creating your own shaders. Feel free to take these shaders, hack and slash them beyond recognition, and invent and discover better ways of doing things while you're at it. Don't forget the main objective of this book: Make pretty pictures. So get to it!

Appendix A. Further Reading

Real-time 3D graphics and OpenGL are popular topics, and there are more information and techniques in practice than can ever be published in a single book. You might find the following resources helpful as you further your knowledge and experience.

Other Good OpenGL Books

OpenGL Programming Guide, 4th Edition: The Official Guide to Learning OpenGL, Version 1.4. OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. Addison-Wesley, 2003.

OpenGL Programming for the X Window System. Mark J. Kilgard. Addison-Wesley, 1996.

Interactive Computer Graphics: A Top-Down Approach with OpenGL, 3rd Edition. Edward Angel. Addison-Wesley, 2002.

The OpenGL Extensions Guide. Eric Lengyel. Charles River Media, 2003.

OpenGL Shading Language. Randi J. Rost. Addison-Wesley, 2004.

3D Graphics Books

3D Computer Graphics. Alan Watt. Addison-Wesley, 1993.

3D Math Primer for Graphics and Game Development. Fletcher Dunn and Ian Parbery. Wordware Publishing, 2002.

Advanced Animation and Rendering Techniques: Theory and Practice. Alan Watt and Mark Watt (contributor). Addison-Wesley, 1992.

Introduction to Computer Graphics. James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes, and Richard L. Phillips. Addison-Wesley, 1993.

Open Geometry: OpenGL + Advanced Geometry. Georg Glaeser and Hellmuth Stachel. Springer-Verlag, 1999.

Mathematics for 3D Game Programming & Computer Graphics. Eric Lengyel. Charles River Media, 2001.

Web Sites

The OpenGL SuperBible Web Site

<http://www.starstonesoftware.com/OpenGL>

The Official OpenGL Web Site

<http://www.opengl.org>

The Khronos Group OpenGL ES Home Page

<http://www.khronos.org/opengles/index.html>

The SGI OpenGL Extension Registry

<http://oss.sgi.com/projects/ogl-sample/registry/>

SGI's OpenGL Web Site

http://www.sgi.com/software/opengl/tech_info.html

ATI's Developers Home Page

<http://www.ati.com/developer/index.html>

NVidia's Developers Home Page

<http://developer.nvidia.com/page/home>

3Dlabs' Developers Home Page

<http://www.3dlabs.com/support/developer/index.htm>

Many Great OpenGL Tutorials (Game Heavy)

<http://www.gamedev.net/>

<http://nehe.gamedev.net/>

<http://www.xmission.com/~nate/tutors.html>

<http://www.paulsprojects.net/opengl/projects1.html>

http://www.gametutorials.com/Tutorials/OpenGL/OpenGL_Pg1.htm

<http://www.codecolony.de/opengl.htm>

Appendix B. Glossary

Aliasing

Technically, the loss of signal information in an image reproduced at some finite resolution. It is most often characterized by the appearance of sharp jagged edges along points, lines, or polygons due to the nature of having a limited number of fixed-sized pixels.

Alpha

A fourth color value added to provide a degree of transparency to the color of an object. An alpha value of 0.0 means complete transparency; 1.0 denotes no transparency (opaque).

Ambient light

Light in a scene that doesn't come from any specific point source or direction. Ambient light illuminates all surfaces evenly and on all sides.

Antialiasing

A rendering method used to smooth lines and curves and polygon edges. This technique averages the color of pixels adjacent to the line. It has the visual effect of softening the transition from the pixels on the line and those adjacent to the line, thus providing a smoother appearance.

ARB

The Architecture Review Board. The OpenGL ARB meets quarterly and consists of 3D graphics hardware vendors. The ARB maintains the OpenGL Specification document and promotes the OpenGL standard.

Aspect ratio

The ratio of the width of a window to the height of the window. Specifically, the width of the window in pixels divided by the height of the window in pixels.

AUX library

A window system independent utility library. Limited but useful for quick and portable OpenGL demonstration programs. Now largely replaced by the GLUT library.

Bézier curve

A curve whose shape is defined by control points near the curve rather than by the precise set of points that define the curve itself.

Bitplane

An array of bits mapped directly to screen pixels.

Buffer

An area of memory used to store image information. This can be color, depth, or blending information. The red, green, blue, and alpha buffers are often collectively referred to as the color buffers.

Cartesian

A coordinate system based on three directional axes placed at a 90° orientation to one another. These coordinates are labeled x, y, and z.

Clipping

The elimination of a portion of a single primitive or group of primitives. The points that would be rendered outside the clipping region or volume are not drawn. The clipping volume is generally specified by the projection matrix. Clipped primitives are reconstructed such that the edges of the primitive do not lay outside the clipping region.

Clip coordinates

The 2D geometric coordinates that result from the modelview and projection transformation.

Color index mode

A color mode in which colors in a scene are selected from a fixed number of colors available in a palette. These entries are referenced by an index into the palette. This mode is rarely used and even more rarely hardware accelerated.

Convex

A reference to the shape of a polygon. A convex polygon has no indentations, and no straight line can be drawn through the polygon that intersects it more than twice (once entering, once leaving).

Culling

The elimination of graphics primitives that would not be seen if rendered. Backface culling eliminates the front or back face of a primitive so that the face isn't drawn. Frustum culling eliminates whole objects that would fall outside the viewing frustum.

Destination color

The stored color at a particular location in the color buffer. This terminology is usually used when describing blending operations to distinguish between the color already present in the color buffer and the color coming into the color buffer (source color).

Display list

A compiled list of OpenGL functions and commands. When called, a display list executes faster than a manually called list of single commands.

Dithering

A method used to simulate a wider range of color depth by placing different-colored pixels together in patterns that give the illusion of shading between the two colors.

Double buffered

A drawing technique used by OpenGL. The image to be displayed is assembled in memory and then placed on the screen in a single update operation, rather than built primitive by primitive on the screen. Double buffering is a much faster and smoother update operation and can produce animations.

Extruded

The process of taking a 2D image or shape and adding a third dimension uniformly across the surface. This process can transform 2D fonts into 3D lettering.

Eye coordinates

The coordinate system based on the position of the viewer. The viewer's position is placed along the positive z-axis, looking down the negative z-axis.

Frustum

A pyramid-shaped viewing volume that creates a perspective view. (Near objects are large; far objects are small.)

GLUT library

The OpenGL utility library. A window system independent utility library useful for creating sample programs and simple 3D rendering programs that are independent of the operating system and windowing system. Typically used to provide portability between Windows, X-Window, Linux, and so on.

Immediate mode

A graphics rendering mode in which commands and functions have an immediate effect on the state of the rendering engine.

Literal

A value, not a variable name. A specific string or numeric constant embedded directly in source code.

Matrix

A 2D array of numbers. Matrices can be operated on mathematically and are used to perform coordinate transformations.

Mipmapping

A technique that uses multiple levels of detail for a texture. This technique selects from among the different sizes of an image available, or possibly combines the two nearest sized matches to produce the final fragments used for texturing.

Modelview matrix

The OpenGL matrix that transforms primitives to eye coordinates from object coordinates.

Normal

A directional vector that points perpendicularly to a plane or surface. When used, normals must be specified for each vertex in a primitive.

Normalize

The reduction of a normal to a unit normal. A unit normal is a vector that has a length of exactly 1.0.

NURBS

An acronym for non-uniform rational b-spline. This is a method of specifying parametric curves and surfaces.

Open Inventor

A C++ class library and toolkit for building interactive 3D applications. Open Inventor is built on OpenGL.

Orthographic

A drawing mode in which no perspective or foreshortening takes place. Also called parallel projection. The lengths and dimensions of all primitives are undistorted regardless of orientation or distance from the viewer.

Palette

A set of colors available for drawing operations. For 8-bit Windows color modes, the palette contains 256 color entries, and all pixels in the scene can be colored from only this set.

Parametric curve

A curve whose shape is determined by one (for a curve) or two (for a surface) parameters. These parameters are used in separate equations that yield the individual x, y, and z values of the points along the curve.

Perspective

A drawing mode in which objects farther from the viewer appear smaller than nearby objects.

Pixel

Condensed from the words *picture element*. This is the smallest visual division available on the computer screen. Pixels are arranged in rows and columns and are individually set to the appropriate color to render any given image.

Pixmap

A two-dimensional array of color values that comprise a color image. Pixmaps are so called because each picture element corresponds to a pixel on the screen.

Polygon

A 2D shape drawn with any number of sides (must be at least three sides).

Primitive

A 2D polygonal shape defined by OpenGL. All objects and scenes are composed of various combinations of primitives.

Projection

The transformation of lines, points, and polygons from eye coordinates to clipping coordinates on the screen.

Quadrilateral

A polygon with exactly four sides.

Rasterize

The process of converting projected primitives and bitmaps into pixel fragments in the frame buffer.

Render

The conversion of primitives in object coordinates to an image in the frame buffer. The rendering pipeline is the process by which OpenGL commands and statements become pixels on the screen.

Scintillation

A sparkling or flashing effect produced on objects when a non-mipmapped texture map is applied to a polygon that is significantly smaller than the size of the texture being applied.

Source color

The color of the incoming fragment, as opposed to the color already present in the color buffer (destination color). This terminology is usually used when describing how the source and destination colors are combined during a blending operation.

Spline

A general term used to describe any curve created by placing control points near the curve, which have a pulling effect on the curve's shape. This is similar to the reaction of a piece of flexible material when pressure is applied at various points along its length.

Stipple

A binary bit pattern used to mask out pixel generation in the frame buffer. This is similar to a monochrome bitmap, but one-dimensional patterns are used for lines and two-dimensional patterns are used for polygons.

Tessellation

The process of breaking down a complex polygon or analytic surface into a mesh of convex polygons. This process can also be applied to separate a complex curve into a series of less complex lines.

Texel

Similar to pixel (picture element), a texel is a *texture element*. A texel represents a color from a texture that is applied to a pixel fragment in the frame buffer.

Texture

An image pattern of colors applied to the surface of a primitive.

Texture mapping

The process of applying a texture image to a surface. The surface does not have to be planar (flat). Texture mapping is often used to wrap an image around a curved object or to produce patterned surfaces such as wood or marble.

Transformation

The manipulation of a coordinate system. This can include rotation, translation, scaling (both uniform and nonuniform), and perspective division.

Translucence

A degree of transparency of an object. In OpenGL, this is represented by an alpha value ranging from 1.0 (opaque) to 0.0 (transparent).

Vertex

A single point in space. Except when used for point and line primitives, it also defines the point at which two edges of a polygon meet.

Viewing volume

The area in 3D space that can be viewed in the window. Objects and points outside the viewing volume are clipped (cannot be seen).

Viewport

The area within a window that is used to display an OpenGL image. Usually, this encompasses the entire client area. Stretched viewports can produce enlarged or shrunken output within the physical window.

Wireframe

The representation of a solid object by a mesh of lines rather than solid shaded polygons. Wireframe models are usually rendered faster and can be used to view both the front and back of an object at the same time.

Appendix C. OpenGL ES

OpenGL for Embedded Systems (OpenGL ES) is a lightweight version of OpenGL intended to bring portable and standard 3D graphics programming to a wide range of embedded systems. The term *embedded system* refers to a computer embedded in some device. Unlike a typical PC, where the computer's primary task is, well, computing, an embedded system is a computer that facilitates only the operation of the device it is embedded in. Examples of these types of devices are PDAs, cell phones with sophisticated graphics displays, medical and diagnostic devices, and automotive and avionic (airplane) displays. OpenGL ES is maintained by the Khronos Group, a consortium of media-centric companies, which is similar in some respects to the OpenGL Architecture Review Board (ARB). In fact, many of the member companies belong to both the ARB and the Khronos Group.

Basically, OpenGL ES is defined as a subset of OpenGL version 1.3, with a few additional extensions. This subset is further divided into two more "profiles": the Common profile and the Common-Lite or Safety Critical profile. The Common profile is simply a subset of OpenGL functionality intended to make OpenGL smaller in terms of the number of commands and in the memory footprint required for implementation. Another goal is to eliminate functionality that does not make as much sense in the embedded space. The Common-Lite profile is a further reduction of the feature set intended for devices with even less memory or available resources. Also called the Safety Critical profile, it is also useful in cases in which safety certifications are made easier by a smaller API footprint and less code that must be rigorously tested.

Reduction of Data Types

OpenGL supports a wide range of data types, and many functions allow you to specify data in whatever data type is convenient. The first step in reducing OpenGL is to reduce the number of data types supported at the high end and allow some of the smaller data types to be used for operations where they were not used before.

The first data type to go is `GLdouble`. The loss of this data type eliminates any function variations in OpenGL that supported double data types. Functions that are not eliminated altogether (see the following section) but use only doubles (`glOrtho`, for example) are kept, but they are changed to accept floating-point or fixed-point parameters.

To make OpenGL smaller for embedded systems, a new data type was added. The `GLFixed` data type is a fixed-point decimal number, and `GLclampx` is the "clamped" version of `GLfixed`. This type was added because the Common-Lite profile also eliminates `GLfloat`. Floating-point math hardware is often not included on embedded systems, and removing the data type removes the memory and performance overhead of having to emulate floating-point math operations.

The `GLbyte`, `GLubyte`, and `GLshort` function suffixes were also eliminated. This leaves only the command suffixes `i`, `f`, and `x`, with `f` also being eliminated from the Common-Lite profile function suffixes. You still can specify vertex and color data components as short or byte data types (but only in vertex arrays). The only exception is that color values may be specified as ubyte, but not as short. By using these smaller data types, you can reduce program size and data storage as well.

Totally Gone

The OpenGL ES specification (included on the CD) is more about what is removed from OpenGL than what OpenGL ES contains. This means it is not possible to understand OpenGL ES operation outside an understanding of OpenGL. Essentially, for both the Common and Common-Lite implementations, the following pieces of functionality have been removed in their entirety:

- Use of `Begin/End` for geometry specification (OpenGL ES uses only vertex arrays.)
- Interleaved arrays or support for `glArrayElement`
- Display lists
- Evaluators
- Color index mode
- User-defined clipping planes
- Line or polygon stippling
- `glRect`
- Imaging subset
- Feedback
- Selection
- Accumulation buffer
- Edge flags
- `glPolygonMode`
- The primitives `GL_QUADS`, `GL_QUAD_STRIP`, and `GL_POLYGON`
- Attribute saving: `glPushAttrib`, `glPopAttrib`, `glPushClientAttrib`, or `glPopClientAttrib`

Greatly Reduced Functionality

Many other features of OpenGL ES are greatly reduced in their scope and functionality. OpenGL ES still provides limited support for specifying the current color, normal, and texture coordinates using fixed-point or floating-point forms of the commands `glColor4`, `glNormal3`, and `MultiTexCoord4`. You use these functions, for example, when one of these states remains constant for an entire vertex array.

The full transformation pipeline is still mostly in place, but it works only with the newly specified subset data types (no doubles and so forth). OpenGL ES also does not support the transpose matrix, and the minimum depth of the modelview matrix stack has been changed from 32 to 16.

Texture Mapping

Only 2D textures are supported in OpenGL ES. Multitexture remains optional, but the `GL_COMBINE` texture environment is not. There is no support for texture coordinate generation or cube maps. There is no support for texture borders or the wrap modes `GL_CLAMP` or `GL_CLAMP_TO_BORDER`. Also missing are texture proxies, the LOD clamping, and bias parameters. Finally, texture compression is supported, but you cannot read back a compressed texture, and you cannot use `glTexImage2D` to compress an uncompressed image.

Raster Operations

Most raster functionality is gone from OpenGL ES. The `glPixelStore` function is still partially supported, but only for packing and unpacking texture data. You can still read pixels with `glReadPixels`, but `glDrawPixels`, `glPixelTransfer`, and `glPixelZoom` are not supported. Although `glReadPixels` still exists, you cannot use it to read from the depth or stencil buffers. The `glReadBuffer`, `glDrawBuffer`, and `glCopyPixels` functions were also dropped. Polygon offset is supported only in fill mode (`glPolygonMode` is no longer supported anyway).

Lighting

OpenGL ES must still support at least eight light sources, and two-sided lighting is still supported, although both sides must now have the same material properties (no difference between front and back material properties any longer). The only color material mode is `GL_AMBIENT_AND_DIFFUSE`, and the secondary color and local viewer lighting models were dropped.

Conclusion

OpenGL ES is a lean-and-mean 3D API providing the bare minimum, but sufficient, features to meet the needs of 3D graphics programmers in the widely defined embedded systems marketplace. Programming for OpenGL ES requires an SDK for the target platform that contains information on how to create and use a 3D context for that device's screen. PC emulators for OpenGL ES are just becoming available at the time of this printing, and you can check the Khronos (www.khronos.org) Web site for a list of vendors supporting OpenGL and links to further developers' resources.

CD-ROM

[What's on the CD-ROM](#)

[Windows Installation Instructions](#)

[License Agreement](#)

What's on the CD-ROM

The companion CD-ROM contains all of the source code for the book, tools, and OpenGL demos.

Windows Installation Instructions

1.

Insert the disc into your CD-ROM drive.

2.

From the Windows desktop, double-click the My Computer icon.

3.

Double-click the icon representing your CD-ROM drive.

4.

Double-click on `start.exe`. Follow the on-screen prompts to access the CD-ROM information.

NOTE

If you have the AutoPlay feature enabled, `start.exe` will be launched automatically whenever you insert the disc into your CD-ROM drive.

License Agreement

By opening this package, you are also agreeing to be bound by the following agreement:

You may not copy or redistribute the entire CD-ROM as a whole. Copying and redistribution of individual software programs on the CD-ROM is governed by terms set by individual copyright holders.

The installer and code from the author(s) are copyrighted by the publisher and the author(s). Individual programs and other items on the CD-ROM are copyrighted or are under an Open Source license by their various authors or other copyright holders.

This software is sold as-is without warranty of any kind, either expressed or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. Neither the publisher nor its dealers or distributors assumes any liability for any alleged or actual damages arising from the use of this program. (Some states do not allow for the exclusion of implied warranties, so the exclusion might not apply to you.)

NOTE: This CD-ROM uses long and mixed-case filenames requiring the use of a protected-mode CD-ROM Driver.