

OpenGL[®] ES 2.0 Programming Guide



Aaftab Munshi ■ Dan Ginsburg ■ Dave Shreiner
Foreword by Neil Trevett, President, Khronos Group

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals. The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, contact:

U.S. Corporate and Government Sales, (800) 382-3419, corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales, international@pearson.com

Visit us on the Web: informit.com/aw

This Book Is Safari Enabled



The Safari® Enabled icon on the cover of your favorite technology book means the book is available through Safari Bookshelf. When you buy this book, you get free access to the online edition for 45 days.

Safari Bookshelf is an electronic reference library that lets you easily search thousands of technical books, find code samples, download chapters, and access technical information whenever and wherever you need it.

To gain 45-day Safari Enabled access to this book:

- Go to informit.com/onlineedition
- Complete the brief registration form
- Enter the coupon code F1GR-7SFI-LPRP-Q2ID-HBCC

If you have difficulty registering on Safari Bookshelf or accessing the online edition, please e-mail customer-service@safaribooksonline.com.

Library of Congress Cataloging-in-Publication Data

Munshi, Aaftab.

The OpenGL ES 2.0 programming guide / Aaftab Munshi, Dan Ginsburg, Dave Shreiner.

p. cm.

Includes index.

ISBN-13: 978-0-321-50279-7 (pbk. : alk. paper)

ISBN-10: 0-321-50279-5 (pbk. : alk. paper) 1. OpenGL. 2. Computer graphics—Specifications. 3. Application program interfaces (Computer software) 4. Computer programming. I. Ginsburg, Dan. II. Shreiner, Dave. III. Title.

T385.M863 2009

006.6'6—dc22

2008016669

Copyright © 2009 by Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc., Rights and Contracts Department
501 Boylston Street, Suite 900, Boston, MA 02116, Fax: (617) 671-3447

ISBN-13: 978-0-321-50279-7

ISBN-10: 0-321-50279-5

Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.
First printing, July 2008

Foreword

Over the years, the “Red Book” has become the authoritative reference for each new version of the OpenGL API. Now we have the “Gold Book” for OpenGL ES 2.0—a cross-platform open standard ushering in a new era of shader programmability and visual sophistication for a wide variety of embedded and mobile devices, from game consoles to automobiles, from set top boxes to mobile phones.

Mobile phones, in particular, are impacting our everyday lives, as the devices we carry with us are evolving into full mobile computers. Soon we will be living in a world where most of us get our pixels delivered on these personal, portable devices—and OpenGL ES will be at the center of this handheld revolution. Devices such as the Apple iPhone already use OpenGL ES to drive their user interface, demonstrating how advanced graphics acceleration can play an important role in making a mobile phone fun, intuitive, and productive to use. But we have only just started the journey to make our handheld computers even more visually engaging. The shader programmability of the new generation of mobile graphics, combined with the portability and location awareness of mobile devices, will forever change how we interact with our phones, the Internet, and each other.

OpenGL ES 2.0 is a critical step forward in this mobile computing revolution. By bringing the power of the OpenGL ES Shading Language to diverse embedded and mobile platforms, OpenGL ES 2.0 unleashes enormous visual computing power, but in a way that is engineered to run on a small battery. Soon after this graphics capability is used to deliver extraordinary user interfaces, it will be leveraged for a wide diversity of visually engaging applications—compelling games, intuitive navigation applications, and more—all in the palm of your hand.

However, these applications will only be successful if enabled by a complete ecosystem of graphics APIs and authoring standards. This is the continuing mission of the Khronos Group—to bring together industry-leading companies and individuals to create open, royalty-free standards that enable the software community to effectively access the power of graphics and media acceleration silicon. OpenGL ES is at the center of this ecosystem, being developed alongside OpenGL and COLLADA. Together, they bring a tremendous cross-standard and multi-platform synergy to advanced 3D on a wide variety of platforms. Indeed, community collaboration has become essential for realizing the potential of OpenGL ES 2.0. The sophistication of a state-of-the-art programmable 3D API, complete with shading language, an effects framework, and authoring pipeline, has required hundreds of man years of design and investment—beyond any single company’s ability to create and evangelize throughout the industry.

As a result of the strong industry collaboration within Khronos, now is the perfect time to learn about this new programmable 3D API as OpenGL ES 2.0-capable devices will soon be appearing in increasing volumes. In fact, it is very possible that, due to the extraordinary volume of the mobile market, OpenGL ES 2.0 will soon be shipping on more devices than any previous 3D API to create an unprecedented opportunity for content developers.

This level of successful collaboration only happens as the result of hard work and dedication of many individuals, but in particular I extend a sincere thanks to Tom Olson, the working group chair that brought OpenGL ES 2.0 to market. And finally, a big thank you to the authors of this book: You have been central to the creation of OpenGL ES 2.0 within Khronos and you have created a great reference for OpenGL ES 2.0—truly worthy of the title “Gold Book.”

Neil Trevett
Vice President Mobile Content, NVIDIA
President, Khronos Group
April 2008

Preface

OpenGL ES 2.0 is a software interface for rendering sophisticated 3D graphics on handheld and embedded devices. OpenGL ES 2.0 is the primary graphics library for handheld and embedded devices with programmable 3D hardware including cell phones, PDAs, consoles, appliances, vehicles, and avionics. With OpenGL ES 2.0, the full programmability of shaders has made its way onto small and portable devices. This book details the entire OpenGL ES 2.0 API and pipeline with detailed examples in order to provide a guide for developing a wide range of high-performance 3D applications for handheld devices.

Intended Audience

This book is intended for programmers interested in learning OpenGL ES 2.0. We expect the reader to have a solid grounding in computer graphics. We will explain many of the relevant graphics concepts as they relate to various parts of OpenGL ES 2.0, but we do expect the reader to understand basic 3D concepts. The code examples in the book are all written in C. We assume that the reader is familiar with C or C++ and will only be covering language topics where they are relevant to OpenGL ES 2.0.

This book covers the entire OpenGL ES 2.0 API along with all Khronos-ratified extensions. The reader will learn about setting up and programming every aspect of the graphics pipeline. The book details how to write vertex and fragment shaders and how to implement advanced rendering techniques such as per-pixel lighting and particle systems. In addition, the book provides performance tips and tricks for efficient use of the API and hardware. After finishing the book, the reader will be ready to write OpenGL ES 2.0 applications that fully harness the programmable power of embedded graphics hardware.

Organization of the Book

This book is organized to cover the API in a sequential fashion, building up your knowledge of OpenGL ES 2.0 as we go.

Chapter 1—Introduction to OpenGL ES 2.0

This chapter gives an introduction to OpenGL ES, followed by an overview of the OpenGL ES 2.0 graphics pipeline. We discuss the philosophies and constraints that went into the design of OpenGL ES 2.0. Finally, the chapter covers some general conventions and types used in OpenGL ES 2.0.

Chapter 2—Hello Triangle: An OpenGL ES 2.0 Example

This chapter walks through a simple OpenGL ES 2.0 example program that draws a triangle. Our purpose here is to show what an OpenGL ES 2.0 program looks like, introduce the reader to some API concepts, and describe how to build and run an example OpenGL ES 2.0 program.

Chapter 3—An Introduction to EGL

This chapter presents EGL, the API for creating surfaces and rendering contexts for OpenGL ES 2.0. We describe how to communicate with the native windowing system, choose a configuration, and create EGL rendering contexts and surfaces. We teach you enough EGL so that you can do everything you will need to do to get up and rendering with OpenGL ES 2.0.

Chapter 4—Shaders and Programs

Shader objects and program objects form the most fundamental objects in OpenGL ES 2.0. In this chapter, we describe how to create a shader object, compile a shader, and check for compile errors. The chapter also covers how to create a program object, attach shader objects to it, and link a final program object. We discuss how to query the program object for information and how to load uniforms. In addition, you will learn about the difference between source and binary shaders and how to use each.

Chapter 5—OpenGL ES Shading Language

This chapter covers the shading language basics needed for writing shaders. The shading language basics described are variables and types, constructors, structures, arrays, attributes, uniforms, and varyings. This chapter also describes some more nuanced parts of the language such as precision qualifiers and invariance.

Chapter 6—Vertex Attributes, Vertex Arrays, and Buffer Objects

Starting with Chapter 6 (and ending with Chapter 11), we begin our walk through the pipeline to teach you how to set up and program each part of the graphics pipeline. This journey begins by covering how geometry is input into the graphics pipeline by discussing vertex attributes, vertex arrays, and buffer objects.

Chapter 7—Primitive Assembly and Rasterization

After discussing how geometry is input into the pipeline in the previous chapter, we then cover how that geometry is assembled into primitives. All of the primitive types available in OpenGL ES 2.0, including point sprites, lines, triangles, triangle strips, and triangle fans, are covered. In addition, we describe how coordinate transformations are performed on vertices and introduce the rasterization stage of the OpenGL ES 2.0 pipeline.

Chapter 8—Vertex Shaders

The next portion of the pipeline that is covered is the vertex shader. This chapter gives an overview of how vertex shaders fit into the pipeline and the special variables available to vertex shaders in the OpenGL ES Shading Language. Several examples of vertex shaders, including computation of per-vertex lighting and skinning, are covered. We also give examples of how the OpenGL ES 1.0 (and 1.1) fixed-function pipeline can be implemented using vertex shaders.

Chapter 9—Texturing

This chapter begins the introduction to the fragment shader by describing all of the texturing functionality available in OpenGL ES 2.0. This chapter covers all the details of how to create textures, how to load them with data,

and how to render with them. The chapter details texture wrap modes, texture filtering, and mipmapping. In addition, you will learn about the various functions for compressed texture images as well as how to copy texture data from the color buffer. This chapter also covers the optional texture extensions that add support for 3D textures and depth textures.

Chapter 10—Fragment Shaders

Chapter 9 focused on how to use textures in a fragment shader. This chapter covers the rest of what you need to know to write fragment shaders. We give an overview of fragment shaders and all of the special built-in variables available to them. We show how to implement all of the fixed-function techniques that were available in OpenGL ES 1.1 using fragment shaders. Examples of multitexturing, fog, alpha test, and user clip planes are all implemented in fragment shaders.

Chapter 11—Fragment Operations

This chapter discusses the operations that can be applied either to the entire framebuffer, or to individual fragments after the execution of the fragment shader in the OpenGL ES 2.0 fragment pipeline. These operations include scissor test, stencil test, depth test, multi-sampling, blending, and dithering. This is the final phase in the OpenGL ES 2.0 graphics pipeline.

Chapter 12—Framebuffer Objects

This chapter discusses the use of framebuffer objects for rendering to off-screen surfaces. There are several uses of framebuffer objects, the most common of which is for rendering to a texture. This chapter provides a complete overview of the framebuffer object portion of the API. Understanding framebuffer objects is critical for implementing many advanced effects such as reflections, shadow maps, and post-processing.

Chapter 13—Advanced Programming with OpenGL ES 2.0

This is the capstone chapter, tying together many of the topics presented throughout the book. We have selected a sampling of advanced rendering techniques and show examples that demonstrate how to implement these features. This chapter includes rendering techniques such as per-pixel lighting using normal maps, environment mapping, particle systems, image

post-processing, and projective texturing. This chapter attempts to show the reader how to tackle a variety of advanced rendering techniques.

Chapter 14—State Queries

There are a large number of state queries available in OpenGL ES 2.0. For just about everything you set, there is a corresponding way to get what the current value is. This chapter is provided as a reference for the various state queries available in OpenGL ES 2.0.

Chapter 15—OpenGL ES and EGL on Handheld Platforms

In the final chapter, we divert ourselves a bit from the details of the API to talk about programming with OpenGL ES 2.0 and EGL in the real world. There are a diverse set of handheld platforms in the market that pose some interesting issues and challenges when developing applications for OpenGL ES 2.0. We cover topics including an overview of handheld platforms, C++ portability issues, OpenKODE, and platform-specific shader binaries.

Appendix A—GL_HALF_FLOAT_OES

This appendix details the half-float format and provides a reference for how to convert from IEEE floating-point values into half-float (and back).

Appendix B—Built-In Functions

This appendix provides a reference for all of the built-in functions available in the OpenGL ES Shading Language.

Appendix C—Shading Language Grammar

This appendix provides a reference for OpenGL ES Shading Language grammar.

Appendix D—ES Framework API

This appendix provides a reference for the utility framework we developed for the book and describes what each function does.

Examples Code and Shaders

This book is filled with example programs and shaders. You can download the examples from the book Web site at www.opengles-book.com.

The examples are all targeted to run on Microsoft Windows XP or Vista with a desktop GPU supporting OpenGL 2.0. The example programs are provided in source code form with Microsoft Visual Studio 2005 project solutions. The examples build and run on the AMD OpenGL ES 2.0 Emulator. Several of the advanced shader examples in the book are implemented in RenderMonkey, a shader development tool from AMD. The book Web site provides links on where to download any of the required tools. The OpenGL ES 2.0 Emulator and RenderMonkey are both freely available tools. For readers who do not own Visual Studio, you can use the free Microsoft Visual Studio 2008 Express Edition available for download at www.microsoft.com/express/.

Errata

If you find something in the book which you believe is in error, please send us a note at errors@opengles-book.com. The list of errata for the book can be found on the book Web site at www.opengles-book.com.

Hello Triangle: An OpenGL ES 2.0 Example



To introduce the basic concepts of OpenGL ES 2.0, we begin with a simple example. In this chapter, we show what is required to create an OpenGL ES 2.0 program that draws a single triangle. The program we will write is just about the most basic example of an OpenGL ES 2.0 application that draws geometry. There are number of concepts that we cover in this chapter:

- Creating an on-screen render surface with EGL.
- Loading vertex and fragment shaders.
- Creating a program object, attaching vertex and fragment shaders, and linking a program object.
- Setting the viewport.
- Clearing the color buffer.
- Rendering a simple primitive.
- Making the contents of the color buffer visible in the EGL window surface.

As it turns out, there are quite a significant number of steps required before we can start drawing a triangle with OpenGL ES 2.0. This chapter goes over the basics of each of these steps. Later in the book, we fill in the details on each of these steps and further document the API. Our purpose here is to get you running your first simple example so that you get an idea of what goes into creating an application with OpenGL ES 2.0.

Code Framework

Throughout the book, we will be building up a library of utility functions that form a framework of useful functions for writing OpenGL ES 2.0 programs. In developing example programs for the book, we had several goals for this code framework:

1. It should be simple, small, and easy to understand. We wanted to focus our examples on the relevant OpenGL ES 2.0 calls and not on a large code framework that we invented. Rather, we focused our framework on simplicity and making the example programs easy to read and understand. The goal of the framework was to allow you to focus your attention on the important OpenGL ES 2.0 API concepts in each example.
2. It should be portable. Although we develop our example programs on Microsoft Windows, we wanted the sample programs to be easily portable to other operating systems and environments. In addition, we chose to use C as the language rather than C++ due to the differing limitations of C++ on many handheld platforms. We also avoid using global data, something that is also not allowed on many handheld platforms.

As we go through the examples in the book, we introduce any new code framework functions that we use. In addition, you can find full documentation for the code framework in Appendix D. Any functions you see in the example code that are called that begin with `es` (e.g., `esInitialize()`) are part of the code framework we wrote for the sample programs in this book.

Where to Download the Examples

You can download the examples from the book Web site at www.opengl-es-book.com.

The examples are all targeted to run on Microsoft Windows XP or Microsoft Windows Vista with a desktop graphics processing unit (GPU) supporting OpenGL 2.0. The example programs are provided in source code form with Microsoft Visual Studio 2005 project solutions. The examples build and run on the AMD OpenGL ES 2.0 emulator. Several of the advanced shader examples in the book are implemented in RenderMonkey, a shader development tool from AMD. The book Web site provides links on where to download any of the required tools. The OpenGL ES 2.0 emulator and RenderMonkey are both freely available tools. Readers who do not own Visual Studio can use the free Microsoft Visual Studio 2008 Express Edition available for download at www.microsoft.com/express/.

Hello Triangle Example

Let's take a look at the full source code for our Hello Triangle example program, which is listed in Example 2-1. For those readers familiar with fixed function desktop OpenGL, you will probably think this is a lot of code just to draw a simple triangle. For those of you not familiar with desktop OpenGL, you will also probably think this is a lot of code just to draw a triangle! Remember though, OpenGL ES 2.0 is fully shader based, which means you can't draw any geometry without having the appropriate shaders loaded and bound. This means there is more setup code required to render than there was in desktop OpenGL using fixed function processing.

Example 2-1 Hello Triangle Example

```
#include "esUtil.h"

typedef struct
{
    // Handle to a program object
    GLuint programObject;

} UserData;

///
// Create a shader object, load the shader source, and
// compile the shader.
//
GLuint LoadShader(const char *shaderSrc, GLenum type)
{
    GLuint shader;
    GLint compiled;

    // Create the shader object
    shader = glCreateShader(type);

    if(shader == 0)
        return 0;

    // Load the shader source
    glShaderSource(shader, 1, &shaderSrc, NULL);

    // Compile the shader
    glCompileShader(shader);

    // Check the compile status
    glGetShaderiv(shader, GL_COMPILE_STATUS, &compiled);
```

```

    if(!compiled)
    {
        GLint infoLen = 0;

        glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &infoLen);

        if(infoLen > 1)
        {
            char* infoLog = malloc(sizeof(char) * infoLen);

            glGetShaderInfoLog(shader, infoLen, NULL, infoLog);
            esLogMessage("Error compiling shader:\n%s\n", infoLog);
            free(infoLog);
        }

        glDeleteShader(shader);
        return 0;
    }

    return shader;
}

///
// Initialize the shader and program object
//
int Init(ESContext *esContext)
{
    UserData *userData = esContext->userData;
    GLbyte vShaderStr[] =
        "attribute vec4 vPosition;    \n"
        "void main()                    \n"
        "{                               \n"
        "    gl_Position = vPosition; \n"
        "}"                                  \n";

    GLbyte fShaderStr[] =
        "precision mediump float;        \n"
        "void main()                      \n"
        "{                               \n"
        "    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0); \n"
        "}"                                  \n";

    GLuint vertexShader;
    GLuint fragmentShader;
    GLuint programObject;
    GLint linked;

```

```

// Load the vertex/fragment shaders
vertexShader = LoadShader(GL_VERTEX_SHADER, vShaderStr);
fragmentShader = LoadShader(GL_FRAGMENT_SHADER, fShaderStr);

// Create the program object
programObject = glCreateProgram();

if(programObject == 0)
    return 0;

glAttachShader(programObject, vertexShader);
glAttachShader(programObject, fragmentShader);

// Bind vPosition to attribute 0
glBindAttribLocation(programObject, 0, "vPosition");

// Link the program
glLinkProgram(programObject);

// Check the link status
glGetProgramiv(programObject, GL_LINK_STATUS, &linked);

if(!linked)
{
    GLint infoLen = 0;

    glGetProgramiv(programObject, GL_INFO_LOG_LENGTH, &infoLen);

    if(infoLen > 1)
    {
        char* infoLog = malloc(sizeof(char) * infoLen);

        glGetProgramInfoLog(programObject, infoLen, NULL, infoLog);
        esLogMessage("Error linking program:\n%s\n", infoLog);

        free(infoLog);
    }

    glDeleteProgram(programObject);
    return FALSE;
}

// Store the program object
userData->programObject = programObject;

glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
return TRUE;
}

```

```

///
// Draw a triangle using the shader pair created in Init()
//
void Draw(ESContext *esContext)
{
    UserData *userData = esContext->userData;
    GLfloat vVertices[] = {0.0f,  0.5f, 0.0f,
                           -0.5f, -0.5f, 0.0f,
                           0.5f, -0.5f, 0.0f};

    // Set the viewport
    glViewport(0, 0, esContext->width, esContext->height);

    // Clear the color buffer
    glClear(GL_COLOR_BUFFER_BIT);

    // Use the program object
    glUseProgram(userData->programObject);

    // Load the vertex data
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, vVertices);
    glEnableVertexAttribArray(0);

    glDrawArrays(GL_TRIANGLES, 0, 3);

    eglSwapBuffers(esContext->eglDisplay, esContext->eglSurface);
}

int main(int argc, char *argv[])
{
    ESContext esContext;
    UserData  userData;

    esInitialize(&esContext);
    esContext.userData = &userData;

    esCreateWindow(&esContext, "Hello Triangle", 320, 240,
                  ES_WINDOW_RGB);

    if(!Init(&esContext))
        return 0;

    esRegisterDrawFunc(&esContext, Draw);

    esMainLoop(&esContext);
}

```

Building and Running the Examples

The example programs developed in this book all run on top of AMD's OpenGL ES 2.0 emulator. This emulator provides a Windows implementation of the EGL 1.3 and OpenGL ES 2.0 APIs. The standard GL2 and EGL header files provided by Khronos are used as an interface to the emulator. The emulator is a full implementation of OpenGL ES 2.0, which means that graphics code written on the emulator should port seamlessly to real devices. Note that the emulator requires that you have a desktop GPU with support for the desktop OpenGL 2.0 API.

We have designed the code framework to be portable to a variety of platforms. However, for the purposes of this book all of the examples are built using Microsoft Visual Studio 2005 with an implementation for Win32 on AMD's OpenGL ES 2.0 emulator. The OpenGL ES 2.0 examples are organized in the following directories:

`Common/`—Contains the OpenGL ES 2.0 Framework project, code, and the emulator.

`Chapter_X/`—Contains the example programs for each chapter. A Visual Studio 2005 solution file is provided for each project.

To build and run the Hello Triangle program used in this example, open `Chapter_2/Hello_Triangle/Hello_Triangle.sln` in Visual Studio 2005. The application can be built and run directly from the Visual Studio 2005 project. On running, you should see the image shown in Figure 2-1.



Figure 2-1 Hello Triangle Example

Note that in addition to providing sample programs, later in the book we provide several examples with a free shader development tool from AMD called RenderMonkey v1.80. RenderMonkey workspaces are used where we want to focus on just the shader code in an example. RenderMonkey provides a very flexible integrated development environment (IDE) for developing shader effects. The examples that have an `.rfx` extension can be viewed using RenderMonkey v1.80. A screenshot of the RenderMonkey IDE with an OpenGL ES 2.0 effect is shown in Color Plate 2.

Using the OpenGL ES 2.0 Framework

In the `main` function in Hello Triangle, you will see calls into several ES utility functions. The first thing the `main` function does is declare an `ESContext` and initialize it:

```
ESContext esContext;
UserData userData;

esInitialize(&esContext);
esContext.userData = &userData;
```

Every example program in this book does the same thing. The `ESContext` is passed into all of the ES framework utility functions and contains all of the necessary information about the program that the ES framework needs. The reason for passing around a context is that the sample programs and the ES code framework do not need to use any global data.

Many handheld platforms do not allow applications to declare global static data in their applications. Examples of platforms that do not allow this include BREW and Symbian. As such, we avoid declaring global data in either the sample programs or the code framework by passing a context between functions.

The `ESContext` has a member variable named `userData` that is a `void*`. Each of the sample programs will store any of the data that are needed for the application in `userData`. The `esInitialize` function is called by the sample program to initialize the context and the ES code framework. The other elements in the `ESContext` structure are described in the header file and are intended only to be read by the user application. Other data in the `ESContext` structure include information such as the window width and height, EGL context, and callback function pointers.

The rest of the `main` function is responsible for creating the window, initializing the draw callback function, and entering the main loop:

```
esCreateWindow(&esContext, "Hello Triangle", 320, 240,
              ES_WINDOW_RGB);

if(!Init(&esContext))
    return 0;

esRegisterDrawFunc(&esContext, Draw);

esMainLoop(&esContext);
```

The call to `esCreateWindow` creates a window of the specified width and height (in this case, 320×240). The last parameter is a bit field that specifies options for the window creation. In this case, we request an RGB framebuffer. In Chapter 3, “An Introduction to EGL,” we discuss what `esCreateWindow` does in more detail. This function uses EGL to create an on-screen render surface that is attached to a window. EGL is a platform-independent API for creating rendering surfaces and contexts. For now, we will simply say that this function creates a rendering surface and leave the details on how it works for the next chapter.

After calling `esCreateWindow`, the next thing the main function does is to call `Init` to initialize everything needed to run the program. Finally, it registers a callback function, `Draw`, that will be called to render the frame. The final call, `esMainLoop`, enters into the main message processing loop until the window is closed.

Creating a Simple Vertex and Fragment Shader

In OpenGL ES 2.0, nothing can be drawn unless a valid vertex and fragment shader have been loaded. In Chapter 1, “Introduction to OpenGL ES 2.0,” we covered the basics of the OpenGL ES 2.0 programmable pipeline. There you learned about the concepts of a vertex and fragment shader. These two shader programs describe the transformation of vertices and drawing of fragments. To do any rendering at all, an OpenGL ES 2.0 program must have both a vertex and fragment shader.

The biggest task that the `Init` function in Hello Triangle accomplishes is the loading of a vertex and fragment shader. The vertex shader that is given in the program is very simple:

```
GLbyte vShaderStr[] =
    "attribute vec4 vPosition;  \n"
    "void main()                \n"
    "{                          \n"
    "    gl_Position = vPosition; \n"
    "};                          \n";
```

This shader declares one input `attribute` that is a four-component vector named `vPosition`. Later on, the `Draw` function in `Hello Triangle` will send in positions for each vertex that will be placed in this variable. The shader declares a `main` function that marks the beginning of execution of the shader. The body of the shader is very simple; it copies the `vPosition` input attribute into a special output variable named `gl_Position`. Every vertex shader must output a position into the `gl_Position` variable. This variable defines the position that is passed through to the next stage in the pipeline. The topic of writing shaders is a large part of what we cover in this book, but for now we just want to give you a flavor of what a vertex shader looks like. In Chapter 5, “OpenGL ES Shading Language,” we cover the OpenGL ES shading language and in Chapter 8, “Vertex Shaders,” we specifically cover how to write vertex shaders.

The fragment shader in the example is also very simple:

```
GLbyte fShaderStr[] =
    "precision mediump float;    \n"
    "void main()                 \n"
    "{                          \n"
    "    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0); \n"
    "};                          \n";
```

The first statement in the fragment shader declares the default precision for float variables in the shader. For more details on this, please see the section on precision qualifiers in Chapter 5. For now, simply pay attention to the `main` function, which outputs a value of (1.0, 0.0, 0.0, 1.0) into the `gl_FragColor`. The `gl_FragColor` is a special built-in variable that contains the final output color for the fragment shader. In this case, the shader is outputting a color of red for all fragments. The details of developing fragment shaders are covered in Chapter 9, “Texturing,” and Chapter 10, “Fragment Shaders.” Again, here we are just showing you what a fragment shader looks like.

Typically, a game or application would not inline shader source strings in the way we have done in this example. In most real applications, the shader would be loaded from some sort of text or data file and then loaded to the API. However, for simplicity and having the example program be self-contained, we provide the shader source strings directly in the program code.

Compiling and Loading the Shaders

Now that we have the shader source code defined, we can go about loading the shaders to OpenGL ES. The `LoadShader` function in the Hello Triangle example is responsible for loading the shader source code, compiling it, and checking to make sure that there were no errors. It returns a *shader object*, which is an OpenGL ES 2.0 object that can later be used for attachment to a *program object* (these two objects are detailed in Chapter 4, “Shaders and Programs”).

Let’s take a look at how the `LoadShader` function works. The shader object is first created using `glCreateShader`, which creates a new shader object of the type specified.

```
GLuint LoadShader(GLenum type, const char *shaderSrc)
{
    GLuint shader;
    GLint compiled;

    // Create the shader object
    shader = glCreateShader(type);

    if(shader == 0)
        return 0;
```

The shader source code itself is loaded to the shader object using `glShaderSource`. The shader is then compiled using the `glCompileShader` function.

```
    // Load the shader source
    glShaderSource(shader, 1, &shaderSrc, NULL);

    // Compile the shader
    glCompileShader(shader);
```

After compiling the shader, the status of the compile is determined and any errors that were generated are printed out.

```
    // Check the compile status
    glGetShaderiv(shader, GL_COMPILE_STATUS, &compiled);

    if(!compiled)
    {
        GLint infoLen = 0;

        glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &infoLen);

        if(infoLen > 1)
```

```

    {
        char* infoLog = malloc(sizeof(char) * infoLen);

        glGetShaderInfoLog(shader, infoLen, NULL, infoLog);
        esLogMessage("Error compiling shader:\n%s\n", infoLog);

        free(infoLog);
    }

    glDeleteShader(shader);
    return 0;
}

return shader;
}

```

If the shader compiles successfully, a new shader object is returned that will be attached to the program later. The details of these shader object functions are covered in the first sections of Chapter 4.

Creating a Program Object and Linking the Shaders

Once the application has created a shader object for the vertex and fragment shader, it needs to create a program object. Conceptually, the program object can be thought of as the final linked program. Once each shader is compiled into a shader object, they must be attached to a program object and linked together before drawing.

The process of creating program objects and linking is fully described in Chapter 4. For now, we provide a brief overview of the process. The first step is to create the program object and attach the vertex shader and fragment shader to it.

```

// Create the program object
programObject = glCreateProgram();

if(programObject == 0)
    return 0;

glAttachShader(programObject, vertexShader);
glAttachShader(programObject, fragmentShader);

```

Once the two shaders have been attached, the next step the sample application does is to set the location for the vertex shader attribute `vPosition`:

```
// Bind vPosition to attribute 0
glBindAttribLocation(programObject, 0, "vPosition");
```

In Chapter 6, “Vertex Attributes, Vertex Arrays, and Buffer Objects,” we go into more detail on binding attributes. For now, note that the call to `glBindAttribLocation` binds the `vPosition` attribute declared in the vertex shader to location 0. Later, when we specify the vertex data, this location is used to specify the position.

Finally, we are ready to link the program and check for errors:

```
// Link the program
glLinkProgram(programObject);

// Check the link status
glGetProgramiv(programObject, GL_LINK_STATUS, &linked);

if(!linked)
{
    GLint infoLen = 0;

    glGetProgramiv(programObject, GL_INFO_LOG_LENGTH, &infoLen);

    if(infoLen > 1)
    {
        char* infoLog = malloc(sizeof(char) * infoLen);

        glGetProgramInfoLog(programObject, infoLen, NULL, infoLog);
        esLogMessage("Error linking program:\n%s\n", infoLog);

        free(infoLog);
    }

    glDeleteProgram(programObject);
    return FALSE;
}

// Store the program object
userData->programObject = programObject;
```

After all of these steps, we have finally compiled the shaders, checked for compile errors, created the program object, attached the shaders, linked the program, and checked for link errors. After successful linking of the program object, we can now finally use the program object for rendering! To use the program object for rendering, we bind it using `glUseProgram`.

```
// Use the program object
glUseProgram(userData->programObject);
```

After calling `glUseProgram` with the program object handle, all subsequent rendering will occur using the vertex and fragment shaders attached to the program object.

Setting the Viewport and Clearing the Color Buffer

Now that we have created a rendering surface with EGL and initialized and loaded shaders, we are ready to actually draw something. The `Draw` callback function draws the frame. The first command that we execute in `Draw` is `glViewport`, which informs OpenGL ES of the origin, width, and height of the 2D rendering surface that will be drawn to. In OpenGL ES, the viewport defines the 2D rectangle in which all OpenGL ES rendering operations will ultimately be displayed.

```
// Set the viewport
glViewport(0, 0, esContext->width, esContext->height);
```

The viewport is defined by an origin (x, y) and a width and height. We cover `glViewport` in more detail in Chapter 7, “Primitive Assembly and Rasterization,” when we discuss coordinate systems and clipping.

After setting the viewport, the next step is to clear the screen. In OpenGL ES, there are multiple types of buffers that are involved in drawing: color, depth, and stencil. We cover these buffers in more detail in Chapter 11, “Fragment Operations.” In the Hello Triangle example, only the color buffer is drawn to. At the beginning of each frame, we clear the color buffer using the `glClear` function.

```
// Clear the color buffer
glClear(GL_COLOR_BUFFER_BIT);
```

The buffer will be cleared to the color specified with `glClearColor`. In the example program at the end of `Init`, the clear color was set to (0.0, 0.0, 0.0, 1.0) so the screen is cleared to black. The clear color should be set by the application prior to calling `glClear` on the color buffer.

Loading the Geometry and Drawing a Primitive

Now that we have the color buffer cleared, viewport set, and program object loaded, we need to specify the geometry for the triangle. The vertices for the triangle are specified with three (x, y, z) coordinates in the `vVertices` array.

```
GLfloat vVertices[] = {0.0f,  0.5f, 0.0f,  
                      -0.5f, -0.5f, 0.0f,  
                      0.5f, -0.5f, 0.0f};  
  
...  
// Load the vertex data  
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, vVertices);  
glEnableVertexAttribArray(0);  
  
glDrawArrays(GL_TRIANGLES, 0, 3);
```

The vertex positions need to be loaded to the GL and connected to the `vPosition` attribute declared in the vertex shader. As you will remember, earlier we bound the `vPosition` variable to attribute location 0. Each attribute in the vertex shader has a location that is uniquely identified by an unsigned integer value. To load the data into vertex attribute 0, we call the `glVertexAttribPointer` function. In Chapter 6, we cover how to load vertex attributes and use vertex arrays in full.

The final step to drawing the triangle is to actually tell OpenGL ES to draw the primitive. That is done in this example using the function `glDrawArrays`. This function draws a primitive such as a triangle, line, or strip. We get into primitives in much more detail in Chapter 7.

Displaying the Back Buffer

We have finally gotten to the point where our triangle has been drawn into the framebuffer. There is one final detail we must address: how to actually display the framebuffer on the screen. Before we get into that, let's back up a little bit and discuss the concept of double buffering.

The framebuffer that is visible on the screen is represented by a two-dimensional array of pixel data. One possible way one could think about displaying images on the screen is to simply update the pixel data in the visible framebuffer as we draw. However, there is a significant issue with updating pixels directly on the displayable buffer. That is, in a typical display system, the physical screen is updated from framebuffer memory at a fixed rate. If

one were to draw directly into the framebuffer, the user could see artifacts as partial updates to the framebuffer where displayed.

To address this problem, a system known as double buffering is used. In this scheme, there are two buffers: a front buffer and back buffer. All rendering occurs to the back buffer, which is located in an area of memory that is not visible to the screen. When all rendering is complete, this buffer is “swapped” with the front buffer (or visible buffer). The front buffer then becomes the back buffer for the next frame.

Using this technique, we do not display a visible surface until all rendering is complete for a frame. The way this is all controlled in an OpenGL ES application is through EGL. This is done using an EGL function called `eglSwapBuffers`:

```
eglSwapBuffers(esContext->eglDisplay, esContext->eglSurface);
```

This function informs EGL to swap the front buffer and back buffer. The parameters sent to `eglSwapBuffers` are the EGL display and surface. These two parameters represent the physical display and the rendering surface, respectively. In the next chapter, we explain `eglSwapBuffers` in more detail and further clarify the concepts of surface, context, and buffer management. For now, suffice to say that after swapping buffers we now finally have our triangle on screen!

Index

Symbols and Numbers

- “.” operator, 81
- “[]” operator, 81, 82
- 2D images, 255
- 2D rectangle, 32
- 2D slice, 312–313
- 2D textures, 182–183, 196, 264–265
- 3D noise, 308, 311–312
- 3D texture(s)
 - attaching to a framebuffer, 266
 - command to load, 207
 - described, 207, 307
 - generating noise, 308
 - noise using, 307–315
 - support for, 207–213
 - updating subregions of, 210–211
- 3D texture extension, 209, 314
- 3D vector, 183–184
- 16-bit floating-point number, 354–355
- 16-bit floating-point vertex data
 - attributes, 353

A

- absolute error, 354
- abstraction layers, 343
- active attributes, 72
- active program, 66
- active uniforms, 67–69
- active vertex attributes, 112
- ADD RGB combine function, 217

- ADD_SIGNED RGB combine function, 217
- Advanced RenderMan: Creating CGI for Motion Pictures*, 319
- aliasing, 249, 319
- aliasing artifacts
 - of procedural textures, 316
 - reducing, 190
 - resolving, 189
- alpha test
 - implementing, 227–229
 - no longer needed, 10
- alpha value, 250
- ambient color, 162, 285
- AMD OpenGL ES 2.0 emulator, 20
- angle and trigonometry functions, 358–359
- antialiasing
 - multisampled, 249–250
 - of procedural textures, 319–322
- applications
 - modifying parameters, 327–330
 - uses of EGL, 12
- ARM processor family, 340
- array(s), 83–84
- array buffer objects, 116
- array indexing, of uniforms, 156–157
- array of structures, 104, 105–106
 - as most efficient for OpenGL ES 2.0, 107
 - storing vertex attributes, 122

- array subscripting, 81
- artifacts
 - avoiding, 143, 144
 - multisampling prone to, 250
 - produced by nearest sampling, 189
- attachment points, 262
- attachments
 - to a framebuffer, 338
 - minimum of one valid, 268
- attenuation, 163
- attribute(s)
 - associated with an `EGLConfig`, 39
 - binding, 31
 - getting and setting, 72
 - in the OpenGL ES Shading Language, 89–90
 - for a vertex shader, 4, 5, 148, 149
- attribute index, 113
- attribute qualifier, 110
- attribute variable name, 115
- attribute variables, 89
- automatic mipmap generation, 193–194

B

- `b` argument data type, 15
- back buffer, 34, 234
- back-facing triangles, 142
- backward compatibility, of OpenGL ES 2.0 and OpenGL ES 1.x, 11–12
- backward differencing, 372
- bias matrix, 303, 304
- bias parameter, 369
- bilinear filtering, 193
- binary compilation tools, 351
- binary fragment shader, 75
- binary operators, 84
- binary shader format, 74, 351
- binary shaders
 - loading, 74–75
 - supporting, 73
- binary vertex shader, 75
- `BinaryShader.lib`, 351
- binding, 115
- bit test, stencil test as, 240
- bitmask, specifying, 54
- bits, in color buffer, 43

- blending
 - enabling, 333
 - in per-fragment operations, 10
 - pixel colors, 246–248
- blending coefficients, 247–248
- blending equation, 246, 248
- Bloom effect, 298
- blur fragment shader, 297–298
- Boolean-based vector types, 79
- bottom clip plane, 138
- bound texture objects, 193
- box filtering technique, 189
- BREW operating system, 339
- buffer(s), 234–235
 - clearing, 235–236
 - involved in drawing, 32
 - requesting additional, 235
 - size of, 234
 - swapping of, 234
 - types of, 234
 - unmapping previously mapped, 125
- buffer object data store, 119
- buffer object names, 117
- buffer objects
 - deleting, 124
 - for each vertex attribute, 122–123
 - mapping, 124–126
 - state associated with, 118
 - types supported, 116
 - usage of, 118
- buffer write mask, 236
- built-in constants, 150, 220–221
- built-in functions, 357–373
 - in applications, 358
 - categories of, 357
 - in the OpenGL ES Shading Language, 86–87
 - overloading, 358
- built-in special variables, 149–150, 219–220
- built-in uniform state, 150
- built-in variables, of a vertex shader, 149

C

- C code, syntax compared to shaders, 78
- C compiler, generating object code, 57–58

C language, compared to C++, 20
C linker, 58
C++
 exceptions, 341
 features to avoid, 342–343
 portability, 341–343
 preprocessor, 92
Carbide, 340
centroid sampling, 250
CheckerAA.rfx RenderMonkey
 workspace, 319
checkerboard pattern, rendering,
 316
checkerboard texture
 fragment shader implementing,
 317–319
 vertex shader implementing,
 316–317
Checker.rfx RenderMonkey workspace,
 316
cleanup stack, 342
clear color, setting, 32
clear value, specifying, 236
client, delaying execution of, 55
client data store, 119
client space, 103
client-server model, 16–17
client-side vertex arrays, 295
clip coordinate, 138
clip coordinate space, 137, 138
clip planes, 229
clip space, 173
clip volume, 138
clipping, 7, 149
clipping operations, 139
clipping planes, 138
clipping stage, 138–139
clockwise (CW) orientation, 142
code, as inputs to the fragment shader,
 218, 219
code framework, 20
code paths, for both source and binary
 shaders, 75
color(s)
 attenuating the final, 307
 combining diffuse and specular, 163
 mapping texture formats, 197–198
 color attachment point, 262
 color attachments, as valid, 267
 color buffer
 clearing, 32
 copying texture data from, 204–207
 depth of, 234
 double buffered, 234
 updating, 237
 color depth, 249
 color value, 8
 color-renderable buffer, 261
 color-renderable formats, 267
 command flavors, examples of, 14
 commands
 buffering on the client side, 17
 multiple flavors of, 15
 in OpenGL ES, 14
 taking arguments in different
 flavors, 14
 common functions, built-in, 361–364
 communications channel, opening,
 36
 comparison operators, 84, 85
 compatibility, maintaining with Open
 GL, 2
 compile errors, 60
 compiling, shaders, 59–60
 compressed 2D texture image, 203
 compressed 3D texture data, 209
 compressed image datatypes, 201
 compressed textures, 201–202
 The Compressorator, 213
 computeLinearFogFactor function,
 227
 condition expression, 154
 conditional checks, 318
 conditional statements, 155–156
 conditional tests, performing, 92
 conditionals, expressing, 87
 configurations
 nonconformant, 42
 number of available, 38
 slow rendering, 42
 sorting, 42–43
 conformance tests, 2
 const qualifier, 82
 const variables, 157

-
- constant(s)
 - built-in, 150, 220–221
 - implementation-specific, 157
 - in the OpenGL ES Shading Language, 82
 - constant color
 - in a fixed function pipeline, 217
 - setting, 248
 - constant integral expressions, 156, 221
 - constant store, 89, 94
 - constant variables, 82
 - constant vertex attribute, 102–103, 109–110
 - constructors, 79–80
 - context
 - creating, 50, 51
 - passing between functions, 26
 - control flow statements, 87–88
 - coordinate systems, 137
 - coordinates, 2D pair of, 182, 183
 - coplanar polygons, 144
 - core functions, in ES framework API, 385–390
 - counterclockwise (CCW) orientation, 142
 - coverage mask, 249, 250
 - coverage value, 250
 - CPUs, for handheld device, 340
 - cube(s)
 - drawing, 132–134
 - geometry for, 389
 - cubemap(s), 167, 183
 - drawing a sphere with a simple, 198–199
 - environment mapping using, 286–289
 - specifying faces of, 183–184
 - using `glGenerateMipmap`, 193
 - cubemap textures, 183–184, 198–201
 - culling operation
 - discarding primitives, 7
 - enabling, 142–143
 - cycles, for procedural texture, 316
- D**
- data elements, in vertex attributes
 - arrays, 335–336
 - data types, 78–79
 - debug output, 390
 - debugging, 66
 - default precision, 96, 221
 - default precision qualifier, 96, 152
 - degenerate triangles, 134, 135
 - depth attachment, as valid, 267
 - depth attachment point, 262
 - depth buffer(s)
 - allocating, 54
 - bit depth of, 234
 - disabling writing, 237
 - including, 235
 - testing, 245–246
 - writing to, 237
 - depth comparison operator, 246
 - depth offset, computing, 145
 - depth range values, 140–141
 - depth renderable buffer, 261
 - depth renderbuffers, sharing, 278
 - depth test, enabling, 245, 333
 - depth-renderable formats, 267
 - `depthRenderbuffer` object, binding, 274
 - derivative computation, 371
 - derivative functions, built-in, 371–373
 - derivatives, 371
 - desktop GPU, 25
 - desktop OpenGL, 21
 - destination alpha buffer, 54
 - diffuse color, 162, 285
 - directional light, 161, 162–163
 - directives, 92–93
 - directories, for examples, 25
 - DirectX, 1
 - `disable` extension behavior, 93
 - `discard` keyword, 227, 229
 - displays, querying and initializing, 12
 - distance to the eye, 225–226
 - dithering, 10, 249, 333
 - divergent flow control, 172
 - `dot` built-in function, 86
 - `DOT3_RGB` combine function, 217
 - double buffering, 33–34
 - double-buffered applications, 234
 - double-buffered color buffer, 234
 - double-buffered surfaces, 257

- do-while loops, 155
- draw buffers, 220
- Draw callback function, 27, 32, 387
- draw distances, reducing, 224
- Draw function, for particle system sample, 294–296
- draw surface, 12
- drawing
 - cubes, 132–134
 - primitives, 131–136
- drawing commands, 6
- drawing surfaces, 35, 253, 254
- dynamic indexing, on a vector, 81

E

- EGL, 12
 - API, 13
 - attributes, 42
 - choosing `EGLConfig`, 39, 42–43
 - commands, 14
 - context, 51
 - creating an on-screen render surface, 27
 - creating pixel buffers, 48–50
 - data types, 14
 - defined, 27
 - display server, 36
 - header file, 13
 - initializing, 36, 37
 - introduction to, 35–55
 - library, 13
 - mechanisms provided by, 35
 - pixel buffer attributes, 47
 - prefix, 14
 - processing and reporting errors, 37
 - specification, 13
 - specifying multisample buffers, 257
 - surface configurations, 38, 42
 - window, 43–46, 52–54
- `EGL_ALPHA_MASK_SIZE` attribute, 40, 43
- `EGL_ALPHA_SIZE` attribute, 40
- `EGL_BACK_BUFFER`, 44
- `EGL_BAD_ALLOC` error, 45, 48
- `EGL_BAD_ATTRIBUTE` error, 48
- `EGL_BAD_CONFIG` error, 45, 48, 51
- `EGL_BAD_CURRENT_SURFACE` error, 55

- `EGL_BAD_MATCH` error, 45, 48
- `EGL_BAD_NATIVE_WINDOW` error code, 45
- `EGL_BAD_PARAMETER` error, 48
- `EGL_BIND_TO_TEXTURE_RGB` attribute, 40
- `EGL_BIND_TO_TEXTURE_RGBA` attribute, 40
- `EGL_BLUE_SIZE` attribute, 40
- `EGL_BUFFER_SIZE` attribute, 40, 43
- `EGL_COLOR_BUFFER_TYPE` attribute, 40, 43
- `EGL_CONFIG_CAVEAT` attribute, 40, 42
- `EGL_CONFIG_ID` attribute, 40, 43
- `EGL_CONFORMANT` attribute, 40
- `EGL_CONTEXT_CLIENT_VERSION` attribute, 50–51
- `EGL_CORE_NATIVE_ENGINE`, 55
- `EGL_DEFAULT_DISPLAY` token, 36
- `EGL_DEPTH_SIZE` attribute, 40, 43, 235
- `EGL_FRONT_BUFFER`, 44
- `EGL_GREEN_SIZE` attribute, 40
- `EGL_HEIGHT` attribute, 47
- `EGL_LARGEST_PBUFFER` attribute, 47
- `EGL_LEVEL` attribute, 40
- `EGL_LUMINANCE_BUFFER`, 43
- `EGL_LUMINANCE_SIZE` attribute, 40
- `EGL_MAX_PBUFFER_HEIGHT` attribute, 40
- `EGL_MAX_PBUFFER_PIXELS` attribute, 41
- `EGL_MAX_PBUFFER_WIDTH` attribute, 40
- `EGL_MAX_SWAP_INTERVAL` attribute, 41
- `EGL_MIN_SWAP_INTERVAL` attribute, 41
- `EGL_MIPMAP_TEXTURE` attribute, 47
- `EGL_NATIVE_RENDERABLE` attribute, 41
- `EGL_NATIVE_VISUAL_ID` attribute, 41
- `EGL_NATIVE_VISUAL_TYPE` attribute, 41, 43
- `EGL_NO_CONTEXT`, 51
- `EGL_NO_SURFACE`, 44
- `EGL_RED_SIZE` attribute, 40
- `EGL_RENDER_BUFFER` attribute, 44
- `EGL_RENDERABLE_TYPE` attribute, 41
- `EGL_RGB_BUFFER`, 43
- `EGL_SAMPLE_BUFFERS` attribute, 41, 43
- `EGL_SAMPLES` attribute, 41, 43

EGL_STENCIL_SIZE attribute, 41, 43, 235
 EGL_SURFACE_TYPE attribute, 41
 EGL_TEXTURE_FORMAT attribute, 47
 EGL_TEXTURE_TARGET attribute, 47
 EGL_TRANSPARENT_BLUE_VALUE attribute, 41
 EGL_TRANSPARENT_GREEN_VALUE attribute, 41
 EGL_TRANSPARENT_RED_VALUE attribute, 41
 EGL_TRANSPARENT_TYPE attribute, 41
 EGL_WIDTH attribute, 47
 EGLBoolean data type, 14
 eglChooseConfig, 39, 42–43
 EGLClientBuffer data type, 14
 EGLConfig
 attributes, 39, 40–41
 data type, 14
 errors, 51
 returning, 38
 values, 38, 39
 EGLContext data type, 14
 EGLContexts, 50, 52
 eglCreateContext function, 50–51
 eglCreatePbufferSurface function, 47–48
 eglCreateWindowSurface function
 calling, 43–44
 EGLNativeWindowType win argument, 141
 possible errors, 45
 EGLDisplay data type, 14, 36
 EGLenum data type, 14
 eglGetConfigAttribute function, 38, 39
 eglGetConfigs, 38
 eglGetDisplay, 36–37
 eglGetError, 37, 44–45, 47–48
 egl.h file, 13
 eglInitialize, 37
 EGLint data type, 14
 eglmakeCurrent function, 52
 EGLNativeDisplayType, 36
 EGLSurface data type, 14
 eglSwapBuffers function
 controlling rendering, 34
 swapping the display buffer, 277
 eglWaitClient function, 55
 eglWaitNative function, 55
 element array buffer objects, 116
 element array, sorting, 295–296
 element indices, 134
 #elif directive, 92
 #else directive, 92
 Embedded Linux, 340, 341
 emulator, OpenGL ES 2.0, 25
 enable extension behavior, 93
 enable tokens, 239
 end position, of a particle, 293
 engines, for rendering, 55
 entities, name queries, 331
 environment mapping, 183
 basics of, 289
 example, 286
 fragment shader, 288–289
 using a cubemap, 286–289
 EnvironmentMapping.rfx, 286
 equations
 combining fixed function pipeline inputs, 217
 lighting in the fragment shader, 285
 specifying planes, 229
 Ericsson Texture Compression (ETC), 201, 213
 error code recovery, 37
 error codes, 16
 #error directive, 92
 error handling, 15–16
 errors, checking for in EGL, 37
 ES 1.1 specification, 2
 ES code framework, 26
 ES framework API functions, 385–394
 ES framework context, 385
 es prefix, 20
 ESContext structure, 26
 esCreateWindow function, 27, 54, 386
 esFrustum function, 390–391
 esGenCube function, 389
 esGenSphere function, 388
 esInitContext function, 385
 esInitialize function, 26
 esLoadProgram function, 388
 esLoadShader function, 387

esLoadTGA function, 388
 esLogMessage function, 390
 esMainLoop function, 386
 ESMatrix type, 390–391
 esMatrixLoadIdentity function, 394
 esMatrixMultiply function, 394
 esOrtho function, 392
 esPerspective function, 391
 esRegisterDrawFunc function, 387
 esRegisterKeyFunc function, 387
 esRegisterUpdateFunc function, 387
 esRotate function, 393
 esScale function, 392
 esTranslate function, 393
 esUtil convenience library, 235
 ESUtil Library, 54
 ETC format, 213
 even-numbered triangle, 134
 example programs, building and running, 25–26
 examples, downloading, 20
 exceptions, avoiding in C++, 342
 executable programs, generating, 64
 exponential fog, 227
 exponential functions, built-in, 360
 extension(s)
 available, 323, 324
 behavior of, 93
 optional, 207–214
 specific to a single vendor, 352
 targeting, 352
 #extension directive, 93
 #extension GL_OES_standard_derivatives directive, 371
 #extension GL_OES_texture_3d directive, 371
 #extension mechanism, 209, 314, 352
 extension strings, 352
 external memory, going to, 277
 eye space, transforming to, 173

F

f argument data type, 15
 facedness, of a primitive, 238
 faces, of cubemaps, 183
 fallback paths, 352
 far clip plane, 138
 FBO texture, binding, 297
 FBOs. *See* framebuffer objects
 features, usage of, 332
 fence, representing, 227
 filter width, 373
 filtering
 algorithm, 193–194
 modes, 188, 190–191
 fixed function fragment pipeline, 216–218
 fixed function pipeline, 2
 configuring to perform a modulate, 218
 implemented by OpenGL ES 1.1, 102
 not mixed with programmable, 11
 fixed function techniques, 216–218, 222–231
 fixed function texture combine units, 11
 fixed function vertex pipeline, 173
 fixed function vertex units, 11
 fixed resolution, 315
 float, precision, 152
 float variables, precision for, 28
 float variant, 103
 floating-point based matrices, 79
 floating-point numbers, converting, 355–356
 floating-point textures, 213
 floating-point variable, precision for, 96
 floating-point-based vector types, 78
 flow control
 divergent, 172
 in the vertex shader, 153–155
 flushing, pending commands, 17
 fog
 implementing using shaders, 224–227
 noise-distorted, 313–315
 fog factor, computing, 173
 for loops
 examples of valid constructs, 154–155
 supporting in a vertex shader, 153
 writing simple, 87

format conversions, for texture copy, 206–207

forward-backward differencing, 371–372

fragment(s)

- applying tests to, 238–246
- with combined stencil and depth tests, 241
- generating for a primitive, 141
- operating on, 7–9

fragment shader(s)

- for alpha test using discard, 228–229
- antialiased checker, 319–321
- attaching to a program object, 30
- in the blurring example, 297–298
- built-in constants for, 220–221
- built-in special variables, 219–220
- computing noise in, 308
- creating, 27–28, 58
- declaring a uniform variable, 196
- default float precision, 97
- for the environment mapping sample, 288–289
- example of, 8–9
- fixed function techniques, 216–218
- in Hello Triangle, 28
- implementing checkerboard texture, 317–319
- inputs to, 7–8
- limitations on, 221
- multistruce example, 222–223
- for noise distorted fog, 313–315
- in OpenGL ES 2.0, 7–9
- output of, 219
- overview, 218–221
- for the particle system, 293–294
- performing fixed function operations, 218
- per-fragment lighting, 283–285
- for projective texturing, 306–307
- rendering linear fog, 226–227
- textures in, 196–198
- user clip plane, 230–231

fragment shader object, 58

framebuffer(s)

- copying to the texture buffer, 254
- depths, 234
- displaying the screen, 33
- as incomplete, 267–268
- masks controlling writing to, 236–238
- reading and writing pixels to, 250–252
- reading back pixels from, 10
- state queries, 337–338
- window system provided, 253

framebuffer attachment(s)

- attaching a 2D texture as, 264–265
- attaching a renderbuffer, 263–264
- attaching an image of a 3D texture as, 266–267
- deleting renderbuffer objects used as, 270
- depth texture used as, 274
- renderbuffer objects versus texture objects, 256–257

framebuffer completeness

- checking for, 267–269
- status, 262

framebuffer names, allocating, 258

Framebuffer objects API, 255

framebuffer objects (FBOs), 255

- binding, 263, 274
- creating, 258
- defining as complete, 267
- deleting, 270
- versus EGL surfaces, 257
- examples, 271–277
- performance tips, 277–278
- for reading and writing pixels, 269
- relationship with renderbuffer objects and textures, 256
- rendering into an off-screen, 297
- rendering to a texture, 46, 194, 271–273
- rendering with, 254
- setting current, 262, 263
- sharing stencil and depth buffers, 257
- states associated with, 262
- supporting only single-buffered attachments, 257
- using, 262–263
- verifying as complete, 268–269

frames, drawing, 32
fresnel term, 289
front buffer, 34, 234
front-facing triangles, 142
full-screen quad, 297
functions. *See also specific functions*
 built-in, 357–373
 in the OpenGL ES Shading
 Language, 85–86
 performed during per-fragment
 operations, 10
fwidth function, 321

G

gl prefix, 14
GenMipMap2D function, 189
geometric functions, built-in, 264–366
geometric primitive, rendering, 249
geometry
 binding and loading, 186–188
 for a cube, 389
 for a primitive, 6
 removing "popping" of, 224
 for a sphere, 388
 for a triangle, 33
get functions, 327
GL prefix, 14
GL_ACTIVE_ATTRIBS, 64
GL_ACTIVE_ATTRIBUTES query, 72
GL_ACTIVE_UNIFORM_MAX_LENGTH,
 64–65
GL_ACTIVE_UNIFORMS, 64–65
GL_ALPHA format, 182, 198
GL_ARRAY_BUFFER, 116, 120
GL_ATTACHED_SHADERS, 65
GL_BLEND token, 239
GL_BUFFER_SIZE, 118
GL_BUFFER_USAGE, 118
GL_CLAMP_TO_EDGE mode, 194, 195
GL_COMPRESSED_TEXTURE_FORMATS, 202
GL_CULL_FACE state, 143
GL DECR stencil function, 241
GL DECR_WRAP stencil function, 241
GL_DELETE_STATUS, 65
GL_DEPTH_TEST token, 239
gl_DepthRange uniform state, 150
GL_DITHER token, 239

GL_DYNAMIC_DRAW value, 118
GL_ELEMENT_ARRAY_BUFFER token, 116
GL_EXTENSIONS string, 352
gl_FragColor, 8, 28, 219
gl_FragCoord, 219–220
GL_FRAGMENT_PRECISION_HIGH
 preprocessor macro, 97, 221
gl_FrontFacing special variable, 150,
 220
GL_FUNC_ADD operator, 248
GL_FUNC_REVERSE_SUBTRACT, 248
GL_FUNC_SUBTRACT operator, 248
GL_HALF_FLOAT_OES vertex data type,
 107, 353–356
GL_INCR stencil function, 241
GL_INCR_WRAP stencil function, 241
GL_INFO_LOG_LENGTH, 61, 65
GL_INVALID_ENUM error, 16, 17, 18, 202
GL_INVALID_FRAMEBUFFER_OPERATION
 error, 269
GL_INVALID_OPERATION error, 16
GL_INVALID_VALUE error, 16
GL_INVERT stencil function, 241
GL_KEEP stencil function, 241
GL_LINE_LOOP, 129
GL_LINE_STRIP, 129
GL_LINEAR magnification filtering, 191
GL_LINEAR texture minification mode,
 192
GL_LINEAR_MIPMAP_LINEAR texture
 minification mode, 192
GL_LINEAR_MIPMAP_NEAREST texture
 minification mode, 192
GL_LINES, 129
GL_LINK_STATUS, 64
GL_LUMINANCE format, 182, 198
GL_LUMINANCE_ALPHA format, 182, 198
GL_MAX_TEXTURE_IMAGE_UNITS
 parameter, 197
GL_MAX_VERTEX_ATTRIBS vec4 vertex
 attributes, 111
gl_MaxCombinedTextureImageUnits
 built-in constant, 151
gl_MaxDrawBuffers constant,
 220–221
gl_MaxFragmentUniformVectors
 constant, 220

`gl_MaxVaryingVectors` built-in constant, 151
`gl_MaxVertexAttribs` built-in constant, 150
`gl_MaxVertexTextureImageUnits` built-in constant, 151
`gl_MaxVertexUniformVectors`, 151, 157, 169
`GL_MIRRORED_REPEAT` mode, 194, 195
`GL_NEAREST`, 191, 192
`GL_NEAREST_MIPMAP_LINEAR` textured minification mode, 192
`GL_NEAREST_MIPMAP_NEAREST` textured minification mode, 192
`GL_NO_ERROR` error code, 15, 16
`GL_NUM_COMPRESSED_TEXTURE_FORMATS`, 202
`GL_OES_compressed_ETC1_RGB8_texture`, 213
`GL_OES_depth_texture` optional extension, 256
`GL_OES_packed_depth_stencil` optional extension, 256
`GL_OES_standard_derivatives` extension, 319, 321
`GL_OES_texture_3D` extension, 207, 256, 314
`GL_OES_texture_float` extension, 213
`GL_OES_texture_float_linear` extension, 213
`GL_OES_texture_half_float` extension, 213
`GL_OES_texture_half_float_linear` extension, 213
`GL_OES_texture_npot` extension, 214
`GL_OES_vertex_half_float` extension string, 353
`GL_OUT_OF_MEMORY` error, 15, 16
`GL_PACK_ALIGNMENT` argument, 188
`gl_PointCoord` variable, 130–131, 220
`GL_POINTS`, 130
`gl_PointSize` variable, 130, 150, 293
`gl_Position` variable, 28, 149
`GL_REFLECTION_MAP` mode, 167
`GL_REPEAT` mode, 194, 195
`GL_REPLACE` stencil function, 241
`GL_RGB` format, 182, 198
`GL_RGBA` format, 182, 198
`GL_SAMPLE_ALPHA_TO_COVERAGE` token, 239
`GL_SAMPLE_COVERAGE` token, 239, 250
`GL_SHADER_COMPILER`, 73
`GL_SHORT`, 353, 354
`GL_SPHERE_MAP` mode, 167
`GL_SRC_ALPHA_SATURATE`, 247, 248
`GL_STATIC_DRAW` value, 118
`GL_STENCIL_TEST` token, 239
`GL_STREAM_DRAW` value, 118
`GL_TEXTURE_WRAP_R_OES`, 209
`GL_TEXTURE_WRAP_S` mode, 194
`GL_TEXTURE_WRAP_T` mode, 194
`gl_TextureImageUnits` constant, 220
`GL_TRIANGLE_FAN`, 128
`GL_TRIANGLE_STRIP`, 128
`GL_TRIANGLES`, 128
`GL_UNSIGNED_BYTE`, 107
`GL_UNSIGNED_SHORT`, 353
`GL_VALIDATE_STATUS`, 65, 66
`GL_ZERO` stencil function, 241
`gl2ext.h` file, 13
`gl2.h` file, 13
`glActiveTexture` function, 196–197
`glAttachShader` function, 63
`glBindAttribLocation` command, 31, 113, 115
`glBindBuffer` command, 117–118
`glBindFramebuffer` command, 262, 274
`glBindRenderbuffer` command, 259, 260
`glBindTexture` function, 185, 188
`GLbitfield` type, 15
`glBlendColor`, 248
`glBlendEquation`, 248
`glBlendEquationSeparate`, 248
`glBlendFunc`, 246
`glBlendFuncSeparate`, 246
`GLboolean` type, 15
`glBufferData` command, 119, 126
`glBufferSubData` command, 119
`GLbyte` type, 15
`glCheckFramebufferStatus` command, 268–269

GLclampf type, 15
 glClear function, 32, 235–236
 glClearColor function, 32, 236
 glClearDepthf function, 236
 glClearStencil function, 236
 glColorMask routine, 237
 glCompileShader function, 29, 59–60
 glCompressedTexImage2D function, 201–202, 213
 glCompressedTexImage3DOES function, 209–210
 glCompressedTexSubImage2D function, 203–204
 glCompressedTexSubImage3DOES function, 211–212
 glCopyTexImage2D function, 205, 206, 254
 glCopyTexSubImage2D function, 206–207, 213, 254
 glCopyTexSubImage3DOES function, 212–213
 glCreateProgram function, 62
 glCreateShader function, 29, 58
 glCullFace, 143
 glDeleteBuffers command, 124
 glDeleteFramebuffers API, 270
 glDeleteProgram function, 31, 62–63
 glDeleteRenderbuffers, 269, 270
 glDeleteShader function, 58–59
 glDeleteTextures function, 185
 glDepthFunc, 246
 glDepthMask, 237
 glDepthRange, 140–141
 glDetachShader function, 63
 glDisable function, 17, 143, 145, 332–333
 glDisableVertexAttribArray command, 109
 glDrawArrays function, 33, 131–133, 134
 glDrawElements function, 132, 133–136
 glEnable function, 17, 143, 145, 238–239, 332–333
 glEnableVertexAttribArray command, 109
 GLenum
 argument, 16
 type, 15
 values, 202
 glFinish command, 17, 55
 GLfixed type, 15
 GLfloat type, 15
 glFlush command, 17
 glFramebufferRenderbuffer command, 263–264
 glFramebufferTexture2D command, 264–265
 glFramebufferTexture3DOES command, 266
 glFrontFace, 142–143
 glGenBuffers command, 117, 118
 glGenFramebuffers, 258, 262
 glGenRenderbuffers, 258, 259
 glGenTextures function, 184, 188
 glGetActiveAttrib command, 72, 112–113
 glGetActiveUniform function, 68–69
 glGetAttachedShaders function, 333
 glGetAttribLocation command, 115
 glGetBooleanv function, 73, 324–325
 glGetBufferParameteriv function, 337
 glGetBufferPointervOES function, 337
 glGetError command, 15, 16
 glGetFloatv function, 324–325
 glGetFramebufferAttachmentParameteriv function, 338
 glGetIntegeriv, 197
 glGetIntegerv function, 202, 324–325, 331
 glGetProgramInfoLog function, 31, 65
 glGetProgramiv function, 31, 64, 67, 68, 112
 glGetRenderbufferParameteriv function, 338
 glGetShaderInfoLog function, 61
 glGetShaderiv function, 60
 glGetShaderSource function, 334
 getString function, 323–324, 353

`glGetTexParameterfv` function, 336
`glGetUniformfv` function, 334
`glGetUniformiv` function, 334
`glGetUniformLocation` function, 69
`glGetVertexAttribfv` function, 335–336
`glGetVertexAttribiv` function, 335–336
`glGetVertexAttribPointerv` function, 335
`glHint` function, 331
`GLint` type, 15
`glIS*` functions, 331
`glIsEnable` function, 332–333
`glIsEnabled` command, 18
`glLineWidth` API call, 129–130
`glLinkProgram` function, 31, 63–64
`glMapBufferOES` command, 124–125
global data, avoiding in C++, 342
global static data, handheld platforms not allowing, 26
`glPixelStorei` function, 187, 203
`glPolygonOffset`, 145
`glReadPixels` command, 250–251, 270–271
`glReleaseShaderCompiler` function, 73–74
`glRenderbufferStorage` command, 260–261, 274
`glSampleCoverage` function, 250
`glScissor` function, 239
`glShaderBinary`, 74–75
`glShaderSource` function, 29, 59
`GLshort` type, 15
`glStencilFunc` function, 240
`glStencilFuncSeparate` function, 240
`glStencilMask`, 237–238
`glStencilMaskSeparate`, 238
`glStencilOp` function, 241–245
`glStencilOpSeparate` function, 241–245
`glTexImage2D` function
 calling, 185–186, 187, 188
 calling each cubemap face, 199
 compared to `glRenderbufferStorage`, 260
 loading a mipmap chain, 189
`glTexImage3DOES` command, 207–209
`glTexParameter`, 190–191, 194
`glTexParameteri`, 188
`glTexSubImage2D` function, 202–203, 211
`glTexSubImage3D` function, 211
`glTexSubImage3DOES` function, 210–211
`GLubyte` type, 15
`GLuint` type, 15
`glUniform*` calls, 69–71
`glUnmapBufferOES` command, 125
`glUseProgram` function, 31–32, 66
`GLushort` type, 15
`glValidateProgram` function, 65–66
`glVertexAttrib*` commands, 102–103
`glVertexAttribPointer` function, 33, 103–107, 108
`glViewport` command, 32, 140, 141
`GLvoid` type, 15
gouraud shaded triangle, 8–9
GPUs
 desktop, 25
 flow control and looping difficult for, 88
 waiting for rendering commands, 125–126
gradient noise, 2D slice of, 312–313
gradient vectors
 calculating, 321
 generated, 309–311
grammar, of the shading language, 375–383
graphics hardware operation, 357
graphics pipeline, 3–11
ground-based fog, 227
guard-band region, 139

H

half-float numbers
 converting to float-point, 355–356
 representations of, 354–355
handheld and embedded devices, 1, 2
handheld devices, diversity of platforms, 339

handheld platforms, 339–341
hardware functionality, built-in
 functions exposing, 357
header files, 13
Hello Triangle
 building and running, 25
 fragment shader in, 28
 full source code for, 21–24
 LoadShader function in, 29
 main function in, 26–27
 using OpenKODE, 343–350
 vertex shader in, 27–28
Hello_Triangle_KD, 343
hidden-surface removal, 245
high precision, in the fragment shader,
 221
highp keyword, 96, 152
hints, 330–331, 372

I

i argument data type, 15
identity matrix, 394
if conditional statements, 156
#if directive, 92
if-else conditional statements, 156
if-then-else logical tests, 87
image(s), updating subregions of, 202
image data
 copying out of the color buffer, 204
 loading, 185, 188
 two-dimensional array of, 182
image postprocessing, 296–300
image processing, 296
image quality, 2
Imagination Technologies, 341
implementation string queries,
 323–324
implementation-dependent limits,
 querying, 324–327
implementation-dependent
 parameters, 325–327
implementation-specific constants, 157
implementation-specific values, 251–252
in qualifier, 85
include files, 13
individual primitives, rendering, 136,
 137

info log
 format of, 61
 for program objects, 65
 retrieving, 61
 written by the compiler, 60
InIt function, 27
inout qualifier, 85
input attribute, in a shader, 28
inputs, to the vertex shader, 4, 5, 148
insufficient memory, 16
int, precision for, 152
integer-based variables, precision for, 96
integer-based vector types, 79
internal memory, 277
INTERPOLATE RGB combine function,
 217
interpolation, 5
interpolators, 94
invariance, 97–100
invariant keyword, 97, 98
inversion flag, 250

K

kdMain function, 350
keyboard input processing callback
 function, 387
Khronos Group, 1
 built-in functions copyrighted by,
 357
 shading language grammar
 copyrighted by, 375
Khronos rendering APIs, 12
Khronos Web site, 18

L

language extension specification, 93
latitude longitude map, 167
lattice-based gradient noise, 309
left clip plane, 138
lexical analysis, 375
libEGL.lib, 13
libGLSLv2.lib, 13
libraries, linking with, 13
lifetime, of a particle, 293
light
 intensity of emitted, 164
 projective view space of, 302

- light bloom effect
 - implementing, 298–300
 - stages of, 299, 300
- light colors, multiplying, 163
- light projection matrix, 303
- light rays, parallel, 161
- light source, 306
- light view matrix, 303
- lighting
 - with a normal map, 280–281
 - per-fragment, 279–285
 - in a vertex shader, 160–166
- lighting equation model, 160
- lighting equations
 - for a directional light, 161
 - in the fragment shader, 285
- lighting shaders, 281–285
- lighting vectors, 281
- limitations
 - on fragment shaders, 221
 - on recursive functions, 86
 - on vertex shaders, 152–159
- limits, implementation-dependent, 324–327
- line primitives, 129
- line rasterization, 7, 141
- line segments, drawing, 129
- linear approximations, 372
- linear fog, 224, 226–227
- linear fog factor, 314
- linear interpolation, 285
- lines, 129–130, 139
- link status, checking, 64
- linking, program objects, 63–64
- literal values, 157, 158–159
- LoadShader function, in Hello Triangle, 29
- location value, for a uniform, 69
- LogicOp, removed as a feature, 10–11
- loop index, in a `for` loop, 154
- loops, 87–88
- `lowp` keyword, 96, 152

M

- macros, 92
- magnification, 190
- magnification filtering mode, 188
- main function
 - of the fragment shader, 9
 - in Hello Triangle, 26–27
 - in a shader, 6, 28
- main loop, starting, 386
- main message processing loop, 27
- make current process, 52
- MakeBinaryShader.exe, 351
- mandatory extensions, 12
- map command, 124
- mapping, buffer objects, 124–126
- masking parameter, for the stencil test, 240
- masks, 236–238
- `material_properties` struct, 161–162
- matrices, 78
 - accessing, 81–82
 - constructing, 80–81
 - floating-point based, 79
 - multiplying, 394
 - number used to skin a vertex, 168
 - for projective texturing, 303–304
 - in transformations, 390
- matrix functions, built-in, 366–367
- matrix palette, 169
- maximum renderbuffer size, 273
- `mediump` keyword, 96, 130, 152
- memory footprint, minimizing, 12
- memory manager, writing on Symbian, 342
- mesh
 - authoring, 182
 - number of indices in, 134
- Microsoft
 - Visual Studio 2005, 25
 - Visual Studio 2008 Express Edition, 20
 - Windows Mobile operating system, 340
 - Windows Vista, 20
 - Windows XP, 20
- minification, 190
- minification filter, 191, 192
- minification filtering mode, 188
- mip levels, 189
- mipmap chain, 189–190, 193

mipmap filtering mode, 193
mipmap generation, automatic,
193–194
MipMap2D example, 189, 192, 193
mipmapping, 189, 190
MODULATE RGB combine function, 217
multiple inheritance, avoiding in C++,
342
multiple-render targets (MRTs), 220–221
multiplication, performing
 component-wise, 367
multisample buffer, allocating, 54
multisample rasterization, 372
multisampled antialiasing, 249–250
multisampling, 234, 249, 257
MultiTexture example, 222–223
multitexturing, implementing,
222–224

N
name queries, of entities, 331
native window, creating, 44
near clip plane, 138
nearest sampling, 189
noise
 computing in the fragment shader,
 308
 generating, 308–313
 using, 313–315
 using a 3D texture, 307–315
noise distorted fog, 313–315
Noise3D sample, 307
Nokia devices, 339, 340
non-floating-point data types, 108
non-power-of-2 textures, 214
nonprogrammable operations,
332–333
normal, as a texture coordinate, 200
normal map, 222, 280–281
normal vector, 280, 289
normalization, of vectors, 285
normalized device coordinates, 139,
140
normalized flag, 108
npot textures, 214
nput dimensions, 214
numeric argument, out of range, 16

O
object name, for an attachment point,
263
object or local coordinate space, 137
object types
 for an attachment point, 262
 creating to render with shaders, 57
odd-numbered triangle, 134
OES_depth_texture extension, 274
OES_map_buffer extension, 124
offline tool, compiling shader source
 code, 73
off-screen rendering area, 46–50
off-screen surfaces, 12
off-screen-buffer, rendering, 262
online linking, 75
online resources, for handheld
 platforms, 340–341
online shader compilation, 73
on-screen rendering area, 43–46
on-screen surfaces, 12
on-screen window, 43
opaque objects, rendering, 237
OpenGL, 1, 16–17
OpenGL 2.0 example, 25
OpenGL ES, 1
 buffer usage enums not supported,
 118
 device constraints addressed by, 1
 specifications, 2
OpenGL ES 1.0, 2
OpenGL ES 1.1, 2, 102, 173
OpenGL ES 1.x, 2, 11–12
OpenGL ES 2.0, 2
 API specification, 3
 downloading, 357, 375
 emulator, 25
 example drawing a triangle, 19–34
 framework, 26–27
 graphics pipeline, 3–11
 header file, 13
 introduction to, 3–11
 library, 13
 not backward compatible, 11–12
 obtaining values from, 323–338
 programming with, 13–18
 shading language, 78–100, 375–383

- OpenGL ES shading language
 - basics of, 78–100
 - built-in functions, 357–373
 - data types, 78–79
 - functions for computing noise, 308
 - grammar for, 376–383
 - OpenGL ES Shading Language
 - Specification (OpenGL ES SL), 3
 - OpenGL ES state, 12, 327–330
 - OpenKODE 1.0 specification, 343–350
 - operators, 84–85
 - opposite vertex order, 135
 - optional extensions, 207–214
 - orientation, of a triangle, 142
 - orthographic projection matrix, 392
 - orthonormal matrices, 169
 - OS APIs, for handheld devices, 343
 - OS-specific calls, removing, 350
 - out qualifier, 85
 - out-of-memory event, 125
- P**
- packing rules, 94, 157
 - parameters, implementation-
 - dependent, 325–327
 - particle effects, 130
 - particle explosion, 290–291
 - particle system
 - additive blend effect for, 294
 - fragment shader, 293–294
 - with point sprites, 290–296
 - sample, 290
 - setup, 290–291
 - vertex shader, 291–293
 - particles, drawing, 294
 - ParticleSystem sample program, 290
 - pbuffers, 253
 - attributes list, 47
 - compared to Windows, 50
 - creating, 46, 48
 - implementing render to texture, 254
 - rendering into, 46
 - performance
 - hints biasing toward, 330–331
 - implications for texture filtering mode, 193
 - performance tips, for framebuffer objects, 277–278
 - per-fragment lighting, 279–285
 - example, 280
 - fragment shader, 283–285
 - variations possible on, 285
 - vertex shader, 281–283
 - per-fragment operations, 9–11
 - PerFragmentLighting.rfx, 280
 - Perlin, Ken, 309
 - perspective division, 139–140
 - perspective projection matrix, 390–391
 - per-vertex data, 101
 - physical address space, 94
 - physical screen, updating, 33
 - physical storage, 94
 - piecewise linear approximations, 372
 - ping-ponging, 299
 - pixel buffers. *See* pbuffers
 - pixel colors, blending, 246
 - pixel ownership test, 10, 257
 - pixels
 - reading, 252, 270–271
 - reading and writing, 250–252
 - two-dimensional fragments
 - representing, 7
 - plane equation, for a clip plane, 230
 - planes, equation specifying, 229
 - point coordinate origin, for point sprites, 130
 - point light, 163
 - point sampling. *See* nearest sampling
 - point size, 130
 - point sprite rasterization, 7, 141
 - point sprites, 130–131
 - clipping, 139
 - no connectivity for, 295
 - particle system with, 290–296
 - rendering, 220
 - writing the size of, 150
 - point sprites primitive, 130
 - pointer address, retrieving, 337
 - pointSizeRange command, 130
 - polygon offset, 144, 145
 - polygons, smoothing joins between, 168
 - portability, of the code framework, 20

- position, of a point sprite, 130
- PostProcess sample, 296, 298
- postshader fragment pipeline, 233
- post-transform vertex cache, 136
- `pow` built-in function, 86
- `#pragma` directive, 92, 99–100
- precision, of shader language types, 334–335
- precision qualifiers
 - for built-in functions, 358
 - for fragment shaders, 221
 - introduction to shading language, 96–97
 - review of, 152
 - setting default, 9
- precompiled binary format, 350
- precomputed lighting, 222
- preprocessing path, 351
- preprocessor, 92
- primitive assembly, 6–7
- primitive assembly stage, 136–137
- primitive types, operations in, 139
- primitives, 6, 127–136
 - converting two-dimensional fragments, 7
 - drawing, 33, 119–124, 131–136, 227
 - types of, 127
- procedural textures
 - antialiasing of, 319–322
 - benefits of, 315
 - disadvantages of, 315–316
 - example, 316–319
 - generation of, 315–322
- profiles, none in OpenGL ES 2.0, 12
- program info log, 65
- program objects, 57, 58
 - attachment to, 29
 - creating, 62, 388
 - creating, attaching shaders to, and linking, 66–67
 - creating and linking, 30–31
 - deleting, 62–63
 - linking, 63–64
 - querying for uniform information, 71
 - sets of uniforms for, 67
 - setting up vertex attributes, 72
 - validating, 65–66
- programmable graphics pipeline, 3
- programmable pipeline, 11, 147, 216
- programmable vertex pipeline, 147
- programming, 13–18
- programs, state queries, 333–335
- projection matrix, of the light source, 303
- projective light space position, 302
- projective spotlight, 301
- projective spotlight shaders, 304–307
- projective texture coordinate, 306, 307
- projective texturing, 300–307
 - basics, 301–302
 - described, 301
 - fragment shader, 306–307
 - matrices for, 303–304
 - vertex shader for, 304–306
- ProjectiveSpotlight, 300
- PVRUniSCo, 351
- PVRUniSCo Editor, 351

Q

- quads, drawing, 194–195
- Qualcomm
 - BREW operating system, 339
 - BREW SDK, 341
- qualifiers, 85–86
- quality, hints biasing toward, 330–331

R

- radius, of a point sprite, 130
- range, of shader language types, 334–335
- rasterization, 127, 141–146
 - controlled by turning on and off features, 332
 - multisample, 372
 - single-sample, 372
- rasterization phase, 7
- rasterization pipeline, 141
- r*-coordinate, 209
- read-only variables, 111
- recursive functions, limitation on, 86
- redundancy, removing, 2
- reflection vector
 - computing, 285, 366
 - in world space, 289

- render targets, blurring, 299
- render to depth texture example, 274–277
- render to texture setup, 297
- renderbuffer(s)
 - attaching to a framebuffer
 - attachment, 263–264
 - state queries, 337–338
 - width and height of off-screen, 277
- renderbuffer objects, 255
 - allocating, 258
 - binding to existing, 260
 - creating, 258
 - deleting, 269–270
 - setting, 259, 260
 - state and default values associated with, 259
 - versus texture objects, 256–257
 - using, 259–261
- rendering
 - advanced techniques, 279
 - completing for a frame, 34
 - reducing visual artifacts of, 249
 - synchronizing, 54–55
- rendering context, 253
 - creating, 13, 50–52
 - required in OpenGL ES commands, 12
- rendering surfaces
 - creating, 12
 - types and configuration of, 38
- Renderman shading language, 309
- RenderMonkey, 20, 26
 - computing a tangent space
 - automatically, 281
 - demonstrating fog computation, 224–225
- repeatability, 316
- REPLACE RGB combine function, 217
- require extension behavior, 93
- .rfx extension, 26
- RGB-based color buffer, 54
- right clip plane, 138
- RM_ClipPlane workspace, 229–230
- rotation matrix, 393
- runtime type information, avoiding in C++, 342

- S**
- s argument data type, 15
- same vertex border, 135–136
- sample coverage mask, 249
- Sample_TextureCubemap example, 198
- sampler uniform variable, 196
- samplers
 - code for setting up, 223–224
 - input to the fragment shader, 8
 - in a vertex shader, 4, 5
- samples, resolving, 249
- scalar-based data types, 78
- scaled color, 248
- scaling factors, 246
- scaling matrix, 392
- scissor box, 239, 333
- scissor test, 10, 239
- scissoring operation, 139
- s-coordinate, 194
- screen, two-dimensional images on, 33
- screen resolution, 125
- Series 60 on Symbian operating system, 339
- shader(s)
 - attached to a program, 333
 - attaching to a program object, 63
 - code paths for loading, 75
 - compiling, 59–60
 - compiling and loading, 29–30
 - creating and compiling, 58–62
 - detaching, 63
 - implementing fixed function techniques, 222–231
 - loading, 61–62
 - loading and checking, 388
 - state queries, 333–335
- shader binaries, 75, 350–351
- shader code, rendering a cubemap, 199–200
- shader compiler, 73–74
- shader development tool, 26
- shader effects, 26
- shader objects, 57
 - creating, 58
 - deleting, 58–59
 - linking in program objects, 75
 - returning, 29

shader program, 4, 8, 148
 shader source code
 providing, 59
 returning, 334
 returning the length of, 60
 shading language. *See* OpenGL ES shading language
 shadow map aliasing, 220
 shareContext parameter, 50
 sheeter binaries, 350–351
 shiny objects, reflected image on, 167
 SIMD array elements, 373
 Simple Bloom effect, 296
 Simple_Texture2D example, 197
 single-sample rasterization, 372
 slice, 212
 Sony Ericsson, 340
 source shader object, 75
 special variables, 149, 219–220
 specular color, computing, 285
 specular exponent, 162
 specular lighting, 285
 sphere, geometry for, 388
 sphere map, 167
 spotlight
 example, 164–166
 lighting equation for, 163–164
 rendering a quad using, 300
 Standard Template Library (STL), 341–342
 start position, of a particle, 293
 state
 of a buffer object, 118
 management, 17–18
 values, 18
 state enables, 17, 18
 state queries
 application-modifiable, 327–330
 implementation-dependent, 325–327
 renderbuffers and framebuffers, 337–338
 shaders and programs, 333–335
 texture objects, 336
 vertex attribute arrays, 335–336
 vertex buffer objects, 337
 static lighting, 222
 stencil and depth tests, 10
 stencil attachment, as valid, 267–268
 stencil attachment point, 262
 stencil buffer
 allocating, 54
 disabling writing to, 237–238
 initializing, 240
 obtaining, 235
 testing, 240–245
 stencil mask, setting, 238
 stencil renderbuffers, 278
 stencil test, 240–245, 333
 stencil-renderable buffer, 261
 stencil-renderable formats, 268
 stored texture image, 315
 structure of arrays, 104, 106–107, 122
 structures, 82–83
 SUBTRACT RGB combine function, 217
 surface configurations, 38
 swizzles, 81
 Symbian
 C++ support, 341–342
 on Sony Ericsson devices using UIQ, 340
 Symbian operating system, 339, 340
 synchronizing, rendering, 54–55

T

tangent matrix, 283
 tangent space, 281, 283
 t-coordinate, 194
 temporary variables, 153
 texels, 182, 183–184
 texture(s). *See also* texturing
 applying to surfaces, 181
 avoiding modifying, 277
 compressed, 201–202
 floating-point, 213
 forms of, 181
 as inputs to the fragment shader, 218
 making the currently bound texture object, 274
 non-power-of-2, 214
 rendering to, 254, 262, 271–273
 using the fragment shader, 196–198
 texture color, in a fixed function pipeline, 216–217

- texture compression formats, 202
- texture coordinates, 182
 - for 2D textures, 182, 183
 - generating in a vertex shader, 167–168
 - specifying, 105
 - transforming, 157–158, 173
 - wrapping, 194–195
- texture copy, format conversions for, 206–207
- texture data, 182, 204–207
- texture filter settings, 336
- texture filtering, mipmapping and, 188–193
- texture filtering mode, 193
- texture formats, mapping to color, 197–198
- texture image units, 220
- texture level, specifying, 263
- texture lookup functions, built-in, 369–371
- texture maps, 46, 222
- texture mip-level, 274
- texture object code, 186
- texture objects
 - binding, 185
 - creating, 184
 - deleting, 185
 - generating, binding, and loading, 186–188
 - modifying state and image of, 265
 - state queries about, 336
 - as texture input and a framebuffer attachment, 267
- texture pixels, 182
- texture subimage specification, 202–204
- texture target, binding texture objects to, 185
- texture units
 - binding sampler and texture to, 197
 - binding textures to, 196–197
 - code for setting up, 223–224
 - maximum number of, 151
 - setting the current, 197
- texture wrap modes, 194, 195
- texture wrapping mode, 182
- texture Z offset, 263
- texture2D function, 197
- texture2DProj built-in function, 301–302
- texture3D built-in function, 209
- textureCube built-in function, 200–201
- textured cubemap face, specifying, 263
- texturing
 - basics of, 181–184
 - procedural, 315–322
 - projective, 300–307
- TGA image, loading, 388
- title-based rendering architecture, 277
- tool suites, generating binary shaders, 351
- top clip plane, 138
- transformation functions, 390–394
- translation matrix, 393
- translucent objects, 237
- transpose argument, 70
- triangle primitives, 128
- triangle rasterization, 7, 141, 143–144
- triangle strips, connecting, 134, 135
- triangles, 128
 - clipping, 139
 - culling, 142–143
 - degenerate, 134, 135
 - drawing, 128, 132
 - facing of, 143
 - geometry for, 33
- trigonometry functions, 358–359
- trivial operation, as a built-in function, 357
- type conversions, 79
- type rules, between operators, 84

U

- ub argument data type, 15
- ui argument data type, 15
- UIQ Symbian operating systems, 340
- uniform declarations, packing, 94–95
- uniform matrices, 283
- uniform storage, 157
- uniform values, 71
- uniform variables. *See* uniforms
- uniformed variable, 334

- uniforms, 67
 - functions for loading, 69–71
 - getting and setting, 67–72
 - input to the fragment shader, 8, 218
 - as inputs to the fragment shader, 219
 - number used in a vertex shader, 157–159
 - in the OpenGL ES Shading Language, 88–89
 - querying for active, 71–72
 - storing, 94
 - in a vertex shader, 4, 5
- unmap command, 124
- unpack alignment, 187–188
- unrolled shader, 153
- update callback function, 387
- update function, for particle system sample, 292–293
- us argument data type, 15
- user clip planes
 - computing, 173
 - implementing using shaders, 229–231
- userData member variable, 26
- utility functions
 - building up a library of, 20
 - performing transformations, 390–394

V

- v_eyeDist varying variable, 226
- valid attachment, 268
- validation, of program objects, 65–66
- variable constructors, 79–81
- variables, precision of, 96
- varying declarations, 91
- varying variables, 5, 90, 148
 - input to the fragment shader, 8
 - interpolating across a primitive, 285
- varying vectors, 151
- varyings
 - as inputs to the fragment shader, 218, 219
 - in the OpenGL ES Shading Language, 90–92
 - storing, 94
- vboIds, 117
- vec4 attributes, 111
- vec4 uniform entries, 151, 220
- vector constructors, 80
- vector input, 357
- vector relational functions, built-in, 367–369
- vectors, 78, 81
- #version directive, 92
- versions, of OpenGL ES, 323, 324
- vertex, storing attributes of, 107
- vertex array attribute, 109–110
- vertex arrays, 101, 103–107
- vertex attribute arrays, 109, 335–336
- vertex attribute data, 102–110
- vertex attribute index, 113
- vertex attribute variables, 110–115
- vertex attributes
 - binding to attribute variables, 113–115
 - data format, 107
 - finding the number of active, 111–112
 - maximum number of, 150
 - minimum supported, 102
 - in primitive assembly stage, 6
 - setting up, 72
 - specifying and binding for drawing primitives, 114
 - specifying with `glVertexAttribPointer`, 104
 - storing, 104, 107, 111
 - user-defined, 102
- vertex buffer objects, 116
 - creating and binding, 116–117
 - drawing with and without, 120–122
 - mapping data storage, 124–125
 - state queries, 337
 - using, 295
- vertex cache, 136
- vertex color, in a fixed function pipeline, 216
- vertex coordinates, 138
- vertex data, 115–116. *See also* vertex attributes
- vertex lighting equation, 173
- vertex normal, 105

- vertex pipeline, 173–179
- vertex position attribute, 104–105
- vertex position, converting, 149
- vertex shader(s)
 - attaching to a program object, 30
 - attributes as per-vertex inputs to, 90
 - binding vertex attributes, 113–115
 - built-in variables, 149–152
 - computing distance to eye, 225–226
 - counting number of uniforms used, 157–159
 - creating, 27–28, 58
 - declaring vertex attributes variables, 110–115
 - default precision, 97
 - described, 147
 - for the environment mapping
 - example, 286–288
 - example using OpenGL ES shading language, 5–6
 - examples, 159–166
 - in Hello Triangle, 27–28
 - implementing checkerboard texture, 316–317
 - inputs and outputs of, 5, 148
 - lighting in a, 160–166
 - limitations on, 152–159
 - maximum number of instructions supported, 153
 - in OpenGL ES 2.0, 4–6
 - overview of, 148–159
 - for a particle system, 291–293
 - performing vertex skinning, 156
 - for per-fragment lighting, 281–283
 - for projective texturing, 304–306
 - specifying and binding to attribute names, 113, 114
 - in a two-component texture coordinate, 196
 - user clip plane, 230–231
 - vertices transformed by, 136
 - writing a simple, 160
- vertex shader code, 169–171
- vertex shader object, 58
- vertex shader stage, 147
- vertex skinning, 168–172

- vertices
 - operating on, 4
 - for a triangle, 33
- view frustum, 7
- view matrix, 303, 305–306
- viewing volume. *See* clip volume
- viewpoint transformation, 140–141
- viewport
 - rendering in, 242–245
 - setting, 32
 - state, 141
- visible buffer. *See* front buffer
- visual artifacts, 189
- Visual Studio 2005, 341
- `vPosition` attribute, 28, 31, 33
- `vVertices` array, 33

W

- warn extension behavior, 93
- Web site, for this book, 20
- while loops, 155
- Win32 API, 340
- winding order, preserving, 136
- window(s)
 - code for creating, 45–46
 - compared to puffers, 50
 - creating specific width and height, 27
 - creating with specified parameters, 386
- window coordinates, 140, 150
- window origin (0, 0), 130
- window relative coordinates, of a fragment, 219–220
- window system, drawable surface, 257
- windowing system
 - communicating with, 36–37
 - guaranteeing rendering completion, 55
- Windows Mobile 6 SDK, downloading, 341
- Windows Mobile Developer Center, 341
- Windows Mobile operating system, 340
- Windows/Linux developers, development environment for, 340

wispy fog effect, 313, 315
world space, 287–288
world-space normal vector, 281
world-space reflection vector, 288
wrap modes, 194, 307
writable static global variables, 340
write masks, 10

X

x argument data type, 15

Z

Z fighting, 98, 143