OpenGL Programming Guide for Mac OS X

Graphics & Imaging > OpenGL

2008-06-09

Contents

Figures, Tables, and Listings

Introduction to OpenGL Programming Guide for Mac OS X

OpenGL Programming Guide for Mac OS X describes the Apple implementation of the OpenGL graphics standard in Mac OS X and shows how to use this implementation effectively to achieve stunning 3D graphics. OpenGL is an open, cross-platform, three-dimensional (3D) graphics standard with broad industry support. OpenGL greatly eases the task of writing real-time 2D or 3D graphics applications by providing a mature, well-documented graphics processing pipeline that supports the abstraction of current and future hardware accelerators.

OpenGL was developed by Silicon Graphics, Inc. (SGI). It is based on the SGI IRIS Graphics Library, first released in 1992. As an open standard, it is now controlled by the OpenGL Architecture Review Board (ARB), a consortium whose members include many of the major companies in the computer graphics industry, one of which is Apple.

OpenGL is an excellent choice for 3D graphics development on the Macintosh platform because it offers the following:

- Reliable implementation. Each implementation of OpenGL, including the Apple one, adheres to the OpenGL specification and must pass a set of conformance tests.
- Industry acceptance. Besides OpenGL for Mac OS X, there are OpenGL implementations for Windows, Linux, Irix, Solaris, and many game consoles.
- Performance. OpenGL uses available graphics processing hardware features to improve rendering speeds.
- Controlled evolution. OpenGL extensions enable developers to take advantage of hardware-specific improvements as they become available. Successful innovations are automatically incorporated into the Apple implementation.
- Full feature set. OpenGL provides hundreds of graphics routines that you use to define objects and apply transformations to them. It also provides routines that let you package data so that it uses the least amount of resources, thereby optimizing performance.
- Platform independence. The Apple implementation of OpenGL is cross-platform, which means that you can leverage your Mac OS X development efforts onto other systems. The OpenGL core functionality abstracts hardware details and guarantees consistent presentation on any compliant hardware and software configuration.

Who Should Read This Document?

Any developer who is familiar with OpenGL code and wants to run OpenGL programs in Mac OS X will want to read this document. OpenGL provides the API that communicates with the graphics hardware. Apple provides APIs that communicate with the Mac OS X windowing system. By reading this guide, you'll see how to use the Apple APIs to draw your OpenGL content onscreen from within a Cocoa or Carbon application. The book discusses the essential concepts for understanding the Apple OpenGL interfaces used for procedural C and Objective-C, and provides techniques and tips for getting the best performance possible on the platform.

This guide assumes that you are an experienced OpenGL programmer who wants to create Mac OS X software that has real-time 2D or 3D graphics. Although this guide provides some advice on optimizing OpenGL code, it does not provide entry-level information on how to use the OpenGL API maintained by the OpenGL Architecture Review Board (ARB). If you are unfamiliar with OpenGL, first read the following programming guide and consult the companion reference as needed:

- OpenGL Programming Guide, by the OpenGL Architecture Review Board; otherwise known as "The Redbook."
- OpenGL Reference Pages.

Organization of This Document

This programming guide contains the following chapters:

- "OpenGL on the Mac Platform" (page 15) discusses fundamental concepts for understanding how to use the Apple implementation of OpenGL, describes the graphics layers and programming interfaces, introduces essential terminology, and provides an overview of an OpenGL program running in Mac OS X.
- "Drawing to a Window or View" (page 27) shows the basics of onscreen drawing using the CGL, AGL, and Cocoa APIs.
- "Drawing to the Full Screen" (page 37) describes how to use the CGL, AGL, and Cocoa APIs for full-screen drawing, and includes information on adjusting the display mode.
- "Drawing Offscreen" (page 45) shows how to draw to GPU memory, offscreen windows, pixel buffers, and framebuffer objects.
- "Determining the OpenGL Capabilities Supported by the Hardware" (page 59) provides information on how to detect which version of OpenGL is available on a system and which features are supported by the hardware.
- "Techniques for Working with Rendering Contexts" (page 65) shows how to create and update rendering contexts, set a context to a specific display, and share contexts.
- "Techniques for Choosing Attributes" (page 79) discusses which render and buffer attributes to use and which to avoid, and which to choose to achieve specific objectives.
- "Techniques for Working with Vertex Data" (page 85) provides guidelines, describes the data path, and shows how to optimize vertex data throughput.
- "Techniques for Working with Texture Data" (page 95) provides guidelines, describes the data path, shows how to use images as textures, and discusses how to optimize texture data throughput.
- "Techniques for Scene Anti-Aliasing" (page 111) describes the primary methods provided by anti-aliasing hardware and shows how to use hints that indicate which method you prefer.
- "Multithreading and OpenGL" (page 117) provides guidelines for multithreading and discusses effective program designs.
- "Improving Performance" (page 121) discusses best practices and shows how to analyze performance.

This programming guide contains these appendixes:

- "OpenGL Functionality by Version" (page 135) contains tables that summarize new functionality and, in the online versions of this document, provides links to the OpenGL specification that describes the functionality in detail.
- "Setting Up Function Pointers to OpenGL Routines" (page 139) describes how to obtain function pointers to arbitrary OpenGL entry points.
- "Quartz Display Services and Full-Screen Mode" (page 145) shows how to use the Quartz Display Services API to accomplish a number of tasks that are useful in full-screen mode.

The "Glossary" (page 151) provides definitions for most of the terminology in the book. You can find definitions for those terms that are not in the glossary either on the OpenGL Foundation website http://www.opengl.org or in OpenGL Programming Guide ("The Redbook").

See Also

You'll want to keep these reference documents handy as you develop your OpenGL program for Mac OS X:

- *NSOpenGLView Class Reference*, *NSOpenGLContext Class Reference*, *NSOpenGLPixelBuffer Class Reference*, and *NSOpenGLPixelFormat Class Reference* provide a complete description of the classes and methods needed to draw OpenGL content in a Cocoa application.
- *AGL Reference* provides a complete description of the functions needed to draw OpenGL content in a Carbon application.
- *CGL Reference* describes functions in the Core OpenGL API, which can be used to draw OpenGL content to the full screen from either a Cocoa or Carbon application.
- *OpenGL Extensions Guide* provides information about OpenGL extensions supported in Mac OS X.
- *Cocoa Drawing Guide* explains how to draw 2D content in a Cocoa application and also contains information on how to set up OpenGL drawing.

You can download sample applications that demonstrate how to use Apple APIs for OpenGL drawing from the ADC Reference Library.

The OpenGL Foundation website, http://www.opengl.org, provides information on OpenGL commands, the Architecture Review Board, logo requirements, OpenGL news, and many other topics. It's a site that you'll want to visit regularly. Among the many resources it provides, the following are important reference documents for OpenGL developers:

- *OpenGL 2.0 Specification* provides detailed information for every OpenGL command.
- *OpenGL Reference* describes GL, the main OpenGL library.
- *OpenGL GLU Reference* describes the OpenGL Utility Library, which contains graphical extensions based entirely on GL functions.
- *OpenGL GLUT Reference* describes the OpenGL Utility Toolkit, a standard API for performing operations associated with a windowing environment other than the Cocoa and Carbon environments.
- OpenGL API Code and Tutorial Listings provides code examples for fundamental tasks, such as modeling and texture mapping, as well as for advanced techniques, such as high dynamic range rendering (HDRR).

Although you don't need to learn how to use a shading language to write OpenGL programs for Mac OS X, it's a growing area that you may want to investigate. The Apple implementation of OpenGL supports shading programs should you want to incorporate them into your code.

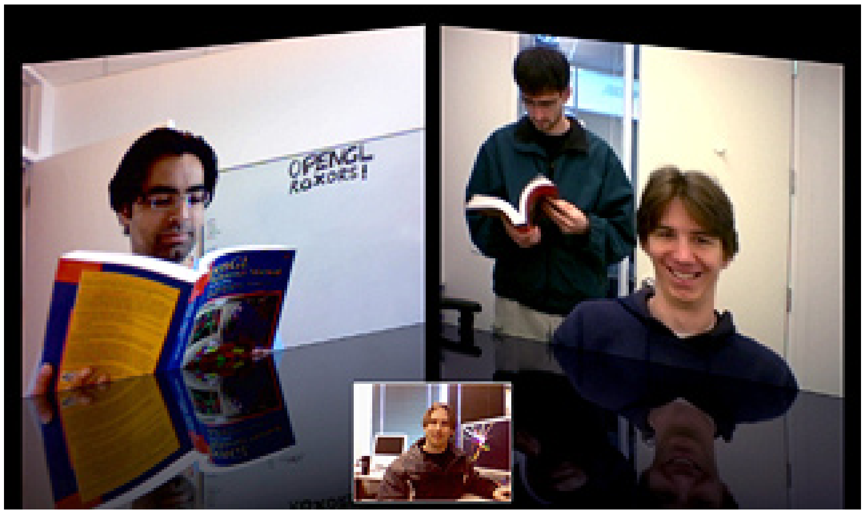
- *OpenGL Shading Language*, by Randi J. Rost, is an excellent guide for those who want to write programs that compute surface properties (also known as **shaders**).
- *Core Image Programming Guide* describes how to use the Core Image API to access built-in image processing filters and how to write your own filters. The appendix Core Image Kernel Language, describes the shading language that's supported in Core Image and provides examples of kernel routines.
- The Quartz Composer application, available in /Developer/Applications/Graphics Tools/, has a kernel patch that you can use to test kernel routines that you write using the Core Image kernel language.

OpenGL on the Mac Platform

You can tell that Apple has an implementation of OpenGL on its platform just by looking at the user interface for many of the applications that are installed with Mac OS X. The reflections built into iChat (Figure 1-1) provide one of the more notable examples. The responsiveness of the windows, the instant results of applying an effect in iPhoto, and many, many other operations in Mac OS X v10.4 are due to the use of OpenGL. OpenGL isn't restricted to just the operating system and Apple applications. Any Mac developer can use OpenGL. In fact, Apple's implementation is available to every Macintosh user as part of Mac OS X.

OpenGL for Mac OS X is implemented as a set of frameworks that contain the OpenGL runtime engine and its drawing software. These frameworks use platform-neutral virtual resources to free your programming as much as possible from hardware considerations. Mac OS X provides a set of application programming interfaces (APIs) that Cocoa and Carbon applications can use to support OpenGL drawing.

Figure 1-1 OpenGL provides the reflections in iChat

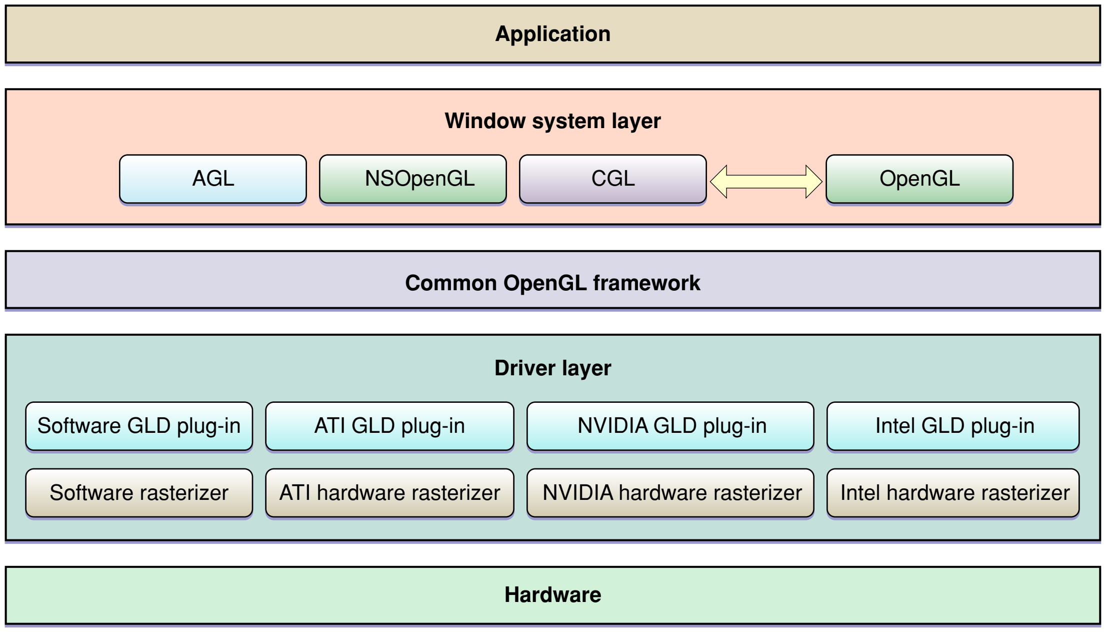


This chapter describes the OpenGL frameworks and the associated APIs, defines the terminology that is Apple-specific, describes how data flows through OpenGL, and provides an overview of the tasks necessary for a Cocoa or Carbon application to tap into that pipeline.

Structure of OpenGL in Mac OS X

Mac OS X supports a display space that can consist of multiple dissimilar displays, each driven by different graphics cards with different capabilities. In addition, multiple OpenGL renderers can drive each graphics card. To accommodate this versatility, OpenGL for Mac OS X is segmented into three well-defined layers: a window system layer, a framework layer, and a driver layer, as shown in Figure 1-2. This segmentation allows for plug-in interfaces to both the window system layer and the framework layer. Plug-in interfaces offer flexibility in software and hardware configuration without violating the OpenGL standard.

Figure 1-2 Layers of OpenGL for Mac OS X



The **window system layer** is what allows your OpenGL program to become a reality onscreen. You'll use the Apple-specific OpenGL APIs provided in this layer—the NSOpenGL classes and the AGL and CGL APIs—to direct where OpenGL drawing takes place and control a variety of aspects of rendering. These APIs contain functions and methods specific to the Mac OS X windowing system. (See "OpenGL APIs Specific to Mac OS X" (page 17) for more information.) This layer also includes the OpenGL libraries—GL, GLU, and GLUT. (See "Apple-Implemented OpenGL Libraries" (page 18) for details.)

The **common OpenGL framework layer** is the software interface to the graphics hardware. This layer contains Apple's implementation of the OpenGL specification.

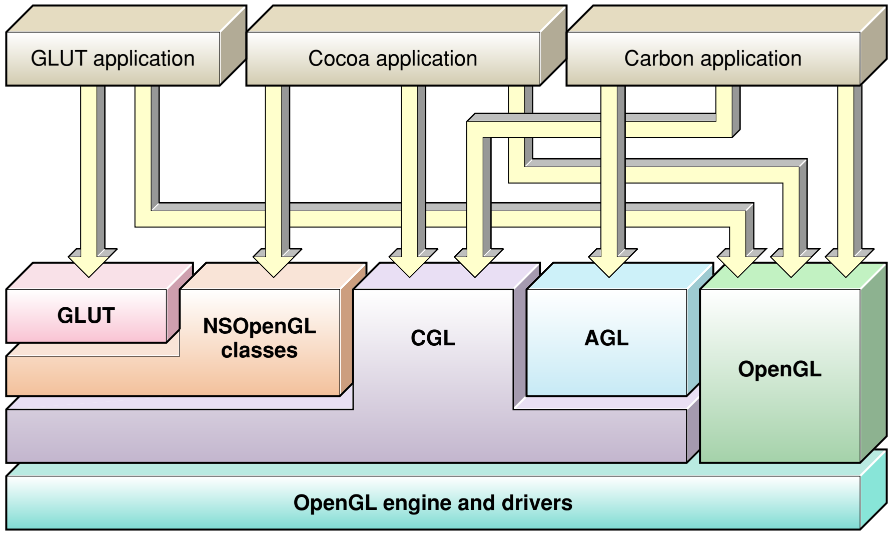
The **driver layer** contains the optional GLD plug-in interface and one or more GLD plug-in drivers, which may have different software and hardware support capabilities. The GLD plug-in interface supports third-party plug-in drivers, allowing third-party hardware vendors to take advantage of newer driver technology.

Programming Interfaces

The programming interfaces that you'll use fall into two categories—those specific to the Macintosh platform and those defined by the OpenGL Architecture Review Board. The Apple-specific programming interfaces are what Cocoa and Carbon applications use to communicate with the Mac OS X windowing system. These APIs don't create OpenGL content, they simply manage content, direct it to a drawing destination (onscreen or offscreen), and control various aspects of the rendering operation. The OpenGL APIs actually create content.

OpenGL routines accept vertex, pixel, and texture data and assemble the data to create content that has the illusion of being three-dimensional. The final content resides in a framebuffer, where it will languish unseen by the user unless your application uses a windowing-system specific API to direct the content onscreen.

Figure 1-3 The programing interfaces used for OpenGL content



OpenGL APIs Specific to Mac OS X

Mac OS X offers three easy-to-use application programming interfaces (APIs) that are specific to the Macintosh platform: the NSOpenGL classes, the AGL API, and the CGL API. Throughout this document, these three APIs are referred to as the Apple-specific OpenGL APIs.

Cocoa provides four classes specifically for OpenGL—`NSOpenGLView`, `NSOpenGLContext`, `NSOpenGLPixelFormat`, and `NSOpenGLPixelBuffer`. The `NSOpenGLView` class provides easy access to a basic OpenGL context that can be set up in Interface Builder. `NSOpenGLView` is a subclass of `NSView` and has the expected facilities to display OpenGL content in a view. `NSOpenGLContext` and `NSOpenGLPixelFormat`, along with `NSView`, are the building blocks for the `NSOpenGLView` class. Applications that subclass `NSOpenGLView` do not need to directly subclass `NSOpenGLPixelFormat` or `NSOpenGLContext`. Applications that need customization or flexibility, can subclass `NSView`. The `NSOpenGLPixelBuffer` class provides hardware-accelerated offscreen drawing. Using the NSOpenGL classes, you can also draw to the full screen.

For detailed information on the NSOpenGL classes, see the following reference documentation:

- *NSOpenGLView Class Reference*
- *NSOpenGLContext Class Reference*
- *NSOpenGLPixelBuffer Class Reference*
- *NSOpenGLPixelFormat Class Reference*

Apple Graphics Library (**AGL**) is the Apple interface to OpenGL for Carbon applications. It can be used by both Mach-O and CFM binaries, although CFM binaries are not recommended in Mac OS X because it's not possible to generate a universal binary with them. (A **universal binary** runs natively on both PowerPC and

Intel-based Macintosh computers.) AGL supports drawing to the full screen as well as to Carbon windows and offscreen locations. In addition to the standard functionality, AGL provides full support for hardware-accelerated offscreen drawing, bitmap font rendering, and the ability to render content directly to a texture (also called **render-to-texture** functionality). The AGL API resides in the AGL framework. Applications must include the `AGL.h` header file (`System/Library/Frameworks/AGL.framework/AGL.h`) to access AGL functionality. *AGL Reference* provides a complete description of this API.

The Core OpenGL API (**CGL**) is the basis for the NSOpenGL classes and AGL. CGL offers the most direct access to system functionality and provides the highest level of graphics performance and control for drawing to the full screen. CGL is windowing-system agnostic but is accessible from both Cocoa and Carbon applications. The CGL API resides in the OpenGL framework. Applications must include the `OpenGL.h` header file (`System/Library/Frameworks/OpenGL.framework/OpenGL.h`) to access CGL functionality. *CGL Reference* provides a complete description of this API.

Apple-Implemented OpenGL Libraries

Mac OS X also provides the full suite of graphics libraries that are part of every implementation of OpenGL: GL, GLU, GLUT, and GLX. Two of these—GL and GLU—provide low-level drawing support. The other two—GLUT and GLX—support drawing to the screen.

Your application typically interfaces directly with the core OpenGL library (GL), the OpenGL Utility library (GLU), and the OpenGL Utility Toolkit (GLUT). The **GL library** provides a low-level modular API that allows you to define graphical objects. It supports the core functions that are common to all OpenGL implementations, as mandated by the OpenGL specification. It provides support for two fundamental types of graphics primitives: objects defined by sets of vertices, such as line segments and simple polygons, and objects that are pixel-based images, such as filled rectangles and bitmaps. The GL API does not handle complex custom graphical objects; your application must decompose them into simpler geometries.

The **GLU library** combines functions from the GL library to support more advanced graphics features. It runs on all conforming implementations of OpenGL. GLU is capable of creating and handling complex polygons (including quartic equations), processing nonuniform rational b-spline curves (NURBs), scaling images, and decomposing a surface to a series of polygons (tessellation).

The **GLUT library** provides a cross-platform API for performing operations associated with the user windowing environment—displaying and redrawing content, handling events, and so on. It is implemented on most UNIX, Linux, and Windows platforms. As such, any code that you write with GLUT can be reused across multiple platforms. However, such code is constrained by a generic set of user interface elements and event-handling options. This book does not show how to use GLUT. If you are interested in GLUT, see the sample code in the ADC Reference Library. GLUT Basics is a simple example that will get you started.

GLX is an OpenGL extension that supports using OpenGL within a window provided by the X Window system. X11 for Mac OS X is available as an optional installation using the Mac OS X installation DVD. (It's not shown in Figure 1-3 (page 17).) See *OpenGL Programming for the X Window System*, published by Addison Wesley for more information.

This document does not show how to use these libraries. For detailed information, either go to the OpenGL Foundation website http://www.opengl.org, or see the most recent version of "The Redbook"—OpenGL Programming Guide, published by Addison Wesley.

Terminology

There are a number of terms that you'll want to understand so that you can write code effectively using OpenGL: renderer, renderer attributes, buffer attributes, pixel format objects, rendering contexts, drawable objects, and virtual screens. As an OpenGL programmer some of these may seem familiar to you. However, understanding the Apple-specific nuances of these terms will help you get the most out of OpenGL on the Macintosh platform.

Renderer

A **renderer** is the combination of the hardware and software that OpenGL uses to create an image from a view and a model. (A software renderer is an exception; it does not use graphics hardware and is typically used as a fallback.) The characteristics of the final image depends on the capabilities of the graphics hardware associated with the renderer and the device used to display the image. A particular renderer supports specific capabilities—for example, the ability to produce environmental effects such as fog.

Mac OS X supports graphics accelerator cards with varying capabilities as well as systems without graphics acceleration hardware. It is possible for multiple renderers, each with different capabilities or features, to drive a single set of graphics hardware.

Renderer and Buffer Attributes

Renderer and buffer attributes are operating system-dependent extensions that communicate to OpenGL the renderer and buffer requirements for your application. The Apple implementation of OpenGL dynamically selects the best renderer for the current rendering task and does so transparently to your application. But, if your application has very specific rendering requirements and wants to control renderer selection, it can do so by supplying the appropriate renderer attributes. Buffer attributes describe such things as color and depth buffer sizes, and whether the data is stereoscopic or monoscopic.

Renderer and buffer attributes are represented by constants defined in the Apple-specific OpenGL APIs. OpenGL uses the attributes you supply to perform the setup work needed prior to drawing content. "Drawing to a Window or View" (page 27) provides simple example that show how to use renderer and buffer attributes. "Techniques for Choosing Attributes" (page 79) provides tips on choosing renderer and buffer attributes to achieve specific rendering goals.

Pixel Format Objects

A **pixel format** describes pixel data storage in memory. The description includes the pixel components (that is, red, blue, green, alpha), the number and order of components, and other relevant information, such as whether a pixel contains stencil and depth values. A **pixel format object** is an opaque data type designed to hold a pixel format along with a list of renderers and display devices that satisfy the requirements specified by an application.

Each of the Apple-specific OpenGL APIs defines a pixel format data type and accessor routines that you can use to obtain the information referenced by this object. See "Virtual Screens" (page 21) for more information on renderer and display devices.

Rendering Contexts

A **rendering context**, or simply context, contains state information for the rendering target of your application. The context affects the rendered result much in the same way that the characteristics of a drawing pen (ink color, point size, type of ink, and so forth) affect what's drawn on a piece of paper. State variables are set per context. Once set, a value remains as such until you change it. State variables include such things as drawing color, the viewing and projection transformations, lighting characteristics, and material properties.

Although your application can maintain more than one context, only one context can be the **current context** in a thread. The current context is the rendering context that receives OpenGL commands issued by your application. The system initializes the context to the default OpenGL state. The context then tracks all state changes made while it is the current context.

Drawable Objects

A **drawable object** refers to an object allocated outside of OpenGL, but that can serve as an OpenGL framebuffer. A drawable object can be the target of OpenGL drawing operations. The behavior of drawable objects is not part of the OpenGL specification. Rather, a drawable object is a platform-specific construct provided by the Mac OS X windowing system.

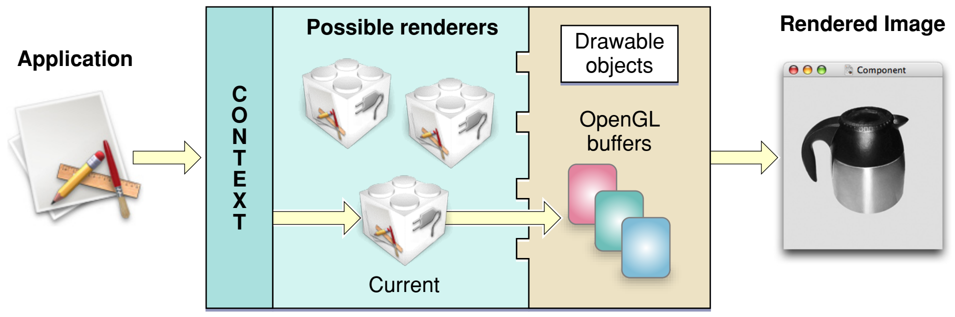
A drawable object can be any of the following: a Carbon window, a Cocoa view, offscreen memory, a full-screen graphics device, or a pixel buffer (available starting in Mac OS X v10.3).

Note: A **pixel buffer** (pbuffer) is an OpenGL buffer designed for hardware-accelerated offscreen drawing and as a source for texturing. An application can render an image into a pixel buffer once and then use the buffer contents multiple times to texture a variety of surfaces without copying the image data.

Before OpenGL can draw to a drawable object, the object must be attached to a rendering context. The characteristics of the drawable object narrow the selection of hardware and software specified by the rendering context. OpenGL automatically allocates buffers, creates surfaces, and specifies which renderer is the current renderer.

The logical flow of data from an application through OpenGL to a drawable object is shown in Figure 1-4. The application issues OpenGL commands that are sent to the current rendering context. The current context, which contains state information, constrains how the commands are interpreted by the appropriate renderer. The renderer converts the OpenGL primitives to an image in the framebuffer. (See also "Running an OpenGL Program in Mac OS X " (page 24).)

Figure 1-4 Data flow through OpenGL



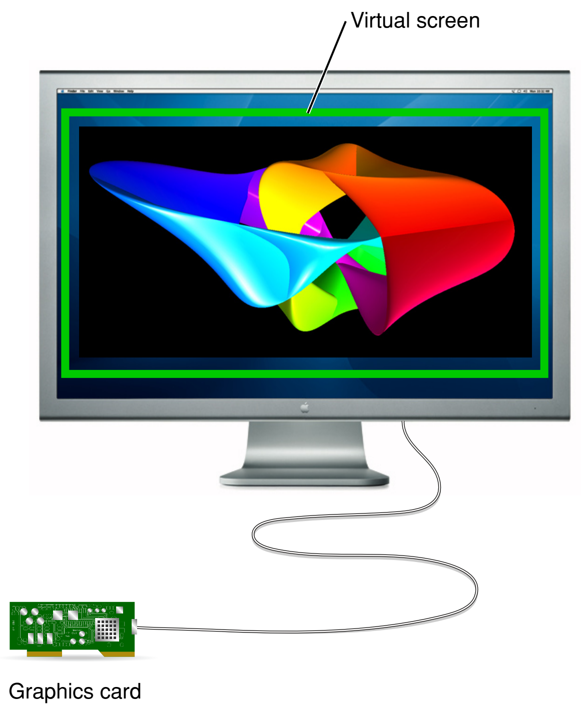
Virtual Screens

The characteristics and quality of the OpenGL content that the user sees depends on both the renderer and the physical display used to view the content. The combination of renderer and physical display is called a **virtual screen**. This important concept has implications for any application that might run on a system that has more than one renderer or more than one display.

A simple system, with one graphics card and one physical display, typically has two virtual screens. One virtual screen consists of a hardware-based renderer and the physical display and the other virtual screen consists of a software-based renderer and the physical display. Mac OS X provides a software-based renderer as a fallback. It's possible for your application to decline the use of this fallback. You'll see how in "Techniques for Choosing Attributes" (page 79).

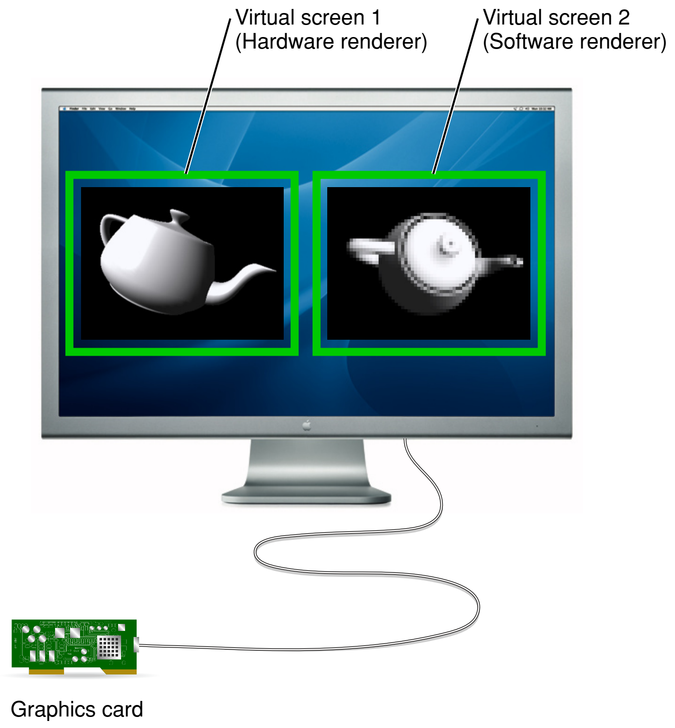
The green rectangle around the OpenGL image in Figure 1-5 surrounds a virtual screen for a system with one graphics card and one display. Note that a virtual screen is not the physical display, which is why the green rectangle is drawn around the application window that shows the OpenGL content. In this case, it is the renderer provided by the graphics card combined with the characteristics of the display.

Figure 1-5 A virtual screen displays what the user sees



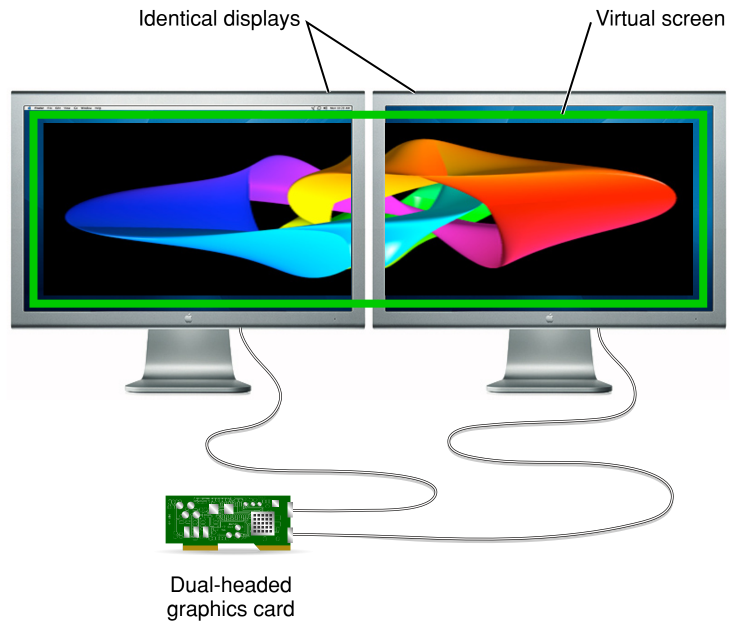
Because a virtual screen is not simply the physical display, a system with one display can use more than one virtual screen at a time, as shown in Figure 1-6. The green rectangles are drawn to point out each virtual screen. Imagine that the virtual screen on the right side uses a software-only renderer and that the one on the left uses a hardware-dependent renderer. Although this is a contrived example, it illustrates the point.

Figure 1-6 Two virtual screens



It's also possible to have a virtual screen that can represent more than one physical display. The green rectangle in Figure 1-7 is drawn around a virtual screen that spans two physical displays. In this case, the same graphics hardware drives a pair of identical displays. This is also true when mirroring is enabled.

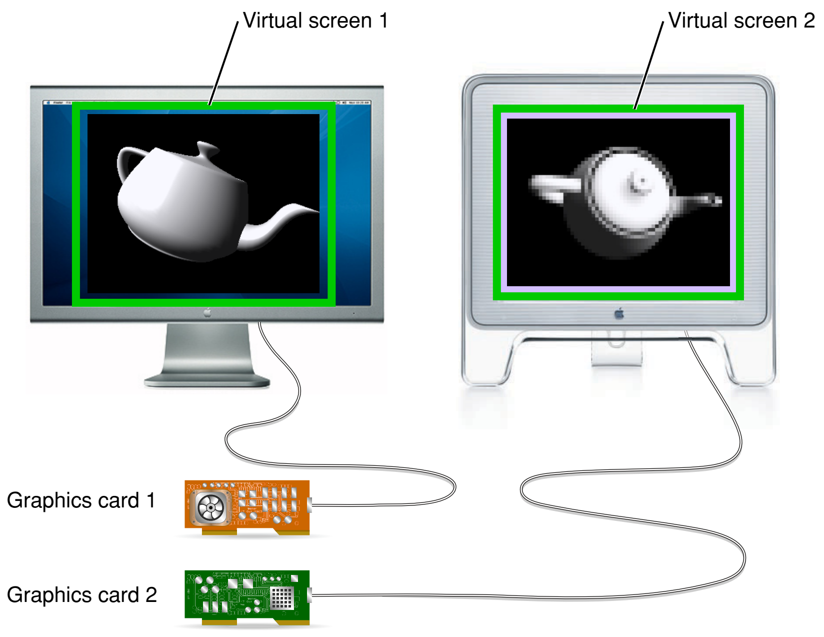
Figure 1-7 A virtual screen can represent more than one physical screen



The concept of a virtual screen is particularly important when the user drags an image from one physical screen to another. When this happens, the virtual screen may change, and with it, a number of attributes of the imaging process, such as the current renderer, may change. With the dual-headed graphics card shown in Figure 1-7 (page 22), dragging between displays preserves the same virtual screen. However, Figure 1-8 shows the case for which two displays represent two unique virtual screens. Not only are the two graphics cards different, but it's possible that the renderer, buffer attributes, and pixel characteristics are different. A change in any of these three items can result in a change in the virtual screen.

When the user drags an image from one display to another, and the virtual screen is the same for both displays, the image quality should appear similar. However, for the case shown in Figure 1-8, the image quality can be quite different.

Figure 1-8 Two virtual screens and two graphics cards



OpenGL for Mac OS X transparently manages rendering across multiple monitors. A user can drag a window from one monitor to another, even though their display capabilities may be different or they may be driven by dissimilar graphics cards with dissimilar resolutions and color depths.

OpenGL dynamically switches renderers when the virtual screen that contains the majority of the pixels in an OpenGL window changes. When a window is split between multiple virtual screens, the framebuffer is rasterized entirely by the renderer driving the screen that contains the largest segment of the window. The regions of the window on the other virtual screens are drawn by copying the rasterized image. When the entire OpenGL drawable object is displayed on one virtual screen, there is no performance impact from multiple monitor support.

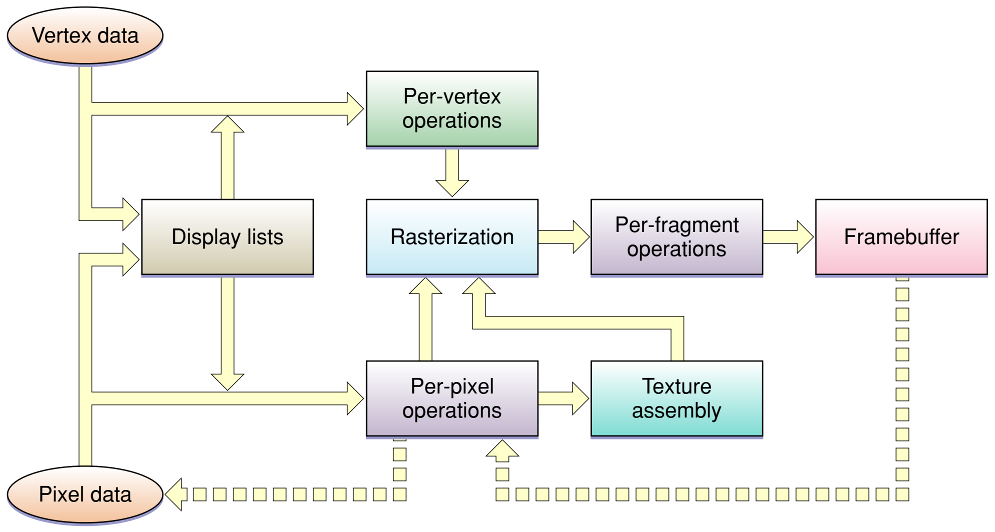
Applications need to track virtual screen changes and, if appropriate, update the current application state to reflect changes in renderer capabilities. See "Techniques for Working with Rendering Contexts" (page 65).

Running an OpenGL Program in Mac OS X

Figure 1-9 shows the flow of data in an OpenGL program, regardless of the platform that the program runs on. Pixel data and vertex data can be sent to OpenGL for processing in two ways. The first is by issuing OpenGL commands that are executed immediately, either to assemble a model from vertex data or a texture from pixel data. When an application issues OpenGL commands that are executed immediately, OpenGL is said to be operating in **immediate mode**. There are two immediate mode paths in the figure: one from vertex data to per-vertex operations and the other from pixel data to per-pixel operations.

The "display lists" rectangle in the figure represents the second way that an application can send data to OpenGL. A **display list** is a set of OpenGL commands that is assembled and named by an application. The display list is then stored on the OpenGL server. The application can refer to the list by its assigned name when the data defined by the list is needed. Display lists are ideal for computing-intensive operations because at the time you need to use the data, it is already uploaded to the GPU and is usually preprocessed. There are two display list paths in the figure, one for vertex data and one for pixel data.

Figure 1-9 The flow of data through OpenGL



Per-vertex operations include such things as applying transformation matrices to add perspective or to clip and applying lighting effects. Per-pixel operations include such things as color conversion and applying blur and distortion effects. Pixels destined for textures are sent to texture assembly where OpenGL stores textures until it needs to apply them onto an object.

OpenGL rasterizes the processed vertex and pixel data, meaning that the data are converged to create fragments. A fragment encapsulates all the values for a pixel, including color, depth, and sometimes texture values. These values are used during anti-aliasing and any other calculations needed to fill shapes and to connect vertices.

Per-fragment operations include applying environment effects, depth and stencil testing, and performing other operations such as blending and dithering. Some operations—such as hidden-surface removal—end the processing of a fragment. OpenGL draws fully processed fragments into the appropriate location in the framebuffer.

The dashed arrows in Figure 1-9 indicate reading pixel data back from the framebuffer. They represent operations performed by OpenGL functions such as `glReadPixels`, `glCopyPixels`, and `glCopyTexImage2D`.

So far you've seen how OpenGL operates on any platform. But how do Cocoa and Carbon applications provide data to the OpenGL for processing? Regardless of the application environment (Cocoa or Carbon), a Mac OS X application must perform these tasks:

- Set up a list of buffer and renderer attributes that define the sort of drawing you want to perform. (See "Renderer and Buffer Attributes" (page 19).)
- Request the system to create a pixel format object that contains a pixel format that meets the constraints of the buffer and render attributes and a list of all suitable combinations of displays and renderers. (See "Pixel Format Objects" (page 19) and "Virtual Screens" (page 21).)
- Create a rendering context to hold state information that controls such things as drawing color, view and projection matrices, characteristics of light, and conventions used to pack pixels. When you set up this context, you must provide a pixel format object because the rendering context needs to know the set of virtual screens that can be used for drawing. (See "Rendering Contexts" (page 20).)
- Bind a drawable object to the rendering context. The drawable object is what captures the OpenGL drawing sent to that rendering context. (See "Drawable Objects" (page 20).)
- Make the rendering context the current context. OpenGL automatically targets the current context. Although your application might have several rendering contexts set up, only the current one is the active one for drawing purposes.
- Issue OpenGL drawing commands. If you've completed the previous tasks, the contents of the framebuffer shown in Figure 1-9 are drawn to the drawable object that's attached to the current rendering context.

The tasks described in the first five bullet items are platform-specific. "Drawing to a Window or View" (page 27) provides simple examples of how to perform them. As you read other parts of this document, you'll see there are a number of other tasks that, although not mandatory for drawing, are really quite necessary for any application that wants to use OpenGL to perform complex 3D drawing efficiently on a wide variety of Macintosh systems.

See Also

Reference documentation for the Apple-specific OpenGL programming interfaces:

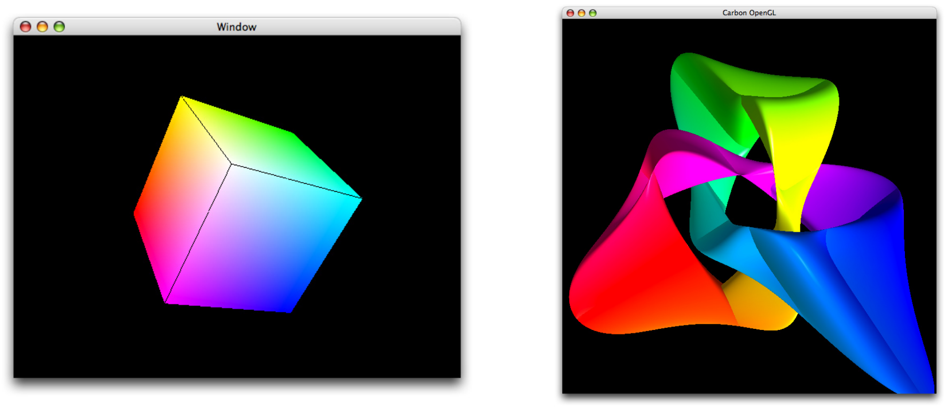
- *AGL Reference*
- *CGL Reference*
- *NSOpenGLContext Class Reference*
- *NSOpenGLPixelBuffer Class Reference*
- *NSOpenGLPixelFormat Class Reference*
- *NSOpenGLView Class Reference*

The Apple Developer Connection OpenGL technology page links to high-level technical articles on OpenGL and Mac OS X.

Drawing to a Window or View

The OpenGL programming interface provides hundreds of drawing commands that drive graphics hardware. It doesn't provide any commands that interface with the windowing system of an operating system. Without a windowing system, the 3D graphics of an OpenGL program are trapped inside the GPU. Figure 2-1 shows a cube drawn to a Cocoa view and a trefoil drawn to a Carbon window. (You can just as easily draw the trefoil to the Cocoa view and the cube to the Carbon window.)

Figure 2-1 OpenGL content in a Cocoa view (left) and a Carbon window (right)



This chapter shows how to display OpenGL drawing onscreen using the APIs provided by Mac OS X. You'll see how to draw to Cocoa views and Carbon windows. (This chapter does not show how to use GLUT.) The first section describes the overall approach to drawing onscreen and provides an overview to the functions and methods used by each API. You'll want to read this regardless of the application framework that you use. The remaining sections in the chapter provide information that's specific to Cocoa or Carbon. After you consult the appropriate section, take a look at "What's Next" (page 36) for pointers to optimization strategies and other information that will help your OpenGL application to perform at its best.

General Approach

Mac OS X provides three interfaces for drawing OpenGL content onscreen: the NSOpenGL classes, AGL, and CGL. (See "Programming Interfaces" (page 16) for more information). You use the NSOpenGL classes from within the Cocoa application framework, while AGL is the interface that supports drawing OpenGL content to a Carbon application. CGL can be used from either a Cocoa or Carbon application. For drawing to a view or a window, you'll either use the NSOpenGL classes (for a Cocoa view) or AGL (for a Carbon window), because CGL supports drawing only to the full screen.

Regardless of the application framework, to draw OpenGL content to a window or view, you need to perform these tasks:

1. Set up the renderer and buffer attributes that support the OpenGL drawing you want to perform.

Each of the OpenGL APIs in Mac OS X has its own set of constants that represent renderer and buffer attributes. For example, the all-renderers attribute is represented by the `NSOpenGLPFAAllRenderers` constant in Cocoa and the `AGL_ALL_RENDERERS` constant in the AGL API.

2. Request, from the operating system, a pixel format object that encapsulates pixel storage information and the renderer and buffer attributes required by your application. The returned pixel format object contains all possible combinations of renderers and displays available on the system that your program runs on and that meets the requirements specified by the attributes. The combinations are referred to as virtual screens. (See "Virtual Screens" (page 21).)

   There may be situations for which you want to ensure that your program uses a specific renderer. "Techniques for Choosing Attributes" (page 79) discusses how to set up an attributes array that will guarantee the system passes back a pixel format object that uses only that renderer.

   You'll need to provide code that handles the case of getting back a `NULL` pixel format object.

3. Create a rendering context and bind the pixel format object to it. The rendering context keeps track of state information that controls such things as drawing color, view and projection matrices, characteristics of light, and conventions used to pack pixels.

   Your application needs a pixel format object to create a rendering context.

4. Release the pixel format object. Once the pixel format object is bound to a rendering context, its resources are no longer needed.

5. Bind a drawable object to the rendering context. You'll either bind a Cocoa view or a Carbon window to the context.

6. Make the rendering context the current context. The system sends OpenGL drawing to whichever rendering context is designated as the current one. It's possible for you to set up more than one rendering context, so you'll need to make sure that the one you want to draw to is the current one.

7. Perform your drawing.

The specific functions or methods that you use to perform each of the steps are discussed in the sections that follow.

Drawing to a Cocoa View

There are two ways to draw OpenGL content to a Cocoa view. You can either use the `NSOpenGLView` class or create a custom `NSView` class. If your application has modest drawing requirements, then you can use the `NSOpenGLView` class. For example, if your application draws to a single view and does not support dragging the view between monitors, you can use the `NSOpenGLView` class. See "Drawing to an NSOpenGLView Class: A Tutorial."

If your application is more complex and needs to support drawing to multiple rendering contexts, you may want to consider subclassing the `NSView` class. For example, if your application supports drawing to multiple views at the same time, you'll need to set up a custom `NSView` class. See "Drawing OpenGL Content to a Custom View" (page 31).

Drawing to an NSOpenGLView Class: A Tutorial

The `NSOpenGLView` class is a lightweight subclass of the `NSView` class that provides convenience methods for setting up OpenGL drawing. An `NSOpenGLView` object maintains an `NSOpenGLPixelFormat` object and an `NSOpenGLContext` object into which OpenGL calls can be rendered. It provides methods for accessing and managing the pixel format object and the rendering context, and handles notification of visible region changes.

An `NSOpenGLView` object does not support subviews. You can, however, divide the view into multiple rendering areas using the OpenGL function `glViewport`.

This section provides step-by-step instructions for creating a simple Cocoa application that draws OpenGL content to a view. The tutorial assumes that you know how to use Xcode and Interface Builder. If you have never created an application using the Xcode development environment, see Getting Started with Tools.

1. Open Xcode and create a Cocoa application project named Golden Triangle.
2. Open the Frameworks folder in the Groups & File list. Then select the Linked Frameworks folder.
3. Choose Project > Add to Project and navigate to the OpenGL framework, which is located in the `System/Library/Frameworks` directory. In the sheet that appears, choose `OpenGL.framework` and click Add. Then, in the next sheet that appears, click Add to add the framework to the target.
4. Choose File > New File. Then choose the Objective-C class template.
5. Click Next and name the file `MyOpenGLView.m`. Make sure the checkbox to create `MyOpenGLView.h` is selected. Then click Finish.
6. Open the `MyOpenGLView.h` file and modify the file so that it looks like the code shown in Listing 2-2 to declare the interface.

Listing 2-1 The interface for `MyOpenGLView`

```
#import <Cocoa/Cocoa.h>

@interface MyOpenGLView : NSOpenGLView
{
}
- (void) drawRect: (NSRect) bounds;
@end
```

7. Save and close the `MyOpenGLView.h` file.
8. Open the `MyOpenGLView.m` file and include the `gl.h` file, as shown in Listing 2-3.

Listing 2-2 Include `OpenGL/gl.h`

```
#import "MyOpenGLView.h"
#include <OpenGL/gl.h>

@implementation MyOpenGLView
@end
```

9. Implement the `drawRect:` method as shown in Listing 2-3, adding the code after the `@implementation` statement. The method sets the clear color to black and clears the color buffer in preparation for drawing. Then, `drawRect:` calls your drawing routine, which you'll add next. The OpenGL command `glFlush` draws the content provided by your routine to the view.

Listing 2-3 The `drawRect:` method for `MyOpenGLView`

```
-(void) drawRect: (NSRect) bounds
{
    glClearColor(0, 0, 0, 0);
    glClear(GL_COLOR_BUFFER_BIT);
    drawAnObject();
    glFlush();
}
```

10. Add the code to perform your drawing. In your own application, you'd perform whatever drawing is appropriate. But for the purpose of learning how to draw OpenGL content to a view, you'll add the code shown in Listing 2-4. This code draws a 2-dimensional, gold-colored triangle, whose dimensions are not quite the dimensions of a true golden triangle, but good enough to show how to perform OpenGL drawing.

    Make sure that you insert this routine before the drawRect: method in the `MyOpenGLView.m` file.

Listing 2-4 Code that draws a triangle using OpenGL commands

```
static void drawAnObject ()
{
    glColor3f(1.0f, 0.85f, 0.35f);
    glBegin(GL_TRIANGLES);
    {
        glVertex3f(  0.0,  0.6, 0.0);
        glVertex3f( -0.2, -0.3, 0.0);
        glVertex3f(  0.2, -0.3 ,0.0);
    }
    glEnd();
}
```

11. In the File Name list, double click the `MainMenu.xib` file to open Interface Builder. A default menu bar and window titled "Window" appears when the file opens.

12. Click the window and choose Tools > Inspector.

13. In the Window Attributes pane of the inspector window, change the Title entry to `Golden Triangle`.

14. Choose Tools > Library and type `NSOpenGLView` in the Search field.

15. Drag an `NSOpenGLView` object from the Library to the window. Resize the view to fit the window.

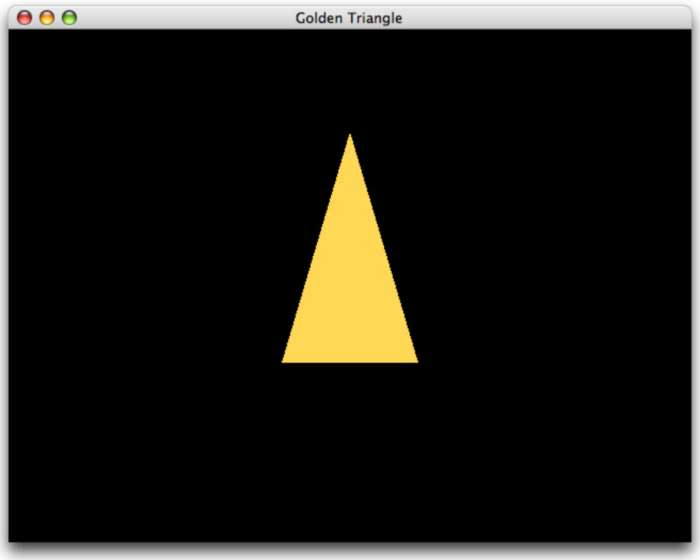
16. In the Identity pane of the inspector for the view, choose `MyOpenGLView` from the Class pop-up menu.

17. Open the Attributes pane of the inspector for the view, and take a look at the renderer and buffer attributes that are available to set. These settings save you from setting attributes programmatically.

    Only those attributes listed in the Interface Builder inspector are set when the view is instantiated. If you need additional attributes, you'll need to set them programmatically.

18. Choose File > Build & Go in Xcode. You should see content similar to the triangle shown in Figure 2-2.

Figure 2-2 The output from the Golden Triangle program



This example is extremely simple. In a more complex application, you'd want to do the following:

- In the interface for the view, declare a variable that indicates whether the view is ready to accept drawing. A view is ready for drawing only if it is bound to a rendering context and that context is set to be the current one.
- Cocoa does not call initialization routines for objects created in Interface Builder. If you need to perform any initialization tasks, do so in the `awakeFromNib` method for the view. Note that because you set attributes in the inspector, there is no need to set them up programmatically unless you need additional ones. There is also no need to create a pixel format object programmatically; it is created and loaded when Cocoa loads the nib file.
- Your `drawRect:` method should test whether the view is ready to draw into. You need to provide code that handles the case when the view is not ready to draw into.
- OpenGL is at its best when doing real-time and interactive graphics. Your application will need to provide a timer or support user interaction.

Drawing OpenGL Content to a Custom View

This section provides an overview of the key tasks you need to perform to customize the `NSView` class for OpenGL drawing. Before you create a custom view for OpenGL drawing, you should read Creating a Custom View in *View Programming Guide for Cocoa*. You will also want to download Custom Cocoa OpenGL (available on the ADC website from Sample Code > Graphics & Imaging > OpenGL), which is a full-featured OpenGL sample application that uses a custom subclass of `NSView` that behaves similarly to the `NSOpenGLView` class. The custom class is declared and defined in the `CustomOpenGLView.h` and `CustomOpenGLView.m` files. After you've set up your custom class, you can use it just as you would use the built-in `NSOpenGLView` class.

When you subclass the `NSView` class to create a custom view for OpenGL drawing, you'll override any Quartz drawing or other content that is in that view. To set up a custom view for OpenGL drawing, subclass `NSView` and create two private variables—one which is an `NSOpenGLContext` object and the other an `NSOpenGLPixelFormat` object, as shown in Listing 2-5.

Listing 2-5 The interface for a custom OpenGL view

```
@class NSOpenGLContext, NSOpenGLPixelFormat;

@interface CustomOpenGLView : NSView
{
  @private
    NSOpenGLContext*     _openGLContext;
    NSOpenGLPixelFormat* _pixelFormat;
}
+ (NSOpenGLPixelFormat*)defaultPixelFormat;
- (id)initWithFrame:(NSRect)frameRect pixelFormat:(NSOpenGLPixelFormat*)format;
- (void)setOpenGLContext:(NSOpenGLContext*)context;
- (NSOpenGLContext*)openGLContext;
- (void)clearGLContext;
- (void)prepareOpenGL;
- (void)update;
- (void)setPixelFormat:(NSOpenGLPixelFormat*)pixelFormat;
- (NSOpenGLPixelFormat*)pixelFormat;
@end
```

In addition to the usual methods for the private variables (`openGLContext`, `setOpenGLContext:`, `pixelFormat`, and `setPixelFormat:`) you'll need to implement the following methods:

- `+ (NSOpenGLPixelFormat*) defaultPixelFormat`

  Use this method to allocate and initialize the `NSOpenGLPixelFormat` object.
- `- (void) clearGLContext`

  Use this method to clear and release the `NSOpenGLContext` object.
- `- (void) prepareOpenGL`

  Use this method to initialize the OpenGL state after creating the `NSOpenGLContext` object.

You need to override the `update` and `initWithFrame:` methods of the `NSView` class.

- `update` calls the `update` method of the `NSOpenGLContext` class.
- `initWithFrame:pixelFormat` retains the pixel format and sets up the notification `NSViewGlobalFrameDidChangeNotification`. See Listing 2-6.

If the custom view is not guaranteed to be in a window, you must also override the `lockFocus` method of the `NSView` class. See Listing 2-7. This method makes sure that the view is locked prior to drawing and that the context is the current one.

Listing 2-6 The `initWithFrame:pixelFormat:` method

```
- (id)initWithFrame:(NSRect)frameRect pixelFormat:(NSOpenGLPixelFormat*)format
{
    self = [super initWithFrame:frameRect];
    if (self != nil) {
        _pixelFormat   = [format retain];
    [[NSNotificationCenter defaultCenter] addObserver:self
                     selector:@selector(_surfaceNeedsUpdate:)
                     name:NSViewGlobalFrameDidChangeNotification
                     object:self];
    }
    return self;
}

- (void) _surfaceNeedsUpdate:(NSNotification*)notification
{
   [self update];
}
```

Listing 2-7 The `lockFocus` method

```
- (void)lockFocus
{
    NSOpenGLContext* context = [self openGLContext];

    [super lockFocus];
    if ([context view] != self) {
        [context setView:self];
    }
    [context makeCurrentContext];
}
```

The `reshape` method is not supported by the `NSView` class. You need to update bounds in the `drawRect:` method, which should take the form shown in Listing 2-8.

Listing 2-8 The `drawRect` method for a custom view

```
-(void) drawRect
{
    [context makeCurrentContext];
    //Perform drawing here
    [context flushBuffer];
}
```

There may be other methods that you want to add. For example, you might consider detaching the context from the drawable object when the custom view is moved from the window, as shown in Listing 2-9.

Listing 2-9 Detaching the context from a drawable object

```
-(void) viewDidMoveToWindow
{
    [super viewDidMoveToWindow];
    if ([self window] == nil)
        [context clearDrawable];
}
```

Drawing to a Carbon Window

This section describes the steps for setting up onscreen drawing to a Carbon window. To get an idea of how these steps fit into an full application, you should look at the sample application *GLCarbonAGLWindow*.

Follow these steps to use the AGL API to set up onscreen drawing to a Carbon window:

1. Set up an array of attributes that describes the buffer characteristics and renderer capabilities that you want. You can supply any of the pixel format attributes or extended attributes defined in AGL Constants in *AGL Reference*.

   This example in Listing 2-10 (page 35) sets up attributes for RGBA, double buffering, and a pixel depth of 24 bits. Your code would set up whatever attributes are appropriate. In later chapters in this book, you'll see how to choose attributes for specific purposes. (See "Techniques for Choosing Attributes" (page 79).)

2. Obtain a pixel format object by passing the attributes array to the function `aglChoosePixelFormat`.

   The pixel format object contains a list of all appropriate renderer-display combinations. In the example shown here, it's likely that the list will contain at least two items—one that uses a hardware renderer and another that uses a software renderer.

3. Bind the pixel format object to a rendering context by passing the pixel format object to the function `aglCreateContext`.

   If the pixel format object has more than one pixel format (renderer-display combination) in it, AGL uses the first in the list. You can call the function `aglNextPixelFormat` if you want to use the next pixel format in the list.

4. Release the pixel format object by calling the function `aglDestroyPixelFormat`.

5. Get the port associated with the Carbon window that you want to draw into by calling the Window Manager function `GetWindowPort`. After you attach a rendering context to the Carbon window, its viewport is set to the full size of the window.

   Note: The AGL API for drawing to a Carbon window was developed prior to Mac OS X. Because of this heritage, the `AGLDrawable` data type is a `CGrafPtr` data type under the hood. That's why you must call `GetWindowPort` to obtain the associated graphics port from the `WindowRef` data type passed to `MySetWindowAsDrawableObject`.

6. Bind the window to the rendering context by passing the port to the function `aglSetDrawable`.

7. Make the rendering context the current context by calling function `aglSetCurrentContext`.

Listing 2-10 shows how to implement these steps and how to check for errors along the way by calling the application-defined function `MySetWindowAsDrawableObject`. It's recommended that your application provides a similar error-checking function. In the case of an error you'll either want to notify the user and abort the program or take some sort of fallback action that ensures you application can draw OpenGL content. (See "Ensuring a Valid Pixel Format Object" (page 80) for an example of backing out of attributes. See "Retrieve Error Information Only When Debugging" (page 127) for guidelines on error checking and performance.)

Note that the example passes the pixel format object returned from the `aglChoosePixelFormat` function to the function `aglCreateContext`. By default, AGL uses the first pixel format in the pixel format object regardless of how many pixel formats are actually in the object. You can iterate through the pixel format object using the function`aglNextPixelFormat`.

Listing 2-10 Setting a Carbon window as a drawable object

```
OSStatus MySetWindowAsDrawableObject  (WindowRef window)
{
    OSStatus err = noErr;
    Rect rectPort;
    GLint attributes[] =  { AGL_RGBA,
                        AGL_DOUBLEBUFFER,
                        AGL_DEPTH_SIZE, 24,
                        AGL_NONE };
    AGLContext  myAGLContext = NULL;
    AGLPixelFormat myAGLPixelFormat;

    myAGLPixelFormat = aglChoosePixelFormat (NULL, 0, attributes);
    err = MyAGLReportError ();
    if (myAGLPixelFormat) {
        myAGLContext = aglCreateContext (myAGLPixelFormat, NULL);
        err = MyAGLReportError ();
    }
    if (! aglSetDrawable (myAGLContext, GetWindowPort (window)))
            err = MyAGLReportError ();
    if (!aglSetCurrentContext (myAGLContext))
            err = MyAGLReportError ();
    return err;
}

OSStatus MyAGLReportError (void)
{
    GLenum err = aglGetError();
    if (AGL_NO_ERROR != err) {
        char errStr[256];
        sprintf (errStr, "AGL: %s",(char *) aglErrorString(err));
        reportError (errStr);
    }
    if (err == AGL_NO_ERROR)
        return noErr;
    else
        return (OSStatus) err;
}
```

Note: Although this example shows how to draw OpenGL content to an entire Carbon window, it is possible for Carbon applications to draw to a part of a window. Carbon developers can find additional information on using windows by reading *Handling Carbon Windows and Controls*.

What's Next

After you've successfully drawn OpenGL content onscreen from within a Cocoa or a Carbon application, you'll want to move on to more complex tasks. Most 3D applications have sophisticated needs, especially with regard to performance and the need to ensure that the application works with a variety of graphics cards and displays. Some of the chapters that follow will help you to fine tune your code. Other chapters provide guidance and code examples for accomplishing common tasks, such as checking for OpenGL functionality or using images as textures.

See Also

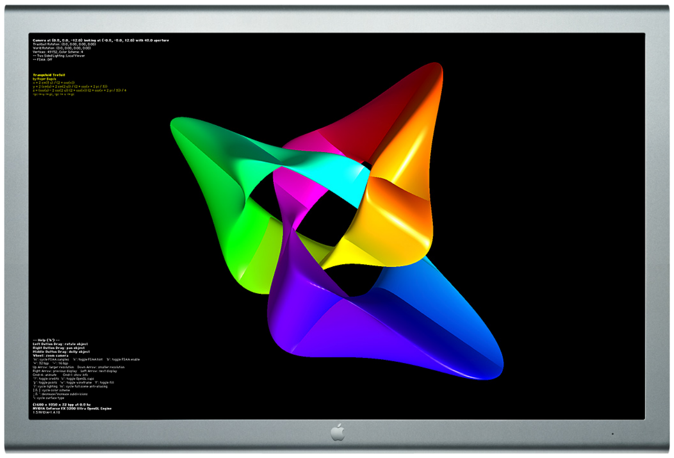
OpenGL sample code projects (ADC Reference Library):

- *Cocoa OpenGL* sets up a window and handles events for drawing OpenGL content to a Cocoa view.
- *Custom Cocoa OpenGL* uses a custom view in Cocoa for OpenGL drawing.
- *GLCarbonAGLWindow* contains code that sets up a Carbon window for OpenGL drawing, handles events, and has a virtual trackball as well as a number of other features.

Drawing to the Full Screen

In Mac OS X, you don't have to restrict your OpenGL drawing to views and windows. You also have the option to draw to the entire screen. The primary difference between drawing to a view or window and drawing to the full screen is that you must prevent other applications and system services from trying to do the same thing. You can capture the display by using the Quartz Display Services API. Once captured by your application, other applications are not notified of display changes, thus preventing them from repositioning their windows and preventing the Finder from repositioning desktop icons. The screen is all yours for OpenGL drawing.

Figure 3-1 Drawing OpenGL content to the full screen



Each of the Apple-specific OpenGL APIs provides routines for setting up full-screen drawing. The approach for using each is similar, as you'll see by reading the first section in this chapter, which describes the general approach. This chapter also provides specific information for using each of the Apple-specific OpenGL APIs and shows how to use Quartz Display Services to switch the display mode and change screen resolutions, two tasks that are useful for any application that uses the full screen.

General Approach

Many of the tasks for setting up full-screen drawing are similar to those required to set up drawing OpenGL content to a Cocoa view or a Carbon window. The tasks that are similar are explained in detail in "Drawing to a Window or View" (page 27) but only mentioned here. If you haven't read that chapter, you should read it first.

Drawing OpenGL content to a full screen requires performing the following tasks:

1. Capture the display you want to draw to by calling the Quartz Display Services function `CGDisplayCapture` and supplying a display ID that represents a unique ID for an attached display. The constant `kCGDirectMainDisplay` represents the main display, the one that's shown in the menu bar.

If you want to capture all the displays attached to a system, call the function `CGDisplayCaptureAllDisplays`.

2. Convert the display ID to an OpenGL display mask by calling the function `CGDisplayIDToOpenGLDisplayMask`.
3. Set up the renderer and buffer attributes that support the OpenGL drawing you want to perform, making sure to include a full-screen attribute and the OpenGL display mask that you obtained in the previous step.
4. Request a pixel format object that encapsulates the renderer and buffer attributes required by your application.

   Some OpenGL renderers, such as the software renderer, do not support full-screen mode. If the system returns `NULL` for the pixel format object, your application needs to take appropriate action.
5. Create a rendering context and bind the pixel format object to it.
6. Release the pixel format object.
7. Make the context the current context.
8. Bind a full-screen drawable object to the rendering context.
9. Perform your drawing.
10. When you are done drawing, perform the necessary cleanup work and make sure that you release the captured display.

Using Cocoa to Create a Full-Screen Context

When you set up an attributes array, you need to include the attribute `NSOpenGLPFAFullScreen` to specify that only renderers that are capable of rendering to the full screen should be considered when the system creates a pixel format object. You also need to include the attribute `NSOpenGLPFAScreenMask` along with the appropriate OpenGL display mask.

Listing 3-1 is a code fragment that shows how to use the `NSOpenGLPixelFormat` and `NSOpenGLContext` classes along with calls from Quartz Display Services to set up full-screen drawing in a Cocoa application. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-1 Using Cocoa to set up full-screen drawing

```
CGDisplayErr err;
NSOpenGLContext *fullScreenContext;
NSOpenGLPixelFormatAttribute attrs[] = {                              // 1
    NSOpenGLPFAFullScreen,
    NSOpenGLPFAScreenMask,
                CGDisplayIDToOpenGLDisplayMask(kCGDirectMainDisplay),
    NSOpenGLPFAColorSize, 24,                                         // 2
    NSOpenGLPFADepthSize, 16,
    NSOpenGLPFADoubleBuffer,
    NSOpenGLPFAAccelerated,
    0
};
NSOpenGLPixelFormat *pixelFormat = [[NSOpenGLPixelFormat alloc]
                                initWithAttributes:attrs];
fullScreenContext = [[NSOpenGLContext alloc] initWithFormat:pixelFormat
                         shareContext:NULL];
[pixelFormat release];
pixelFormat = nil;
if (fullScreenContext == nil) {
        NSLog(@"Failed to create fullScreenContext");
        return;
}
err = CGCaptureAllDisplays();                                        // 3
if (err != CGDisplayNoErr) {
        [fullScreenContext release];
        fullScreenContext = nil;
        return;
}
[fullScreenContext setFullScreen];                                   // 4
[fullScreenContext makeCurrentContext];                              // 5
```

Here's what the code does:

1. Sets up an array of renderer and buffer attributes, including the appropriate attributes to specify full-screen mode and the display ID for the main display. This example also supplies a number of other attributes. You would supply the attributes that are appropriate for your application.

2. Supplies a color size that matches the current display depth. Note that this value must match the current display depth.

3. Calls the Quartz Display Services function that captures all displays. If you want to capture only one display, you can call the function `CGDisplayCapture`, passing the ID of the display that you want to capture.

4. Attaches the full-screen drawable object to the rendering context.

5. Makes the full-screen context the current context that will receive OpenGL commands. If you fail to perform this step, you won't see any content drawn to the screen.

When you no longer need to draw full-screen OpenGL content, you must release resources and release the captured display (or displays).

Using AGL to Create a Full-Screen Context

This extended code example is an excerpt from an application that uses an application-defined structure—`pRecContext`—to store information about the context, including display IDs for the displays attached to the system and a rendering context. The `MySetupAGL` routine in Listing 3-2 takes as parameters a `pRecContext` data type, a width and height that specifies the screen resolution, a bit depth, and the refresh rate of the display.

The `MySetupAGL` routine sets the display mode and sets up a full-screen context. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-2 A function that sets up a full-screen context using AGL

```
OSStatus MySetupAGL (pRecContext pContextInfo, size_t width, size_t height,
                        size_t depth, CGRefreshRate refresh)
{
    OSStatus err = noErr;
    GLint attribs[] = { AGL_RGBA, AGL_NO_RECOVERY,
            AGL_FULLSCREEN, AGL_DOUBLEBUFFER,
            AGL_DEPTH_SIZE, 32,
                        0 };                                              // 1
    AGLPixelFormat pixelFormat = NULL;
    long i, index;
    GDHandle gdhDisplay;
    CFDictionaryRef refDisplayMode = 0;

    if (NULL == pContextInfo)
        return paramErr;
    refDisplayMode = CGDisplayBestModeForParametersAndRefreshRate(
                            pContextInfo->display,
                            depth, width, height, refresh, NULL);         // 2
    if (refDisplayMode) {
        gOldDisplayMode = CGDisplayCurrentMode( pContextInfo->display);   // 3
        gOldDisplayModeValid = GL_TRUE;
        CGDisplaySwitchToMode (pContextInfo->display, refDisplayMode);    // 4
    }
    for (i = 0; i < gNumDisplays; i++) {                                  // 5
        if (pContextInfo->display == gDisplayCaps[i].cgDisplayID) index = i;
    }
    err = DMGetGDeviceByDisplayID ((DisplayIDType)pContextInfo->display,
                                 &gdhDisplay, false);                     // 6
    if (noErr == err)
        if (!(pixelFormat = aglChoosePixelFormat (&gdhDisplay, 1, attribs)))
            err = aglReportError ();
    if (pixelFormat) {
        if (!(pContextInfo->aglContext = aglCreateContext( pixelFormat,
                                                       NULL)))
            err = aglReportError ();
        aglDestroyPixelFormat (pixelFormat);
    }
    if (pContextInfo->aglContext) {
        short fNum;
        GLint swap = 1;
        if (!aglSetCurrentContext (pContextInfo->aglContext))             // 7
            err = aglReportError ();
        if ((noErr == err) && !aglSetFullScreen( pContextInfo->aglContext,
                                    0, 0, 0, 0))                          // 8
            err = aglReportError ();
        if (noErr == err) {
            if (!aglSetInteger (pContextInfo->aglContext,
                                AGL_SWAP_INTERVAL, &swap));               // 9
                err = aglReportError ();
            /* Your code to perform other initializations here */
        }
    }
    return err;
}
```

Here's what the code does:

1. Sets up renderer and buffer attributes. You must supply `AGL_FULLSCREEN` when you want to set up a full-screen context using the AGL API. This example also provides a number of other attributes: RGBA pixel format, double buffering, a depth size of 32 bits, and the no recovery attribute. No recovery indicates that if a suitable hardware renderer isn't found, the operating system should not substitute a software renderer.

2. Obtains the best display mode for the screen resolution, bit depth, and refresh rate passed to the `MySetupAGL` function.

3. Gets the current display mode and then saves it so that it can be restored later. It's recommended practice for you to save and restore the display mode.

4. Switches to the display mode.

5. Gets the display capabilities of interest for current display. For more information on determining the capabilities of a display, see "Determining the OpenGL Capabilities Supported by the Hardware" (page 59).

6. Calls the Display Manager function that obtains a handle for the video device with the specified display ID. You must pass this handle to `aglChoosePixelFormat`.

7. Sets the current context to the newly created context. If you fail to perform this task, you won't see any OpenGL content drawn on the screen.

8. Attaches the full-screen drawable object to the rendering context.

9. Synchronizes to the refresh rate by setting the swap interval to `1`. (Recall that the `swap` variable was previously assigned a value of `1`). For more information, see "Synchronize with the Screen Refresh Rate" (page 125). The function `aglSetInteger` allows you to set a variety of rendering context parameters. For more information see "Techniques for Working with Rendering Contexts" (page 65).

Using CGL to Create a Full-Screen Context

Because the CGL API is at a lower level in the system architecture than either Cocoa or the AGL API, you can use it to create a full-screen context in either a Cocoa or a Carbon application. The code in Listing 3-3 shows how to capture the main display and create a full-screen context. As you can see, the code parallels the examples shown in "Using Cocoa to Create a Full-Screen Context" (page 38) and "Using AGL to Create a Full-Screen Context" (page 39). A detailed explanation for each numbered line of code appears following the listing.

Depending on what you want to accomplish, there are a number of modifications that you can make to the code, such as adjusting the display mode and synchronizing rendering to the screen refresh rate. See "Adjusting Display Modes" (page 42) and "Quartz Display Services and Full-Screen Mode" (page 145).

Listing 3-3 Setting up a full-screen context using CGL

```
CGDisplayCapture (kCGDirectMainDisplay);                          // 1
CGLPixelFormatAttribute attribs[] = { kCGLPFADoubleBuffer,
        kCGLPFAFullScreen,
        kCGLPFADisplayMask,
        CGDisplayIDToOpenGLDisplayMask(kCGDirectMainDisplay),
        NULL
```

```
    };                                                                  // 2
CGLPixelFormatObj pixelFormatObj;
long numPixelFormats ;

CGLChoosePixelFormat( attribs, &pixelFormatObj, &numPixelFormats );

CGLContextObj contextObj ;
CGLCreateContext( pixelFormatObj, NULL, &contextObj );

CGLDestroyPixelFormat( pixelFormatObj );

CGLSetCurrentContext( contextObj );                                     // 3
CGLSetFullScreen( contextObj );                                         // 4

//****** Perform your application's main loop

CGLSetCurrentContext(NULL);                                             // 5
CGLClearDrawable(contextObj);
CGLDestroyContext(contextObj);
CGReleaseAllDisplays();
```

Here's what the code does:

1. Captures the main display.
2. Sets up an array of attributes that includes the full-screen attribute and the display mask associated with the captured display.
3. Sets the current context to the one it will use for full-screen drawing.
4. Attaches a full-screen drawable object to the current context.
5. After all drawing is completed, sets the current context to `NULL`, and goes on to perform the other necessary clean up work: clearing the drawable object, destroying the rendering context, and releasing the displays.

Adjusting Display Modes

The Quartz Display Services API provides several functions that adjust the display mode:

- `CGDisplayBestModeForParameters` finds the display mode that is closest to a specified depth and screen size.
- `CGDisplayBestModeForParametersAndRefreshRate` finds the display mode that is closest to a specified depth and resolution, and that also uses a refresh rate equal to or near the specified rate.
- `CGDisplayBestModeForParametersAndRefreshRateWithProperty` finds the display mode that is closest to a specified depth, resolution, and refresh rate and that also has a specific property. Properties include whether the mode is safe for hardware, is interlaced, is stretched, or can provide output suitable for television.

If you want to adjust the display mode, you first need to capture the display, as shown in Listing 3-4. The Quartz Display Services function `CGDisplaySwitchToMode` switches to the display mode returned by the function `CGDisplayBestModeForParameters`, which in this case, is the best display mode for the main display with a bit depth of 32 bits per pixel and a screen resolution of 1024 by 768 pixels. The display mode that's returned is not always what you asked for. It's the closest mode for the given parameter. The last parameter passed to this function—`exactMatch`—specifies whether the returned display mode matches exactly. If you don't need this information, you can pass `NULL`. When your application quits, Quartz Display Services automatically restores the user's display settings.

Note: Calling `CGDisplaySwitchToMode` does not guarantee that the display mode switches successfully. Displays have physical limitations that can prevent them from operating in a particular mode.

Listing 3-4 Adjusting the display mode

```
CGDisplayCapture (kCGDirectMainDisplay ) ;
CGDisplaySwitchToMode (kCGDirectMainDisplay,
            CGDisplayBestModeForParameters (kCGDirectMainDisplay,
                                     32, 1024, 768, NULL) );
```

Listing 3-5 shows how to switch the main display to a pixel depth of 32 bits per pixel, a resolution of 640 x 480, and a refresh rate of 60 Hz. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-5 Switching the resolution of a display

```
CFDictionaryRef displayMode ;
CFNumberRef number ;
boolean_t exactMatch ;

CGDisplayCapture (kCGDirectMainDisplay);                                        // 1
displayMode =
    CGDisplayBestModeForParametersAndRefreshRate (kCGDirectMainDisplay,
                            32,640,480,60,&exactMatch);                         // 2
if (exactMatch){                                                                // 3
     CGDisplaySwitchToMode (kCGDirectMainDisplay, displayMode);
}
else {
    // Your code to take appropriate action
}
// Run the event loop.
CGReleaseAllDisplays();                                                         // 4
```

Here's what the code does:

1. Captures the main display.
2. Requests a display mode with a depth of 32 bits per pixel, a resolution 640 x 480, and a refresh rate 60 Hz. The function finds the best match for these parameters.
3. If there is an exact match, then switches to the display mode.
4. Before the application quits, releases all displays.

What's Next?

The Quartz Display Services API performs a number of other tasks that are useful when drawing OpenGL to the full screen. "Quartz Display Services and Full-Screen Mode" (page 145) describes many of them, including enumerating displays and display modes, accessing display properties, fading the display, and programmatically controlling the pointer. You may also want to read "Draw Only When Necessary" (page 124) to see how to use Quartz Display Services to synchronize drawing with the screen refresh.

See Also

OpenGL sample code projects (ADC Reference Library):

- *GLCarbonAGLFullScreen* is a full-featured OpenGL application that uses the AGL API to draw to the full screen.
- *GLCarbonCGLFullScreen* is a full-featured OpenGL application that uses the CGL API to draw to the full screen.
- *NSOpenGL Fullscreen* shows how to create and switch between windowed and full-screen OpenGL contexts.

Drawing Offscreen

OpenGL programs draw offscreen for many reasons. They may need to store intermediate rendering results as a scene is built or they may need to store data that is used repeatedly, such as a texture. Mac OS X provides several options for rendering offscreen:

- Offscreen drawable objects. The Apple-specific OpenGL APIs provide routines that support drawing to CPU memory and are supported only by the software renderer. These objects are available in Mac OS X v10.0 and later, but are not recommended for performance-critical applications. See "Setting Up an Offscreen Drawable Object" (page 45).
- Hidden Cocoa views or Carbon windows. Starting in Mac OS X v10.2, the `NSOpenGLContext` class and AGL API provide routines that use the GPU to draw to hidden windows or views and then draw that content to an onscreen rendering context. See "Using a Hidden View or Window" (page 47).
- Pixel buffer drawable objects. Each of the Apple-specific OpenGL APIs provides routines for drawing to offscreen memory that's located on the GPU. An application can render an image into a pixel buffer once and then use the buffer contents multiple times to texture a variety of surfaces without copying the image data. Pixel buffers are available starting in Mac OS X v10.3. See "Rendering to a Pixel Buffer" (page 48).
- Framebuffer objects. A recently added OpenGL extension, these objects allow you to draw to buffers other than the usual buffers provided by OpenGL or the Mac OS X windowing system. Because these objects are window-system agnostic, they are easier to set up and more efficient to use than pixel buffers. Framebuffer objects are available in Mac OS X v10.4.3 and later, but not all hardware supports their use. See "Rendering to a Framebuffer Object" (page 51).

Setting Up an Offscreen Drawable Object

Offscreen drawable objects reside in CPU memory and are supported only by the software renderer. If you must support versions of Mac OS X prior to 10.2, you may need to use offscreen drawable objects. Otherwise, you should consider one of the other options for drawing offscreen.

The general procedure for setting up an offscreen drawable object is similar to setting up other drawable objects:

1. Specify renderer and buffer attributes, making sure to specify the offscreen attribute.
2. Obtain a pixel format object.
3. Create a context and make it current.
4. Bind the context to an offscreen drawable object.

Each of the Apple-specific OpenGL APIs provides a routine for binding the context to an offscreen drawable object:

- The `setOffScreen:width:height:rowbytes:` method of the `NSOpenGLContext` class instructs the receiver to render into an offscreen buffer.
- The AGL function `aglSetOffScreen` attaches an AGL rendering context to an offscreen buffer.
- The CGL function `CGLSetOffScreen` attaches a CGL rendering context to an offscreen buffer.

After creating and drawing to an offscreen context, call the OpenGL function `glFinish` to ensure that all submitted OpenGL commands have finished rendering into the memory buffer before you access the data. You can read the buffer contents by calling the OpenGL function `glReadPixels`, or you can use the buffer contents as a texture by calling the appropriate copy and surface texture functions.

The code in Listing 4-1 shows how to use the CGL API to create an offscreen drawable object that has a resolution of 1024 x 768 pixels and a depth of 32 bits per pixel. A detailed explanation for each numbered line of code appears following the listing.

Listing 4-1 Using CGL to draw to an offscreen drawable object

```
CGLPixelFormatAttribute attribs[] =                                    // 1
{
    kCGLPFAOffScreen,
    kCGLPFAColorSize, 32,
    NULL
} ;
CGLPixelFormatObj pixelFormatObj;
long numPixelFormats;
CGLChoosePixelFormat (attribs, &pixelFormatObj, &numPixelFormats);     // 2

CGLContextObj contextObj;
CGLCreateContext (pixelFormatObj, NULL, &contextObj);                  // 3
CGLDestroyPixelFormat (pixelFormatObj);
CGLSetCurrentContext (contextObj);                                     // 4
void* memBuffer = (void *) malloc (1024 * 768 * 32 / 8);               // 5
CGLSetOffScreen (contextObj, 1024, 768, 1024 * 4, memBuffer);          // 6
//***** Perform offscreen drawing
CGLSetCurrentContext (NULL);
CGLClearDrawable (contextObj);
CGLDestroyContext (contextObj);
```

Here's what the code does:

1. Sets up an array of pixel format attributes—an offscreen drawable object and a color buffer with a size of 32 bytes. Note that the list must be terminated by `NULL`.
2. Creates a pixel format object that has the specified renderer and buffer attributes.
3. Creates a CGL context using the newly created pixel format object.
4. Sets the current context to the newly created offscreen CGL context.
5. Allocates memory for the offscreen drawable object.
6. Binds the CGL context to the newly allocated offscreen memory buffer. You need to specify the width and height of the offscreen buffer (in pixels), the number of bytes per row, and a pointer to the block of memory you want to render the context into. The number of bytes per row must be at least the width times the bytes per pixels.

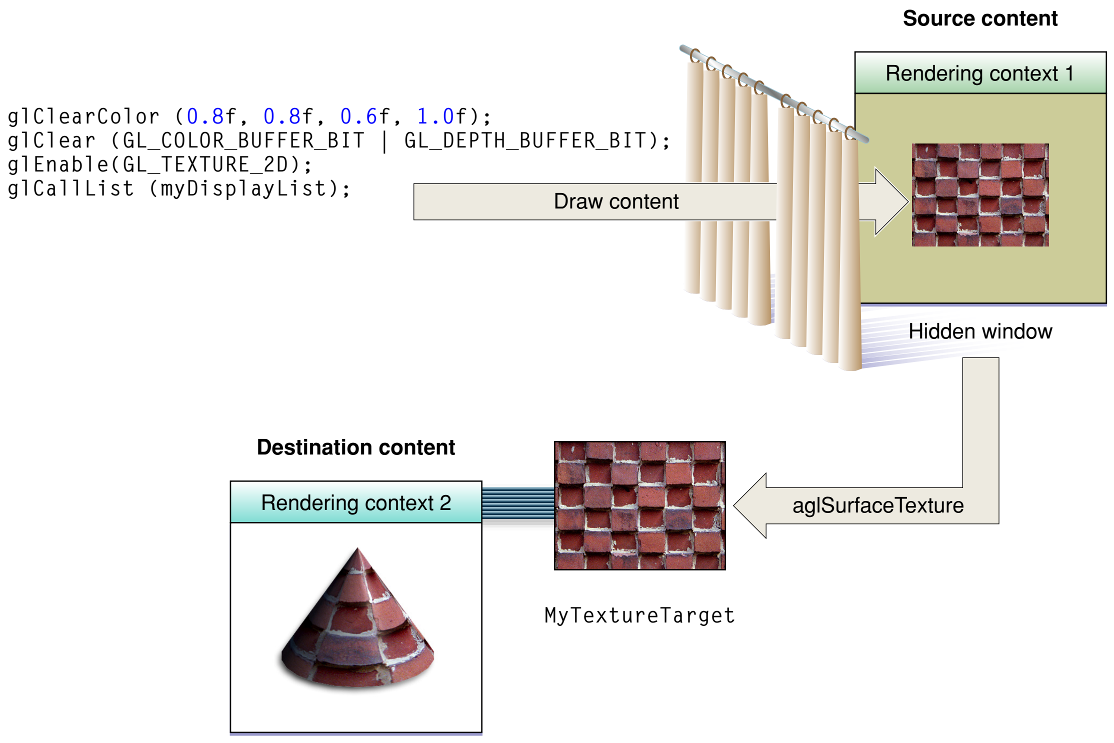
Using a Hidden View or Window

Starting in Mac OS X v10.2, you can use a hidden Carbon window or Cocoa view as a texture source. Cocoa supports this kind of offscreen drawing through the `createTexture:fromView:internalFormat:` method of the `NSOpenGLContext` class. Carbon provides the function `aglSurfaceTexture`.

Note: Although these routines provide a flexible way to render to an offscreen texture and then use that texture as a source, you should consider using pixel buffers and framebuffers instead. If your application provides support for Mac OS X v10.2, however, you must use hidden views and windows if you want accelerated offscreen drawing, because the other hardware-accelerated options are not available for Mac OS X v10.2

The crucial concept behind using hidden views and windows is that there are two rendering contexts involved, as shown in Figure 4-1: one that's bound to the hidden drawable object and the other that's bound to the destination window. You must make sure that the current rendering context is set to the appropriate context prior to drawing.

Figure 4-1 Using the content from a hidden window as a texture source



```
glClearColor (0.8f, 0.8f, 0.6f, 1.0f);
glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glEnable(GL_TEXTURE_2D);
glCallList (myDisplayList);
```

Drawing the texture content is just like drawing to an offscreen window. The content is treated as a texture only when you actually use it for the destination rendering context. You generate a texture name, bind it, and set the texture environment after you set the current context to the destination context. Once the texture is set, you call the routine that makes the texture content available to the destination window (either the `createTexture:fromView:internalFormat:` method of the `NSOpenGLContext` class or the AGL function `aglSurfaceTexture`). Then you can draw using the texture.

Figure 4-1 depicts the steps, explained below, that are required to use an offscreen window as a texture source:

1. Create a window to use as the texture source. The window should specify a hidden attribute.
2. Create a destination window to use the texture in.
3. Set up each window as a drawable object attached to an AGL context. That is, set up buffer and renderer attributes, get a pixel format object, create an AGL context, and attach the window to the context. For details, see "Drawing to a Window or View" (page 27). The pixel format object for each context must be compatible, but the contexts do not need to be shared.
4. Set the current rendering context to the texture source context and draw the texture. OpenGL draws the contents to the hidden window.
5. Set the current rendering context to the destination window.
6. Enable texturing by calling the function `glEnable`.
7. Generate a texture name and bind the name to a texture target, using code similar to the following:

   ```
   glGenTextures (1, &mySurfaceTexName);
   glBindTexture (GL_TEXTURE_2D, mySurfaceTexName);
   glTexParameterf (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
   ```

   This code sets up a power-of-two texture. You can just as easily use a rectangular texture by providing the `GL_TEXTURE_RECTANGLE_ARB` option.
8. Map the contents of the texture source window to the texture target bound in the destination window, using code similar to the following:

   ```
   aglSurfaceTexture (myDestinationContext,GL_TEXTURE_2D,
               GL_RGBA8, mySurfaceTextureContext);
   ```
9. Draw to the destination window, using the texture just as you would any other texture.
10. When you are done using the texture, unbind it by calling `glBindTexture` with the texture set to `0`.

    ```
    glBindTexture (GL_TEXTURE_2D, 0);
    ```
11. Flush the content to the destination window by calling the function `aglSwapBuffers`.

Rendering to a Pixel Buffer

The OpenGL extension string `GL_APPLE_pixel_buffer` provides hardware-accelerated offscreen rendering to a pixel buffer. A pixel buffer is typically used as a texture source. It can also be used for remote rendering.

When you are using a pixel buffer as a texture source, keep in mind that you must manage two rendering contexts. One is the rendering context attached to the pixel buffer. That's the context that you must draw to when you create the texture content. The other is the rendering context attached to the onscreen drawable object that will use the texture.

The first step in using a pixel buffer is to create it. The Apple-specific OpenGL APIs each provide a routine for this purpose:

- The `NSOpenGLPixelBuffer` method `initWithTextureTarget:textureInternalFormat:textureMaxMipMapLevel:pixelsWide:pixelsHigh:`
- The AGL function `aglCreatePBuffer`
- The CGL function `CGLCreatePBuffer`

Each of these routines requires that you provide a texture target, an internal format, a maximum mipmap level, and the width and height of the texture.

The texture target must be one of these OpenGL texture constants: `GL_TEXTURE_2D` for a 2D texture, `GL_TEXTURE_RECTANGLE_ARB` for a rectangular (not power-of-two) texture, or `GL_TEXTURE_CUBE_MAP` for a cube map texture.

The internal format specifies how to interpret the data for texturing operations. You can supply any of these options: `GL_RGB` (each pixel is a three-component group), `GL_RGBA` (each pixel is a four-component group), or `GL_DEPTH_COMPONENT` (each pixel is a single depth component).

The maximum mipmap level should be `0` for a pixel buffer that does not have a mipmap. The value that you supply should not exceed the actual maximum number of mipmap levels that can be represented with the given width and height.

Note that none of the routines that create a pixel buffer allocate the storage needed. The storage is allocated by the system at the time that you attach the pixel buffer to a rendering context.

Setting Up a Pixel Buffer for Offscreen Drawing

After you create a pixel buffer, the general procedure for using a pixel buffer for drawing is similar to the way you set up windows and views for drawing:

1. Specify renderer and buffer attributes.
2. Obtain a pixel format object.
3. Create a rendering context and make it current.
4. Attach a pixel buffer to the context using the appropriate Apple OpenGL attachment function:
    - The `setPixelBuffer:cubeMapFace:mipMapLevel:currentVirtualScreen:` method of the `NSOpenGLContext` class instructs the receiver to render into a pixel buffer.
    - The AGL function `aglSetPBuffer` attaches an AGL rendering context to a pixel buffer.
    - The CGL function `CGLSetPBuffer` attaches a CGL rendering context to a pixel buffer.
5. Draw, as you normally would, using OpenGL.

Using a Pixel Buffer as a Texture Source

Pixel buffers let you perform direct texturing without incurring the cost of extra copies. After drawing to a pixel buffer, you can create a texture by following these steps:

1. Generate a texture name by calling the OpenGL function `glGenTextures`.
2. Bind the named texture to a target by calling the OpenGL function `glBindTexture`.
3. Set the texture parameters by calling OpenGL function `glTexEnvParameter`.
4. Set up the pixel buffer as the source for the texture by calling one of the following Apple OpenGL functions:
   - The `setTextureImageToPixelBuffer:colorBuffer:` method of the `NSOpenGLContext` class attaches the image data in the pixel buffer to the texture object currently bound by the receiver.
   - The AGL function `aglTexImagePBuffer` binds the contents of an AGL pixel buffer as the data source for a texture object.
   - The CGL function `CGLTexImagePBuffer` binds the contents of a CGL pixel buffer as the data source for a texture object.

   The context that you attach to the pixel buffer is the target rendering context: the context that uses the pixel buffer as the source of the texture data. Each of these routines requires a `source` parameter, which is an OpenGL constant that specifies the source buffer to texture from. The source parameter must be a valid OpenGL buffer, such as `GL_FRONT`, `GL_BACK`, or `GL_AUX0`, and should be compatible with the buffer attributes used to create the OpenGL context associated with the pixel buffer. This means that the pixel buffer must possess the buffer in question for texturing to succeed. For example, if the buffer attribute used with the pixel buffer is only single buffered, then texturing from the `GL_BACK` buffer will fail.

   If you modify content of any pixel buffer that contains mipmap levels, you must call the appropriate Apple OpenGL function again (`setTextureImageToPixelBuffer:colorBuffer:`, `aglTexImagePBuffer`, or `CGLTexImagePBuffer`) before drawing with the pixel buffer to ensure that the content is synchronized with OpenGL. To synchronize the content of pixel buffers without mipmaps, simply rebind to the texture object using `glBind`.
5. Draw primitives using the appropriate texture coordinates. (See "The Redbook"—*OpenGL Programming Guide*—for details.)
6. Call `glFlush` to cause all drawing commands to be executed.
7. When you no longer need the texture object, call the OpenGL function `glDeleteTextures`.
8. Set the current context to `NULL` using one of the Apple OpenGL routines:
   - The `makeCurrentContext` method of the `NSOpenGLContext` class
   - The AGL function `aglSetCurrentContext`
   - The CGL function `CGLSetCurrentContext`
9. Destroy the pixel buffer by calling `CGLDestroyPBuffer`.
10. Destroy the context by calling `CGLDestroyContext`.

11. Destroy the pixel format by calling `CGLDestroyPixelFormat`.

You might find these guidelines useful when using pixel buffers for texturing:

- You cannot make OpenGL texturing calls that modify pixel buffer content (such as `glTexSubImage2D` or `glCopyTexImage2D`) with the pixel buffer as the destination. You can use texturing commands to read data from a pixel buffer, such as `glCopyTexImage2D`, with the pixel buffer texture as the source. You can also use OpenGL functions such as `glReadPixels` to read the contents of a pixel buffer directly from the pixel buffer context.
- Texturing can fail to produce the intended results without reporting an error. You must make sure that you enable the proper texture target, set a compatible filter mode, and adhere to other requirements described in the OpenGL specification.
- You are not required to set up context sharing when you texture from a pixel buffer. You can have different pixel format objects and rendering contexts for both the pixel buffer and the target drawable object, without sharing resources, and still texture using a pixel buffer in the target context.

Rendering to a Pixel Buffer on a Remote System

Follow these steps to render to a pixel buffer on a remote system. The remote system does not need to have a display attached to it.

1. When you set the renderer and buffer attributes, include the remote pixel buffer attribute `kCGLPFARemotePBuffer`.
2. Log in to the remote machine using the `ssh` command to ensure security.
3. Run the application on the target system.
4. Retrieve the content.

Rendering to a Framebuffer Object

The OpenGL framebuffer extension provides a mechanism for applications to render offscreen to a destination other than the usual OpenGL buffers or destinations provided by the windowing system. This destination is called a framebuffer object.

Note: Extensions are available on a per-renderer basis. Before you use the framebuffer extension you must check each renderer to make sure that it supports the extension.

A **framebuffer object** (FBO) contains state information for the OpenGL framebuffer and its set of images. A framebuffer object is similar to a drawable object, except that a drawable object is a window-system-specific object, whereas a framebuffer object is a window-agnostic object that's defined in the OpenGL standard, not by Apple. After drawing to a framebuffer object it is straightforward to display the content onscreen. A single command redirects all subsequent drawing back to the drawable object provided by the window system, where the FBO content can then be used as a texture.

Framebuffer objects offer a number of benefits over using pixel buffers.

- They are window-system independent, which makes porting code easier.
- They are easy to set up and save memory. There is no need to set up attributes and obtain a pixel format object.
- They use a single OpenGL context, whereas each pixel buffer must be bound to a context.
- You can switch between them faster since there is no context switch as with pixel buffers. What this means is that an application doesn't need to ensure that all rendering commands to the offscreen context are complete before using the results in the window context. Since there is only one context, commands are guaranteed to be serialized.
- They can share depth buffers; pixel buffers cannot.
- You can use them for 2D pixel images and for texture images.

Completeness is a key concept to understanding framebuffer objects. **Completeness** is a state that indicates whether a framebuffer object meets all the requirements for drawing. You test for this state after performing all the necessary setup work. If a framebuffer object is not complete, it cannot be used effectively as the destination for rendering operations and the source for read operations.

Completeness is dependent on many factors that are not possible to condense into one or two statements, but these factors are thoroughly defined in the OpenGL specification for the framebuffer object extension. The specification describes the requirements for internal formats of images attached to the framebuffer, how to determine if a format is color-, depth-, and stencil-renderable, as well as a number of other requirements.

Prior to using framebuffer objects, you should take a look at the OpenGL specification, which not only defines the framebuffer object API, but provides detailed definitions of all the terms necessary to understand their use and shows several code examples.

The remainder of this section provides an overview of how to use a framebuffer in the simplest case. You'll get an idea of how the setup of a framebuffer object compares to the other methods described in this chapter. To learn how powerful framebuffer objects are and to see examples of how to use them for a variety of purposes (such as for mipmaps) you'll want to read the OpenGL specification.

Similar to pixel buffers, framebuffer objects are suited for two types of drawing: textures and images. The functions used to set up textures and images are slightly different. The API for images uses the renderbuffer terminology defined in the OpenGL specification. A **renderbuffer image** is simply a 2D pixel image. The API for textures uses texture terminology, as you might expect. For example, one of the calls for setting up a framebuffer object for a texture is `glFramebufferTexture2DEXT`, whereas the call for setting up a framebuffer object for an image is `glFramebufferRenderbufferEXT`. You'll see how to set up a simple framebuffer object for each type of drawing, starting first with textures.

Drawing a Texture Offscreen

These are the basic steps needed to set up a framebuffer object for drawing a texture offscreen:

1. Make sure the framebuffer extension (`GL_EXT_framebuffer_object`) is supported on the system that your code runs on. See "Determining the OpenGL Capabilities Supported by the Hardware" (page 59).

2. Check the renderer limits. For example, you might want to call the OpenGL function `glGetIntegerv` to check the maximum texture size (`GL_MAX_TEXTURE_SIZE`) or find out the maximum number of color buffers (`GL_MAX_COLOR_ATTACHMENTS_EXT`).

3. Generate a framebuffer object name by calling the following function:

   ```
   void glGenFramebuffersEXT (GLsizei n, GLuint *ids);
   ```

   `n` is the number of framebuffer object names that you want to create.

   On return, `*ids` points to the generated names.

4. Bind the framebuffer object name to a framebuffer target by calling the following function:

   ```
   void glBindFramebufferEXT(GLenum target, GLuint framebuffer);
   ```

   `target` should be the constant `GL_FRAMEBUFFER_EXT`.

   `framebuffer` is set to an unused framebuffer object name.

   On return, the framebuffer object is initialized to the state values described in the OpenGL specification for the framebuffer object extension. Each attachment point of the framebuffer is initialized to the attachment point state values described in the specification. The number of attachment points is equal to `GL_MAX_COLOR_ATTACHMENTS_EXT` plus 2 (for depth and stencil attachment points).

5. Generate a texture name.

6. Bind the texture name to a texture target.

7. Set up the texture environment and parameters.

8. Define the texture by calling the appropriate OpenGL function to specify the target, level of detail, internal format, dimensions, border, pixel data format, and texture data storage.

9. Attach the texture to the framebuffer by calling the following function:

   ```
   void glFramebufferTexture2DEXT (GLenum target, GLenum attachment,
                                   GLenum textarget, GLuint texture,
                                   GLint level);
   ```

   `target` must be `GL_FRAMEBUFFER_EXT`.

   `attachment` must be one of the attachment points of the framebuffer: `GL_STENCIL_ATTACHMENT_EXT`, `GL_DEPTH_ATTACHMENT_EXT`, or `GL_COLOR_ATTACHMENTn_EXT`, where `n` is a number from `0` to `GL_MAX_COLOR_ATTACHMENTS_EXT-1`.

   `textarget` is the texture target.

   `texture` is an existing texture object.

   `level` is the mipmap level of the texture image to attach to the framebuffer.

10. Check to make sure that the framebuffer is complete by calling the following function:

    ```
    GLenum glCheckFramebufferStatusEXT(GLenum target);
    ```

    `target` must be the constant `GL_FRAMEBUFFER_EXT`.

    This function returns a status constant. You must test to make sure that the constant is `GL_FRAMEBUFFER_COMPLETE_EXT`. If it isn't, see the OpenGL specification for the framebuffer object extension for a description of the other constants in the status enumeration.

11. Render content to the texture. You must make sure to bind a different texture to the framebuffer object or disable texturing before you render content. That is, if you render to a framebuffer object texture attachment with that same texture currently bound and enabled, the result is undefined.
12. To view the contents of the texture, make the window the target of all rendering commands by calling the function `glBindFramebufferEXT` and passing the constant `GL_FRAMEBUFFER_EXT` and `0`. The window is always specified as `0`.
13. Use the texture attachment as a normal texture by binding it, enabling texturing, and drawing.
14. Delete the texture.
15. Delete the framebuffer object by calling the following function:

    ```
    void  glDeleteFramebuffersEXT (GLsizei n, const GLuint *framebuffers);
    ```

    `n` is the number of framebuffer objects to delete.

    `*framebuffers` points to an array that contains the framebuffer object names.

Listing 4-2 shows code that performs these tasks. This example sets up and draws to a single framebuffer object. Your application can set up more than one framebuffer object if it requires them.

Listing 4-2 Setting up a framebuffer for texturing

```
GLuint framebuffer, texture;
GLenum status;
glGenFramebuffersEXT(1, &framebuffer);
// Set up the FBO with one texture attachment
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, framebuffer);
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, TEXWIDE, TEXHIGH, 0,
                GL_RGBA, GL_UNSIGNED_BYTE, NULL);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
                GL_TEXTURE_2D, texture, 0);
status = glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT);
if (status != GL_FRAMEBUFFER_COMPLETE_EXT)
                // Handle error here
// Your code to draw content to the FBO
// ...
// Make the window the target
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
//Your code to use the contents of the FBO
// ...
//Tear down the FBO and texture attachment
glDeleteTextures(1, &texture);
glDeleteFramebuffersEXT(1, &framebuffer);
```

Drawing a Renderbuffer Image Offscreen

There is a lot of similarity between setting up a framebuffer object for drawing images and setting one up to draw textures. These are the basic steps needed to set up a framebuffer object for drawing a 2D pixel image (a renderbuffer image) offscreen:

1. Make sure the framebuffer extension (`EXT_framebuffer_object`) is supported on the renderer that your code runs on. See "Determining the OpenGL Capabilities Supported by the Hardware" (page 59).

2. Check the renderer limits. For example, you might want to call the OpenGL function `glGetIntegerv` to find out the maximum number of color buffers (`GL_MAX_COLOR_ATTACHMENTS_EXT`).

3. Generate a framebuffer object name by calling the function `glGenFramebuffersEXT`. (See Step 3 in "Drawing a Texture Offscreen" (page 52).)

4. Bind the framebuffer object name to a framebuffer target by calling the function `glBindFramebufferEXT`. (See Step 4 in "Drawing a Texture Offscreen" (page 52).)

5. Generate a renderbuffer object name by calling the following function:

   ```
   void  glGenRenderbuffersEXT (GLsizei n, GLuint *renderbuffers );
   ```

   `n` is the number of renderbuffer object names to create.

   `*renderbuffers` points to storage for the generated names.

6. Bind the renderbuffer object name to a renderbuffer target by calling the following function:

   ```
   void glBindRenderbufferEXT (GLenum target, GLuint renderbuffer);
   ```

   `target` must be the constant `GL_RENDERBUFFER_EXT`

   `renderbuffer` is the renderbuffer object name generated previously.

7. Create data storage and establish the pixel format and dimensions of the renderbuffer image by calling the following function:

   ```
   void glRenderbufferStorageEXT (GLenum target, GLenum internalformat,
                                  GLsizei width, GLsizei height);
   ```

   `target` must be the constant `GL_RENDERBUFFER_EXT`.

   `internalformat` is the pixel format of the image. The value must be `RGB`, `RGBA`, `DEPTH_COMPONENT`, `STENCIL_INDEX`, or one of the other formats listed in the OpenGL specification.

   `width` is the width of the image, in pixels.

   `height` is the height of the image, in pixels.

8. Attach the renderbuffer to a framebuffer target by calling the function `glFramebufferRenderbufferEXT`.

   ```
   void glFramebufferRenderbufferEXT(GLenum target, GLenum attachment,
                                 GLenum renderbuffertarget, GLuint renderbuffer);
   ```

   `target` must be the constant `GL_FRAMEBUFFER_EXT`.

`attachment` should be one of the attachment points of the framebuffer: `GL_STENCIL_ATTACHMENT_EXT`, `GL_DEPTH_ATTACHMENT_EXT`, or `GL_COLOR_ATTACHMENTn_EXT`, where n is a number from `0` to `GL_MAX_COLOR_ATTACHMENTS_EXT-1`.

`renderbuffertarget` must be the constant `GL_RENDERBUFFER_EXT`.

`renderbuffer` should be set to the name of the renderbuffer object that you want to attach to the framebuffer.

9. Check to make sure that the framebuffer is complete by calling the following function:

   ```
   enum  glCheckFramebufferStatusEXT(GLenum target);
   ```

   `target` must be the constant `GL_FRAMEBUFFER_EXT`.

   This function returns a status constant. You must test to make sure that the constant is `GL_FRAMEBUFFER_COMPLETE_EXT`. If it isn't, see the OpenGL specification for the framebuffer object extension for a description of the other constants in the status enumeration.

10. Draw the 2D pixel image to the renderbuffer.

11. To view the contents of the renderbuffer, make the window the target of all rendering commands by calling the function `glBindFramebufferEXT` and passing the constant `GL_FRAMEBUFFER_EXT` and `0`. The window is always specified as `0`.

12. To access the contents of the renderbuffer object, use OpenGL functions such as `glReadPixels` or `glCopyTexImage2D`.

13. Delete the framebuffer object with its renderbuffer attachment.

Listing 4-3 shows code that sets up and draws to a single renderbuffer object. Your application can set up more than one renderbuffer object if it requires them.

Listing 4-3 Setting up a renderbuffer for drawing images

```
GLuint framebuffer, renderbuffer;
GLenum status;
// Set the width and height appropriately for you image
GLuint texWidth = 1024,
       texHeight = 1024;
//Set up a FBO with one renderbuffer attachment
glGenFramebuffersEXT(1, &framebuffer);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, framebuffer);
glGenRenderbuffersEXT(1, &renderbuffer);
glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, renderbuffer);
glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT, GL_RGBA8, texWidth, texHeight);
glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
                 GL_RENDERBUFFER_EXT, renderbuffer);
status = glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT);
if (status != GL_FRAMEBUFFER_COMPLETE_EXT)
                // Handle errors
//Your code to draw content to the renderbuffer
// ...
// Make the window the target
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
//Your code to use the contents
```

```
// ...
// Delete the renderbuffer attachment
glDeleteRenderbuffersEXT(1, &renderbuffer);
```

See Also

This chapter provided an overview of the various ways to perform offscreen OpenGL drawing. It's only a starting point, especially for those who want to use textures and who are concerned with performance. You'll also want to read the following:

- "Techniques for Working with Texture Data" (page 95)
- "Improving Performance" (page 121)
- OpenGL specification for the framebuffer object extension describes the framebuffer API in detail and provides sample code for setting up and using framebuffer objects and renderbuffers.

OpenGL sample code projects (ADC Reference Library):

- *AGLSurfaceTexture* shows how to render to a texture.
- *GLCarbon1ContextPbuffer* renders surfaces into a pixel buffer and uses it as a texture to draw on a cube; uses a single context.
- *GLCarbonSharedPbuffer* shares a single pixel buffer with multiple other contexts.
- *Quartz Composer Offline Rendering* shows how to render a Quartz Composer composition as a series of images using an OpenGL pixel buffer.

Determining the OpenGL Capabilities Supported by the Hardware

One of the benefits of using OpenGL is that it is extensible. An extension is typically introduced by one or more vendors and then later is accepted by the OpenGL Architecture Review Board (ARB). Some extensions are promoted from a vendor-specific extension to a common one while others become part of the core OpenGL API. Extensions allow OpenGL to embrace innovation, but they also have implications for how you verify that the OpenGL functionality you want to use is available.

Because extensions can be introduced at the vendor level, more than one extension can provide the same basic functionality. There might also be an ARB extension that has functionality similar to that of a vendor-specific extension. As particular functionality becomes widely adopted, it can be moved into the core OpenGL API by the ARB. As a result, functionality that you want to use could be included as an extension, as part of the core API, or both. For example, the ability to combine texture environments is supported through the `GL_ARB_texture_env_combine` and the `GL_EXT_texture_env_combine` extensions. It's also part of the core OpenGL version 1.3 API. Although each has similar functionality, they use a different syntax. What this means is that you may need to check in several places (core OpenGL API and extension strings) to determine whether a specific renderer supports functionality that you want to use.

Detecting Functionality

OpenGL has two types of commands—those that are part of the core API and those that are part of an extension to OpenGL. Your application first needs to check for the version of the core OpenGL API and then check for the available extensions. Keep in mind that OpenGL functionality is available on a per-renderer basis. Not all renderers support all the available functionality. For example, a software renderer might not support fog effects even though fog effects are available in an OpenGL extension installed on the current system. For this reason, it's important that you check for particular functionality on a per-renderer basis.

Regardless of which extension you are checking for, the approach is the same. You need to call the OpenGL function `glGetString` twice. The first time pass the `GL_VERSION` constant. The function returns a string that specifies the version of OpenGL. The second time, pass the `GL_EXTENSIONS` constant. The function returns a pointer to an extension name string. The **extension name string** is a space-delimited list of the OpenGL extensions that are supported by the current renderer. This string can be rather long, so make sure that you don't allocate a fixed-length string for the return value of the `glGetString` function. That is, do not use the function `strcpy`; use a pointer and evaluate the string in place.

Pass the extension name string to the function `gluCheckExtension` along with the name of the extension you want to check for. The `gluCheckExtension` function returns a Boolean value that indicates whether or not the extension is available for the current renderer.

If an extension becomes part of the core OpenGL API, OpenGL continues to export the name strings of the promoted extensions. It also continues to support the previous versions of any extension that has been exported in earlier versions of Mac OS X. The fact that extensions are not typically removed guarantees that the methodology you use today to check for a feature will work in all future versions of Mac OS X.

OpenGL has a tremendous amount of functionality, as you can see by looking at the extensions listed in "OpenGL Functionality by Version" (page 135). You need to call `gluCheckExtension` for each extension you want to check, and you need to check each extension for each renderer. Checking for functionality, although fairly straightforward, involves writing a large chunk of code. The best way to check for OpenGL functionality is to implement a capability-checking function that you call when your program starts up, and then any time a display configuration changes. Listing 5-1 shows a code excerpt that checks for a few extensions. (Note that it is not a standalone function.) A detailed explanation for each line of code appears following the listing.

You can extend this example to make a comprehensive functionality-checking routine for your application. For more details, see the `GLCheck.c` file in the *Cocoa OpenGL* sample application.

Listing 5-1 Checking for OpenGL functionality

```
GLint maxRectTextureSize;
GLint myMaxTextureUnits;
GLint myMaxTextureSize;
const GLubyte * strVersion;
const GLubyte * strExt;
float myGLVersion;
GLboolean isVAO, isTexLOD, isColorTable, isFence, isShade,
          isTextureRectangle;
strVersion = glGetString (GL_VERSION);                                  // 1
sscanf((char *)strVersion, "%f", &myGLVersion);
strExt = glGetString (GL_EXTENSIONS);                                   // 2
glGetIntegerv(GL_MAX_TEXTURE_UNITS, &myMaxTextureUnits);                // 3
glGetIntegerv(GL_MAX_TEXTURE_SIZE, &myMaxTextureSize);                  // 4
isVAO =
    gluCheckExtension ((const GLubyte*)"GL_APPLE_vertex_array_object",strExt); // 5

isFence = gluCheckExtension ((const GLubyte*)"GL_APPLE_fence", strExt);  // 6
isShade =
     gluCheckExtension ((const GLubyte*)"GL_ARB_shading_language_100", strExt);// 7

isColorTable =
     gluCheckExtension ((const GLubyte*)"GL_SGI_color_table", strExt) ||
             gluCheckExtension ((const GLubyte*)"GL_ARB_imaging", strExt);   // 8
isTexLOD =
     gluCheckExtension ((const GLubyte*)"GL_SGIS_texture_lod", strExt) ||
                                 (myGLVersion >= 1.2);                  // 9
isTextureRectangle = gluCheckExtension ((const GLubyte*)
                                "GL_EXT_texture_rectangle", strExt);
if (isTextureRectangle)
      glGetIntegerv (GL_MAX_RECTANGLE_TEXTURE_SIZE_EXT, &maxRectTextureSize);
else
     maxRectTextureSize = 0;                                            // 10
```

Here what the code does:

1. Gets a string that specifies the version of OpenGL.

2. Gets the extension name string.

3. Calls the OpenGL function `glGetIntegerv` to get the value of the attribute passed to it which, in this case, is the maximum number of texture units.

4. Gets the maximum texture size.

5. Checks whether vertex array objects are supported.

6. Checks for the Apple fence extension.

7. Checks for support for version 1.0 of the OpenGL shading language.

8. Checks for RGBA-format color lookup table support. In this case, the code needs to check for the vendor-specific string and for the ARB string. If either is present, the functionality is supported.

9. Checks for an extension related to the texture level of detail parameter (LOD). In this case, the code needs to check for the vendor-specific string and for the OpenGL version. If either the vendor string is present or the OpenGL version is greater than or equal to 1.2, the functionality is supported.

10. Gets the OpenGL limit for rectangle textures. For some extensions, such as the rectangle texture extension, it may not be enough to check whether the functionality is supported. You may also need to check the limits. You can use `glGetIntegerv` and related functions (`glGetBooleanv`, `glGetDoublev`, `glGetFloatv`) to obtain a variety of parameter values.

Keep in mind that you must check functionality on a per-renderer basis. The code in Listing 5-2 shows one way to query the current renderer. It uses the CGL API, which can be called from Cocoa or Carbon applications. In reality, you need to iterate over all displays and all renderers for each display to get a true picture of the OpenGL functionality available on a particular system. You also need to update the your functionality "snapshot" each time the list of displays or display configuration changes.

Listing 5-2 Setting up a valid rendering context to get renderer functionality information

```
#include <OpenGL/OpenGL.h>
#include <ApplicationServices/ApplicationServices.h>
CGDirectDisplayID display = CGMainDisplayID ();                    // 1
CGOpenGLDisplayMask myDisplayMask =
                CGDisplayIDToOpenGLDisplayMask (display);          // 2

{ // Check capabilities of display represented by display mask
    CGLPixelFormatAttribute attribs[] = {kCGLPFADisplayMask,
                              myDisplayMask,
                                NULL};                             // 3
    CGLPixelFormatObj pixelFormat = NULL;
    long numPixelFormats = 0;
    CGLContextObj myCGLContext = 0;
    CGLContextObj curr_ctx = CGLGetCurrentContext ();              // 4
    CGLChoosePixelFormat (attribs, &pixelFormat, &numPixelFormats); // 5
    if (pixelFormat) {
        CGLCreateContext (pixelFormat, NULL, &myCGLContext);       // 6
        CGLDestroyPixelFormat (pixelFormat);                       // 7
        CGLSetCurrentContext (myCGLContext);                       // 8
        if (myCGLContext) {
            // Check for capabilities and functionality here
        }
    }
    CGLDestroyContext (myCGLContext);                              // 9
    CGLSetCurrentContext (curr_ctx);                               // 10
}
```

Here's what the code does:

1. Gets the display ID of the main display.
2. Maps a display ID to an OpenGL mask.
3. Fills a pixel format attributes array with the display mask attribute and the mask value.
4. Saves the current context so that it can be restored later.
5. Gets the pixel format object for the display. The `numPixelFormats` parameter specifies how many pixel formats are listed in the pixel format object.
6. Creates a context based on the first pixel format in the list supplied by the pixel format object. Only one renderer will be associated with this context.

   In your application, you would need to iterate through all pixel formats for this display.
7. Destroys the pixel format object when it is no longer needed.
8. Sets the current context to the newly created, single-renderer context. Now you are ready to check for the functionality supported by the current renderer. See Listing 5-1 (page 60) for an example of functionality checking code.
9. Destroys the context because it is no longer needed.
10. Restores the previously saved context as the current context, thus ensuring no intrusion upon the user.

Guidelines for Code That Checks for Functionality

The guidelines in this section will ensure that your functionality checking code is thorough yet efficient. See "Detecting Functionality" (page 59) for specific details on implementing these guidelines.

- Don't rely on what's in a header file. A command declaration in a header file does not ensure that a feature is supported by the current renderer. Neither does linking against a stub library that exports a function.
- Make sure that a renderer is attached to a valid rendering context before you check the functionality of that renderer.
- Check the API version or the extension name string for the current renderer before you issue OpenGL commands.
- Check only once per renderer. After you've determined that the current renderer supports an OpenGL command, you don't need to check for that functionality again for that renderer.
- Ensure that your code supports a feature, whether the feature is part of the core OpenGL API or is an extension. Keep in mind that different constants and command names are often used for functionality that is both part of the core API and an extension.
- Enable only those OpenGL features that are tested. Enabling untested features can lead to application failures.

See Also

OpenGL extension information:

- The OpenGL Extensions Registry at http://www.opengl.org/registry/.
- *OpenGL Extensions Guide* provides a list of extensions and availability according to OpenGL version, Mac OS X version, and renderer.

Many OpenGL sample code projects (ADC Reference Library) contain code to check for OpenGL functionality. For example, see the `glCheck.c` and `glCheck.h` files in the *Cocoa OpenGL* sample application or in the *GLCarbonCGLFullScreen* sample application.

Techniques for Working with Rendering Contexts

A rendering context is a container for state information. When you designate a rendering context as the current rendering context, subsequent OpenGL commands modify the drawable object associated with that context. The actual drawing surfaces are never really owned by the rendering context but are created, as needed, only when the rendering context is actually attached to a drawable object. You can attach multiple rendering contexts to a set of drawing surfaces. Each context draws with its own unique "pen" represented by its current state.

"Drawing to a Window or View" (page 27), "Drawing to the Full Screen" (page 37), and "Drawing Offscreen" (page 45) show how to create a rendering context and attach it to a drawable object. As you'll recall, each of the Apple-specific OpenGL APIs provides a routine that's fairly easy to use for creating a rendering context. This chapter goes beyond creating rendering contexts; it shows how to set context parameters, update rendering contexts, and set up a shared context.

Context Parameters

A rendering context has a variety of parameters that you can set to suit the needs of your OpenGL drawing. Some of the most useful, and often overlooked, context parameters are discussed in this section: swap interval, surface opacity, surface drawing order, and back-buffer size control.

Each of the Apple-specific OpenGL APIs provides a routine for setting and getting rendering context parameters:

- The `setValues:forParameter:` method of the `NSOpenGLContext` class takes as arguments a list of values and a list of parameters.
- The `aglSetInteger` function takes as parameters a rendering context, a constant that specifies an option, and a value for that option.
- The `CGLSetParameter` function takes as parameters a rendering context, a constant that specifies an option, and a value for that option.

Some parameters need to be enabled for their values to take effect. The reference documentation for a parameter indicates whether a parameter needs to be enabled. See *NSOpenGLContext Class Reference*, *AGL Reference*, and *CGL Reference*.

Swap Interval

The **swap interval** parameter synchronizes the vertical retrace. If the swap interval is set to `0` (the default), buffers are swapped as soon as possible, without regard to the vertical refresh rate of the monitor. If the swap interval is set to any other value, the buffers are swapped only during the vertical retrace of the monitor. For more information, see "Draw Only When Necessary" (page 124).

You can use the following constants to specify that you are setting the swap interval value:

- For Cocoa, use `NSOpenGLCPSwapInterval`.
- For Carbon, use `AGL_SWAP_INTERVAL`.
- If you are using the CGL API, use `kCGLCPSwapInterval`. See Listing 6-1.

Listing 6-1 Using CGL to set up synchronization

```
long sync = 1;
// ctx must be a valid context
CGLSetParameter (ctx, kCGLCPSwapInterval, &sync);
```

Surface Opacity

OpenGL surfaces are typically rendered as opaque. Thus the background color for pixels with alpha values of `0.0` is the surface background color. If you set the value of the **surface opacity** parameter to `0`, then the contents of the surface are blended with the contents of surfaces behind the OpenGL surface. This operation is equivalent to OpenGL blending with a source contribution proportional to the source alpha and a background contribution proportional to `1` minus the source alpha. A value of `1` means the surface is opaque (the default); `0` means completely transparent.

You can use the following constants to specify that you are setting the surface opacity value:

- For Cocoa, use `NSOpenGLCPSurfaceOpacity`.
- For Carbon, use `AGL_SURFACE_OPACITY`.
- If you are using the CGL API, use `kCGLCPSurfaceOpacity`. See Listing 6-2.

Listing 6-2 Using CGL to set surface opacity

```
long opaque = 0;
// ctx must be a valid context
CGLSetParameter (ctx, kCGLCPSurfaceOpacity, &opaque);
```

Surface Drawing Order

The **surface drawing order** parameter specifies the position of the OpenGL surface relative to the window. A value of `1` means that the position is above the window; a value of `-1` specifies a position that is below the window. When you have overlapping views, setting the order to `-1` causes OpenGL to draw underneath, `1` causes OpGL to draw on top. This parameter is useful for drawing user interface controls on top of an OpenGL view.

You can use the following constants to specify that you are setting the surface drawing order value:

- For Cocoa, use `NSOpenGLCPSurfaceOrder`.
- For Carbon, use `AGL_SURFACE_ORDER`.
- If you are using the CGL API, use `kCGLCPSurfaceOrder`. See Listing 6-3.

Listing 6-3 Using CGL to set surface drawing order

```
long order = -1; // below window
// ctx must be a valid context
CGLSetParameter (ctx, kCGLCPSurfaceOrder, &order);
```

Vertex and Fragment Processing

CGL provides two parameters for checking whether the system is using the GPU for processing: `kCGLCPGPUVertexProcessing` and `kCGLCPGPUFragmentProcessing`. To check vertex processing, pass the vertex constant to the `CGLGetParameter` function. To check fragment processing, pass the fragment constant to `CGLGetParameter`.

Important: Although you can perform these queries at any time, keep in mind that such queries force an internal state validation, which can impact performance. For best performance, do not use these queries inside your drawing loop. Instead, perform the queries once at initialization or context setup time to determine whether OpenGL is using the CPU or the GPU for processing, and then act appropriately in your drawing loop.

Listing 6-4 Using CGL to check whether the GPU is processing vertices and fragments

```
BOOL gpuProcessing;
GLint fragmentGPUProcessing, vertexGPUProcessing;
CGLGetParameter (CGLGetCurrentContext(), kCGLCPGPUFragmentProcessing,
                                         &fragmentGPUProcessing);
CGLGetParameter(CGLGetCurrentContext(), kCGLCPGPUVertexProcessing,
                                         &vertexGPUProcessing);
gpuProcessing = (fragmentGPUProcessing && vertexGPUProcessing) ? YES : NO;
```

Back Buffer Size Control

Normally, the back buffer is the same size as the window or view that it's drawn into, and it changes size when the window or view changes size. For a window whose size is 720 by 480 pixels, the OpenGL back buffer is sized to match. If the window grows to 1024 by 768 pixels, for example, then the back buffer tracks this growth. If you do not want this behavior, use the **back buffer size control** parameter.

Using this parameter fixes the size of the back buffer and lets the system scale the image automatically when it moves the data to a variable size buffer (see Figure 6-1). The size of the back buffer remains fixed at the size that you set up regardless of whether the image is resized to display larger onscreen.

You can use the following constants to specify that you are setting the surface drawing order value:

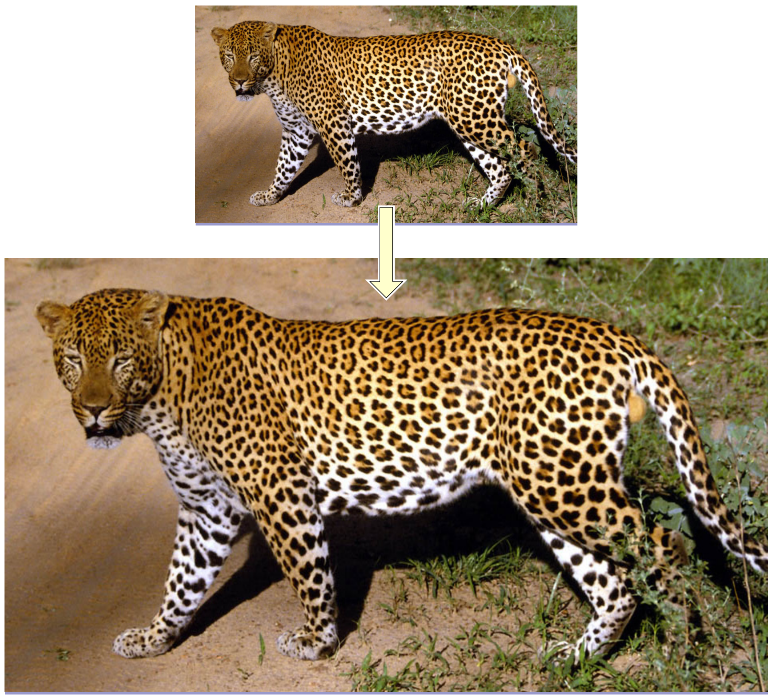
- If you are using the CGL API, use `kCGLCPSurfaceBackingSize`, as shown in Listing 6-5.
- For Carbon, use `AGL_SURFACE_BACKING_SIZE`.

Listing 6-5 Using CGL to set up back buffer size control

```
long dim[2] = {720, 480};
// ctx must be a valid context
CGLSetParameter(ctx, kCGLCPSurfaceBackingSize, dim);
```

```
CGLEnable (ctx, kCGLCESurfaceBackingSize);
```

Figure 6-1 A fixed size back buffer and variable size front buffer



Updating a Rendering Context

A rendering context must be updated whenever the renderer or geometry changes. A renderer change can occur when the user drags a window from one display to another. Geometry changes occur when the display mode changes or when a a window is resized or moved. If the system does not update the context automatically, then your application must perform the update. You need to track the appropriate events and call the update function provided by the Apple-specific OpenGL API that you're using.

Updating a rendering context is not the same as flushing graphics buffers. An update notifies the rendering context of geometry changes; it doesn't flush content. Calling an update function allows the OpenGL engine to ensure that the surface size is set and that the renderer is properly updated for any virtual screen changes. If you don't update the rendering context you either do not see any OpenGL rendering or you see rendering artifacts.

The routine that you call for updating determines how events related to renderer and geometry changes are handled. For applications that subclass `NSOpenGLView`, Cocoa calls the `update` method automatically. Applications that use the `NSOpenGLContext` class without subclassing its view must call the `update` method of `NSOpenGLContext` directly. For a full-screen Cocoa application, calling the `setFullScreen` method of `NSOpenGLContext` ensures that depth, size, or display changes take affect.

Carbon applications drawing OpenGL content to a window should call the function `aglUpdateContext`. For full-screen CGL and AGL applications, you need to call `CGLSetFullScreen` and `aglSetFullScreen` respectively to ensure that depth, size, or display changes take affect rather than calling an update function.

Your application must update the rendering context after the system event but before drawing to the context. If the drawable object is resized, you may want to issue a `glViewport` command to ensure that the content scales properly.

Note: Some system-level events (such as display mode changes) that require a context update could reallocate the buffers of the context, thus you need to redraw the entire scene after all context updates.

It's important that you don't update rendering contexts more than necessary. Your application should respond to system-level events and notifications rather than updating every frame. For example, you'll want to respond to window move and resize operations and to display configuration changes such as a color depth change.

The sections that follow describe in more detail how to use Cocoa, AGL, and CGL to update a rendering context, but you'll want to read "Tracking Renderer Changes" before going on to the sections specific to the three APIs.

Tracking Renderer Changes

It's fairly straightforward to track geometry changes, but how are renderer changes tracked? This is where the concept of a virtual screen becomes important (see "Virtual Screens" (page 21)). A change in the virtual screen indicates a renderer change, a change in renderer capability, or both. When your application detects a window resize event, window move event, or display change, it can then check for a virtual screen change and respond to the change appropriately. This ensures that the current application state reflects any changes in renderer capabilities.

Each of the Apple-specific OpenGL APIs has a function that returns the current virtual screen number:

- The `currentVirtualScreen` method of the `NSOpenGLContext` class
- The `aglGetVirtualScreen` function
- The `CGLGetVirtualScreen` function

The virtual screen number represents an index in the list of virtual screens that were set up specifically for the pixel format object used for the rendering context. The number is unique to the list but is meaningless otherwise.

Updating a Rendering Context for a Custom Cocoa View

If you subclass `NSView` instead of using the `NSOpenGLView` class, your application must update the rendering context. That's due to a slight difference between the events normally handled by the `NSView` class and those handled by the `NSOpenGLView` class. Cocoa does not call a `reshape` method for the `NSView` class when the size changes because that class does not export a `reshape` method to override. Instead, you need to perform reshape operations directly in your `drawRect:` method, looking for changes in view bounds prior to actually drawing content. Using this approach provides results that are equivalent to using the `reshape` method of the `NSOpenGLView` class.

Listing 6-6 is a partial implementation of a custom view that shows how to handle context updates. The `update` method is called after move, resize, and display change events and when the surface needs updating. The class adds an observer to the notification `NSViewGlobalFrameDidChangeNotification`, giving a callback upon which to base context updates. This notification is posted whenever an `NSView` object that has attached surfaces (that is, `NSOpenGLContext` objects) resizes, moves, or changes coordinate offsets.

It's slightly more complicated to handle changes in the display configuration. For that, you need to register for the notification `NSApplicationDidChangeScreenParametersNotification` through the `NSApplication` class. This notification is posted whenever the configuration of any of the displays attached to the computer is changed (either programmatically or when the user changes the settings in the interface).

Listing 6-6 Handling context updates for a custom view

```
#import <Cocoa/Cocoa.h>
#import <OpenGL/OpenGL.h>
#import <OpenGL/gl.h>

@class NSOpenGLContext, NSOpenGLPixelFormat;

@interface CustomOpenGLView : NSView
{
  @private
  NSOpenGLContext*   _openGLContext;
  NSOpenGLPixelFormat* _pixelFormat;
}

- (id)initWithFrame:(NSRect)frameRect
        pixelFormat:(NSOpenGLPixelFormat*)format;

- (void)update;
@end

@implementation CustomOpenGLView

- (id)initWithFrame:(NSRect)frameRect
        pixelFormat:(NSOpenGLPixelFormat*)format
{
  self = [super initWithFrame:frameRect];
  if (self != nil) {
    _pixelFormat   = [format retain];
  [[NSNotificationCenter defaultCenter] addObserver:self
                   selector:@selector(_surfaceNeedsUpdate:)
                   name:NSViewGlobalFrameDidChangeNotification
                   object:self];
  }
  return self;
}

- (void)dealloc
  [[NSNotificationCenter defaultCenter] removeObserver:self
                     name:NSViewGlobalFrameDidChangeNotification
                     object:self];
  [self clearGLContext];
  [_pixelFormat release];
  [super dealloc];
}
```

```
- (void)update
{
  if ([_openGLContext view] == self) {
    [_openGLContext update];
  }
}

- (void) _surfaceNeedsUpdate:(NSNotification*)notification
{
  [self update];
}

@end
```

Updating a Rendering Context for a Carbon Window

The simplest way to handle resize and move events is by using Carbon events. To cover window resize and move operations, your application can handle the events `kEventWindowBoundsChanged` and `kEventWindowZoomed`. The system generates the event `kEventWindowBoundsChanged` for window resize and window drag operations, which takes care of most cases other than a window zoom operation. For that, track the event `kEventWindowZoomed`. For more information on these and other Carbon events see *Carbon Event Manager Programming Guide* and *Carbon Event Manager Reference*.

Listing 6-7 demonstrates a simple window event handler. Note that the supporting routines needed by the window event handler—`MyHandleWindowUpdate`, `MyDisposeGL`, and `MyBuildGL`—are not shown in the listing. These are routines that you need to write. A detailed explanation for each numbered line of code appears following the listing.

Listing 6-7 Handling Carbon events associated with an AGL context

```
#include <Carbon/Carbon.h>

static pascal OSStatus windowEvtHndlr (EventHandlerCallRef myHandler,
                                       EventRef event,
                                       void* userData)
{
  WindowRef     window;
  AGLContext    aglContext = (AGLContext) userData;                   // 1
  Rect          rectPort = {0,0,0,0};
  OSStatus      result = eventNotHandledErr;
  UInt32        class = GetEventClass (event);
  UInt32        kind = GetEventKind (event);

  GetEventParameter(event, kEventParamDirectObject, typeWindowRef,
                    NULL, sizeof(WindowRef), NULL, &window);
  if (window) {
    GetWindowPortBounds (window, &rectPort);
  }
  switch (class) {
    case kEventClassWindow:
      switch (kind) {
        case kEventWindowActivated:                                   // 2
        case kEventWindowDrawContent:                                 // 3
          MyHandleWindowUpdate(window);
          break;
        case kEventWindowClose:                                                // 4
          HideWindow (window);
          MyDisposeGL (window);
          break;
        case kEventWindowShown:                                                // 5
          MyBuildGL (window);
          if (window == FrontWindow ())
                SetUserFocusWindow (window);
          InvalWindowRect (window, &rectPort);
          break;
        case kEventWindowBoundsChanged: //6)
          MyResizeGL (window, aglContext);
          MyHandleWindowUpdate(window);
          break;
        case kEventWindowZoomed:                                               // 7
          MyResizeGL (window, aglContext);
          break;
      }
      break;
  }
  return result;
}
```

Here's what the code does:

1. Stores the rendering context, which is passed to the event handler through the `userData` parameter.
2. Passes the activation event through, which prevents an initial flash of the screen.
3. Handles a draw content event by calling your window update function.
4. Handles a window close event by calling your dispose function to perform the necessary cleanup work.
5. Handles a window shown event by calling your function that performs the necessary work to render OpenGL to the window and to make the window frontmost with user focus.
6. Handles a window bounds changed event by resizing the window appropriately and then updating the content.
7. Handles a zoom event by resizing the window.

The code to handle the context update is shown in Listing 6-8. In its simplest form this code ensures the context of interest is current and then updates the context. Your application can also call the function `glViewport` to update the size of the drawable object to the current window size or to some other meaningful value. You might also want to update the projection matrix because the window dimensions have changed, and thus the relative geometry of the window has changed.

Listing 6-8 Updating a context using AGL

```
#include <Carbon/Carbon.h>
#include <AGL/agl.h>
#include <OpenGL/OpenGL.h>

void MyUpdateContextAGL (WindowRef window, AGLContext aglContext)
{
  Rect rectPort;
  aglSetCurrentContext (aglContext);
  aglUpdateContext (aglContext);
  GetWindowPortBounds (window, &rectPort);
  glViewport (0, 0, rectPort.right - rectPort.left,
                  rectPort.bottom - rectPort.top);
  /* Your code to update the projection matrix if needed */
}
```

It's slightly more complicated to handle changes in display configuration. You can detect these using Display Manager callback functions. (See *Display Manager Reference* and *Optimizing Display Modes and Window Arrangement With the Display Manager*.) You need to provide a callback function that conforms to the `DMExtendedNotificationProcPtr` callback. Then, after creating a universal procedure pointer to this function by calling the function `NewDMExtendedNotificationUPP`, register this UPP by calling the function `DMRegisterExtendedNotifyProc`.

Listing 6-9 shows the callback, the UPP creation and registration tasks, and other tasks you need for perform when handling display configuration changes. The callback function `handleWindowDMEvent` is simple. It calls the context update routine and invalidates the full window graphics port bounds to force an update event. Make sure to check for the `kDMNotifyEvent` message type; otherwise, the event is probably not one for which you need to update the context. If you use multiple rendering contexts or windows, it may be helpful to add the window or context to the user data.

Listing 6-9 Handling display configuration changes

```
#include <Carbon/Carbon.h>
#include <AGL/agl.h>

void handleWindowDMEvent (void *userData,
                            short msg, void *notifyData)
{
  AGLContext aglContext = (AGLContext) userData;
  if (kDMNotifyEvent == msg) {
    MyUpdateContextAGL (window, aglContext);
    GetWindowPortBounds (window, &rectPort);
    InvalWindowRect (window, &rectPort);
  }
}

void setupDMNotify (WindowRef window)
{
  gWindowEDMUPP = NewDMExtendedNotificationUPP(handleWindowDMEvent);
  DMRegisterExtendedNotifyProc (gWindowEDMUPP,
                     (void *)window, NULL, &psn);
}

OSStatus disposeDM Notify (WindowRef window)
{
  if (gWindowEDMUPP) {
    DisposeDMExtendedNotificationUPP (gWindowEDMUPP);
    gWindowEDMUPP = NULL;
  }
}
```

Updating Full-screen AGL and CGL Rendering Contexts

It's easier to update full-screen drawable objects than it is windowed ones since the drawable object position is fixed. Its size is directly linked to the display configuration, so full-screen applications need to perform updates only when they actually change the configuration. Instead of calling a context update routine, a full-screen application issues a set-full-screen call. Listing 6-10 and Listing 6-11 (page 75) show examples of AGL and CGL routines, respectively, to reset the full-screen context.

For AGL, the `aglSetFullScreen` function handles screen capture and display resizing; thus you just need to ensure that a valid full-screen pixel format object and rendering context are created prior to resizing. For CGL, you can use the Quartz Display Services functions `CGCaptureAllDisplays`, `CGDisplayBestModeForParametersAndRefreshRate` (or similar function), and `CGDisplaySwitchToMode` to set the requested display configuration. Then set the pixel format for the display and call the resize function.

Note: When you capture all displays, using either the function `aglSetFullScreen` (but without setting `AGL_FS_CAPTURE_SINGLE`) or the function `CGCaptureAllDisplays`, your application does not see any Display Manager notifications, because the display configuration is fixed and does not change until released. If you do not capture all displays, the application still receives display configuration changes for the noncaptured displays. Normally full-screen applications do not need to handle these display notifications, because they are for the displays not currently in use or of interest.

Listing 6-10 Handling full-screen updates using AGL

```
#include <Carbon/Carbon.h>
#include <AGL/agl.h>
#include <OpenGL/gl.h>

void MyResizeAGLFullScreen (AGLContext aglContext, GLSizei width,
                    GLSizei height)
{
  GLint displayCaps [3];

  if (!aglContext)                                                    // 1
    return;
  aglSetCurrentContext (aglContext);                                  // 2
  aglSetFullScreen (aglContext, width, height, 0, 0);                 // 3
  aglGetInteger (aglContext, AGL_FULLSCREEN, displayCaps);            // 4
  glViewport (0, 0, displayCaps[0], displayCaps[1]);                  // 5
  // Your code to update the projection matrix here if needed
}
```

Here's what the code does:

1. Checks for a valid context and returns if the context is not valid. Note that the `MyResizeAGLFullScreen` function assumes that the pixel format object associated with the context was created with the full-screen attribute.
2. Sets the context to the current context.
3. Attaches a full-screen drawable object to the context to ensure the context is updated.
4. Gets the display capabilities of the display, which are the width, height, and screen resolution.

5. Sets the viewport, specifying `(0,0)` as the left corner of the viewport rectangle, and the width and height of the screen (which were obtained with the previous call to `aglGetInteger`).

Listing 6-11 assumes that the pixel format object associated with the context was created with the full-screen, single display, and pixel depth attributes. Additionally, this code assumes that the screen is captured and set to the requested dimensions. The viewport is not set here since the calling routine actually sets the display size.

Listing 6-11 Handling full-screen updates using CGL

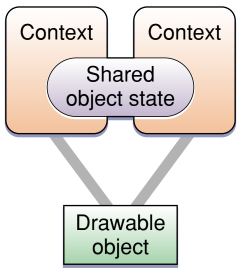
```
#include <Carbon/Carbon.h>
#include <OpenGL/OpenGL.h>
#include <OpenGL/gl.h>

void MyResizeCGL (CGLContextObj cglContext)
{
  if (!cglContext)
    return;
  CGLSetCurrentContext (cglContext);
  CGLSetFullScreen (cglContext);
}
```

Sharing Rendering Contexts

A rendering context does not own the drawing objects attached to it, which leaves open the option for sharing. Rendering contexts can share resources and can be attached to the same drawable object (see Figure 6-2) or to different drawable objects (see Figure 6-3). You set up context sharing—either with more than one drawable object or with another context—at the time you create a rendering context.

Contexts can share *object resources* and their associated *object state* by indicating a shared context at context creation time. Shared contexts share all texture objects, display lists, vertex programs, fragment programs, and buffer objects created before and after sharing is initiated. The state of the objects are also shared but not other context state, such as current color, texture coordinate settings, matrix and lighting settings, rasterization state, and texture environment settings. You need to duplicate context state changes as required, but you need to set up individual objects only once.

Figure 6-2 Shared contexts attached to the same drawable object

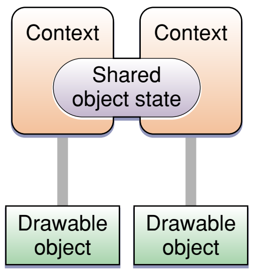


When you create an OpenGL context you can designate a second context to share object resources. All sharing is peer to peer. Shared resources are reference-counted and thus are maintained until explicitly released or when the last context sharing resource is released.

Not every context can be shared with every other context. Much depends on ensuring that the same set of renderers supports both contexts. You can meet this requirement by ensuring each context uses the same virtual screen list, using either of the following techniques:

- Use the same pixel format object to create all the rendering contexts that you want to share.
- Create pixel format objects using attributes that narrow down the choice to a single display. This practice ensures that the virtual screen is identical for each pixel format object.

Figure 6-3 Shared contexts and more than one drawable object



Setting up shared rendering contexts is very straightforward. Each Apple-specific OpenGL API provides functions with an option to specify a context to share in its context creation routine.

- Use the `share` argument for the `initWithFormat:shareContext:` method of the `NSOpenGLContext` class. See Listing 6-12 (page 76).
- Use the `share` parameter for the function `aglCreateContext`. See Listing 6-13 (page 77).
- Use the `share` parameter for the function `CGLCreateContext`. See Listing 6-14 (page 77).

Listing 6-12 ensures the same virtual screen list by using the same pixel format object for each of the shared contexts.

Listing 6-12 Setting up an NSOpenGLContext object for sharing

```
#import <Cocoa/Cocoa.h>
+ (NSOpenGLPixelFormat*)defaultPixelFormat
{
 NSOpenGLPixelFormatAttribute attributes [] = {
                    NSOpenGLPFADoubleBuffer,
                    (NSOpenGLPixelFormatAttribute)nil };
return [[(NSOpenGLPixelFormat *)[NSOpenGLPixelFormat alloc]
                        initWithAttributes:attribs] autorelease];
}

- (NSOpenGLContext*)openGLContextWithShareContext:(NSOpenGLContext*)context
{
     if (_openGLContext == NULL) {
            _openGLContext = [[NSOpenGLContext alloc]
                    initWithFormat:[[self class] defaultPixelFormat]
                    shareContext:context];
        [_openGLContext makeCurrentContext];
        [self prepareOpenGL];
            }
    return _openGLContext;
}

- (void)prepareOpenGL
{
     // Your code here to initialize the OpenGL state
}
```

Listing 6-13 sets up two pixel formats objects—`aglPixFmt` and `aglPixFmt2`—that share the same display.

Listing 6-13 Getting the same virtual screen list with different attributes

```
GLint attrib[] = {AGL_RGBA, AGL_DOUBLEBUFFER, AGL_FULL_SCREEN, AGL_NONE};
GLint attrib2[] = {AGL_RGBA, AGL_DOUBLEBUFFER, AGL_NONE};
disp = GetMainDevice();
aglPixFmt = aglChoosePixelFormat(&disp, 1, attrib);
aglContext = aglCreateContext(aglPixFmt, NULL);
//Use same display
aglPixFmt2 = aglChoosePixelFormat (&disp, 1, attrib2);
aglContext2 = aglCreateContext(aglPixFmt2, aglContext);
```

Listing 6-14 ensures the same virtual screen list by using the same pixel format object for each of the shared contexts.

Listing 6-14 Setting up a CGL context for sharing

```
#include <OpenGL/OpenGL.h>

CGLPixelFormatAttribute attrib[] = {kCGLPFADoubleBuffer, 0};
CGLPixelFormatObj pixelFormat = NULL;
long numPixelFormats = 0;
CGLContextObj cglContext1 = NULL;
CGLContextObj cglContext2 = NULL;
CGLChoosePixelFormat (attribs, &pixelFormat, &numPixelFormats);
CGLCreateContext(pixelFormat, NULL, &cglContext1);
CGLCreateContext(pixelFormat, cglContext1, &cglContext2);
```

See Also

OpenGL sample code projects (Sample Code > Graphics & Imaging > OpenGL):

- *GLCarbon1ContextPbuffer* demonstrates using a single shared rendering context with OpenGL pixel buffer objects.

Techniques for Choosing Attributes

Renderer and buffer attributes determine the renderers that the system chooses for your application. Each of the Apple-specific OpenGL APIs provides constants that specify a variety of renderer and buffer attributes. You supply a list of attribute constants to one of the Apple OpenGL functions for choosing a pixel format object. The pixel format object maintains a list of appropriate renderers. In previous chapters, you saw how to set up an attributes array that contains a small set of attributes.

In a real-world application, selecting attributes is an art because you don't know the exact combination of hardware and software that your application will run on. An attribute list that is too restrictive could miss out on future capabilities or not be able to run on many systems. For example, if you specify a buffer of a specific depth, your application won't be able to take advantage of a larger buffer when more memory is available in the future. In this case, you might specify a required minimum and direct OpenGL to use the maximum available.

Although you might want to specify attributes that make your OpenGL content look and run its best, you also need to consider whether you'll allow your application to run on a less-capable system at the expense of speed or detail. If tradeoffs are acceptable, you'll need to set the attributes accordingly.

Buffer Size Attribute Selection Tips

Follow these guidelines to choose buffer attributes that specify buffer size:

- To choose color, depth, and accumulation buffers that are greater than or equal to a size you specify, use the minimum policy attribute (`NSOpenGLPFAMinimumPolicy`, `AGL_MINIMUM_POLICY`, and `kCGLPFAMinimumPolicy`).
- To choose color, depth, and accumulation buffers that are closest in size to a size you specify, use the closest policy attribute (`NSOpenGLPFAClosestPolicy`, `AGL_CLOSEST_POLICY`, and `kCGLPFAClosestPolicy`).
- To choose the largest color, depth, and accumulation buffers available, use the maximum policy attribute (`NSOpenGLPFAMaximumPolicy`, `AGL_MAXIMUM_POLICY`, and `kCGLPFAMaximumPolicy`). As long as you pass a value that is greater than `0`, this attribute specifies the use of color, depth, and accumulation buffers that are the largest size possible.

Attributes that are not Recommended

There are several renderer and buffer attributes that are no longer recommended either because they are too narrowly focused or no longer useful:

- The robust attribute (`NSOpenGLPFARobust`, `AGL_ROBUST`, and `kCGLPFARobust`) specifies only those renderers that do not have any failure modes associated with a lack of video card resources.

- The multiple-screen attribute (`NSOpenGLPFAMultiScreen`, `AGL_MULTISCREEN`, and `kCGLPFAMultiScreen`) specifies only those renderers that can drive more than one screen at a time.
- The multiprocessing-safe attribute (`AGL_MP_SAFE` and `kCGLPFAMPSafe`) specifies only those renderers that are thread safe. This attribute is deprecated in Mac OS X because all renderers can accept commands for threads running on a second processor. However, this does not mean that all renderers are thread safe or reentrant. See "Multithreading and OpenGL" (page 117).
- The compliant attribute (`NSOpenGLPFACompliant`, `AGL_COMPLIANT`, and `kCGLPFACompliant`) specifies only OpenGL-compliant renderers. All Mac OS X renderers are OpenGL-compliant, so this attribute is no longer useful.

Ensuring that Back Buffer Contents Remain the Same

A backing store attribute (`NSOpenGLPFABackingStore`, `AGL_BACKING_STORE`, or `kCGLPFABackingStore`) is required whenever an application depends on the back buffer contents remaining the same after a swap buffer call.

Ensuring a Valid Pixel Format Object

The pixel format routines (the `initWithAttributes` method of the `NSOpenGLPixelFormat` class, `aglChoosePixelFormat`, and `CGLChoosePixelFormat`) return a pixel format object to your application that you use to create a rendering context. The buffer and renderer attributes that you supply to the pixel format routine determine the characteristics of the OpenGL drawing sent to the rendering context. If the system can't find at least one pixel format that satisfies the constraints specified by the attribute array, it returns `NULL` for the pixel format object. In this case, your application should have an alternative that ensures it can obtain a valid object.

One alternative is to set up your attribute array with the least restrictive attribute first and the most restrictive attribute last. Then, it is fairly easy to adjust the attribute list and make another request for a pixel format object. The code in Listing 7-1 illustrates this technique using the CGL API, but you can just as easily use Cocoa or the AGL API. Notice that the initial attributes list is set up with the supersample attribute last in the list. If the function `CGLChoosePixelFormat` returns `NULL` the first time it's called, the code sets the supersample attribute to `NULL` and once again requests a pixel format object.

Listing 7-1 Using the CGL API to create a pixel format object

```
int last_attribute = 6;
CGLPixelFormatAttribute attribs[] =
{
    kCGLPFAAccelerated,
    kCGLPFAColorSize, 24
    kCGLPFADepthSize, 16,
    kCGLPFADoubleBuffer,
    kCGLPFASupersample,
    0
};

CGLPixelFormatObj pixelFormatObj;
long numPixelFormats;
long value;

CGLChoosePixelFormat (attribs, &pixelFormatObj, &numPixelFormats);

if( pixelFormatObj == NULL ) {
    attribs[last_attribute] = NULL;
    CGLChoosePixelFormat (attribs, &pixelFormatObj, &numPixelFormats);
}

if( pixelFormatObj == NULL ) {
    // Your code to notify the user and take action.
}
```

Ensuring a Specific Type of Renderer

There are times when you'll want to ensure that you obtain a pixel format that supports a specific renderer type, such as a hardware-accelerated renderer. Table 7-1 lists attributes that support specific types of renderers. The table reflects the following tips for setting up pixel formats:

- To select only hardware-accelerated renderers, use both the accelerated and no recovery attributes.
- To use only the floating point software renderer, use the appropriate generic floating-point constant.
- To render to system memory, use the offscreen pixel attribute. Note that this rendering option does not use hardware acceleration.
- To render offscreen with hardware acceleration, specify a pixel buffer attribute. (See "Rendering to a Pixel Buffer" (page 48).)

Table 7-1 Renderer types and pixel format attributes

Renderer type	CGL	AGL	Cocoa
Hardware-accelerated onscreen	`kCGLPFAAccelerated` `kCGLPFANoRecovery`	`AGL_ACCELERATED` `AGL_NO_RECOVERY`	`NSOpen-GLPFAAccelerated` `NSOpenGLPFANoRecovery`
Software (floating-point)	`kCGLPFARendererID` `kCGLRendererGeneric-FloatID`	`AGL_RENDERER_ID` `AGL_RENDERER_-GENERIC_FLOAT_ID`	`NSOpenGLPFARendererID` `kCGLRendererGeneric-FloatID`
Software (deprecated on Intel-based Macs)	`kCGLPFARendererID` `kCGLRenderer-GenericID`	`AGL_RENDERER_ID` `AGL_RENDERER_-GENERIC_ID`	`NSOpenGLPFARendererID` `kCGLRendererGenericID`
System memory (not accelerated)	`kCGLPFAOffScreen`	`AGL_OFFSCREEN`	`NSOpenGLPFAOffScreen`
Hardware-accelerated offscreen	`kCGLPFAPBuffer`	`AGL_PBUFFER`	`NSOpenGLPFAPixel-Buffer`

Ensuring a Single Renderer for a Display

In some cases you may want to use a specific hardware renderer and nothing else. Since the OpenGL framework normally provides a software renderer as a fallback in addition to whatever hardware renderer it chooses, you need to prevent OpenGL from choosing the software renderer as an option. You either need to specify the no recovery attribute for a windowed drawable object or use a full-screen drawable object. (The full-screen attribute always implies not to use the software renderer as an option.)

Limiting a context to use a specific display, and thus a single renderer, has its risks. If your application runs on a system that uses more than one display, then dragging a windowed drawable object from one display to the other will likely yield a less than satisfactory result. Either the rendering will fail, or OpenGL uses the specified renderer to copy the drawing to the second display. The same unsatisfactory result happens when attaching a full-screen context to another display. If you choose to use the hardware renderer associated with a specific display, you need to add code that detects and handles display changes.

The three code examples that follow show how to use each of the Apple-specific OpenGL APIs to set up a context that uses a single renderer. Listing 7-2 shows how to set up an `NSOpenGLPixelFormat` object that supports a single renderer. The attribute `NSOpenGLPFANoRecovery` specifies to OpenGL not to provide the fallback option of the software renderer.

Listing 7-2 Setting an `NSOpenGLContext` object to use a specific display

```
#import <Cocoa/Cocoa.h>
+ (NSOpenGLPixelFormat*)defaultPixelFormat
{
    NSOpenGLPixelFormatAttribute attributes [] = {
                        NSOpenGLPFAScreenMask, 0,
                        NSOpenGLPFANoRecovery,
                        NSOpenGLPFADoubleBuffer,
                        (NSOpenGLPixelFormatAttribute)nil };
CGDirectDisplayID display = CGMainDisplayID ();
// Adds the  display mask attrib for selected display
attributes[1] = (NSOpenGLPixelFormatAttribute)
                    CGDisplayIDToOpenGLDisplayMask (display);
return [[(NSOpenGLPixelFormat *)[NSOpenGLPixelFormat alloc]
initWithAttributes:attributes]
                                       autorelease];
}
```

Listing 7-3 shows how to use AGL to set up a context that uses a single renderer. The attribute `AGL_NO_RECOVERY` specifies to OpenGL not to provide the fallback option of the software renderer.

Listing 7-3 Setting an AGL context to use a specific display

```
#include <AGL/agl.h>
GLint attrib[] = {AGL_RGBA, AGL_DOUBLEBUFFER, AGL_NO_RECOVERY, AGL_NONE};
GDHandle display = GetMainDevice ();
AGLPixelFormat aglPixFmt = aglChoosePixelFormat (&display, 1, attrib);
```

Listing 7-4 shows how to use CGL to set up a context that uses a single renderer. The attribute `kCGLPFAFullScreen` ensures that OpenGL does not provide the fallback option of the software renderer.

Listing 7-4 Setting a CGL context to use a specific display

```
#include <OpenGL/OpenGL.h>
CGLPixelFormatAttribute attribs[] = { kCGLPFADisplayMask, 0,
                                 kCGLPFAFullScreen,
                                 kCGLPFADoubleBuffer,
                                  0 };
CGLPixelFormatObj pixelFormat = NULL;
long numPixelFormats = 0;
CGLContextObj cglContext = NULL;
CGDirectDisplayID display = CGMainDisplayID ();
// Adds the  display mask attrib for selected display
attribs[1] = CGDisplayIDToOpenGLDisplayMask (display);
CGLChoosePixelFormat (attribs, &pixelFormat, &numPixelFormats);
```

See Also

Reference documentation for buffer and renderer attributes in the Constants sections of:

- *AGL Reference*
- *CGL Reference*
- *NSOpenGLPixelFormat Class Reference*

Techniques for Working with Vertex Data

Complex shapes and detailed 3D models require large amounts of vertex data to describe them in OpenGL. Moving vertex data from your application to the graphics hardware incurs a performance cost that can be quite large depending on the size of the data set. Applications that use large vertex data sets can adopt one or more strategies to optimize how the data flows to OpenGL.

Figure 8-1 Vertex data sets can be quite large

```
short face_indicies[16301][6] = {
    {1538,2410,1101 ,0,1,2 },      6196      4,5 },      {696,704,101 ,6,7,5 },
    {1192,117,1113 ,8,9,10 },              11,12,13 }, {540,1192,1113 ,14,8,10 },
    {704,696,398 ,7,6,15 },                ,15,6,16 },    {1605,772,4540 ,17,18,19 },
    {1192,540,1857 ,8,14,13              ,809 ,18,3,20 },    {713,173,1260 ,3,12,11 },
    {173,1192,1857 ,12,8,13            ,1260 ,20,3,11 },   {804,4128,4120 ,21,22,23 },
    {1133,804,4120 ,24,21,23                          },{4140,1534,4136 ,25,27,28 },
    {4531,4140,4136 ,26,25,2                            {4515,1609,4485 ,30,31,32 },
    {710,1396,889 ,33,34,35                              2046,1905,768 ,39,40,41 },
    {2453,5699,5698 ,42,43,44                            68,1905,1393 ,41,40,45 },
    {1393,1490,4112 ,45,46,47                            490,4281,839 ,46,51,52 },
    {4112,1490,839 ,47,46,52                             397,8079,3286 ,54,55,56 },
    {920,4291,1937 ,57,58,59 },                          4915,4914 ,63,64,65 },
    {1093,4537,840 ,66,67,68 },                          59,4158 ,70,71,72 },
    {7248,7323,7322 ,73,74,75 },                         5289,5266 ,79,80,81 },
    {4197,4217,4258 ,82,83,84 },                         98,4536,4535 ,88,89,90 },
```

This chapter provides best practices for working with vertex data, describes how to use extensions to optimize performance, shows how to use a fence command to test for completion of OpenGL commands, and discusses how to set up double buffers.

Best Practices for Working with Vertex Data

Understanding how vertex data flows through an OpenGL program is important to choosing strategies for handling the data. Vertex data can travel through OpenGL in two ways, as shown in Figure 8-2. The first way, from vertex data to per-vertex operations, is as part of an OpenGL command sequence that is issued by the application and executed immediately (immediate mode). The second is packaged as a named display list that can be preprocessed ahead of time and used later in the program.

Figure 8-2 Vertex data path

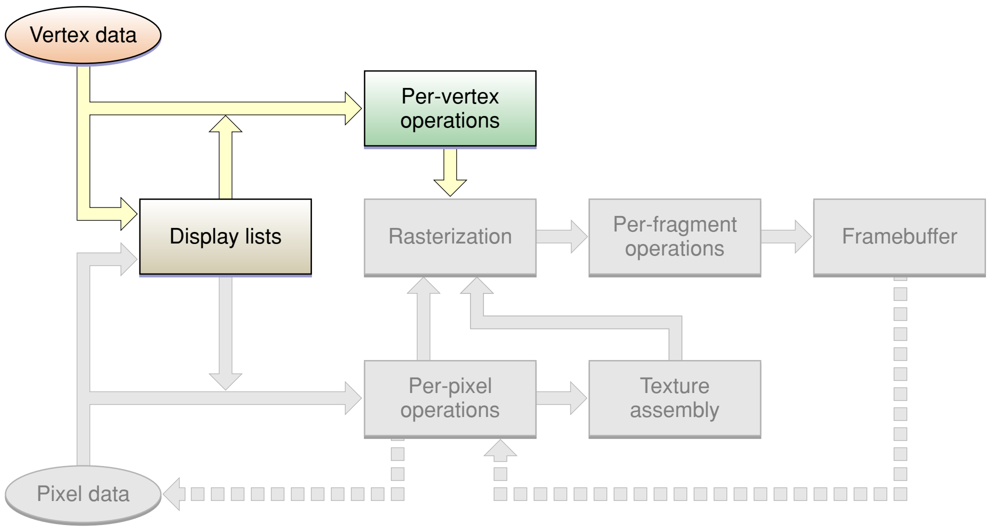
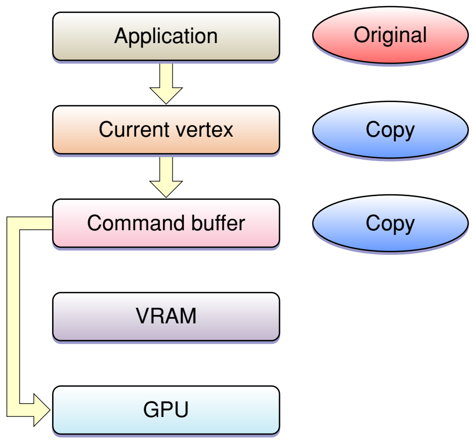


Figure 8-3 provides a closer look at the vertex data path when using immediate mode. Without any optimizations, your vertex data can be copied at various points in the data path. OpenGL is required to capture the current vertex state when you use immediate mode. If your code uses functions that operate on vertex arrays, you can eliminate the command buffer copy shown in Figure 8-3. The OpenGL commands `glDrawRangeElements`, `glDrawElements`, and `glDrawArrays` render multiple geometric primitives from array data, using very few subroutine calls. It's best to use `glDrawRangeElements`, with `glDrawElements` the second choice, and `glDrawArrays` the third.

Figure 8-3 Immediate mode requires a copy of the current vertex data



In addition to using functions that operate on vertex arrays, there are a number of other strategies that you can adopt to optimize the flow of vertex data in your application:

- Minimize data type conversions by supplying OpenGL data types for vertex data. Use `GLfloat`, `GLshort`, or `GLubyte` data types because most graphics processors handle these types natively. If you use some other type, then OpenGL may need to perform a costly data conversion.

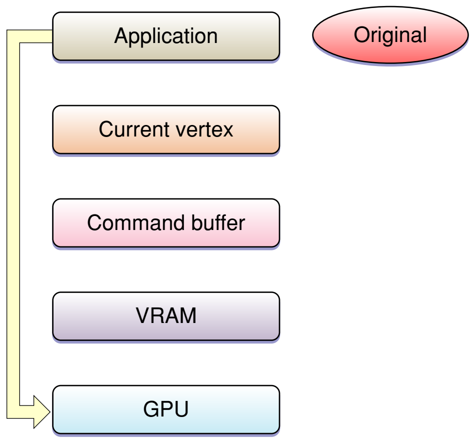
- The most desirable way to handle vertex data is to use the `GL_APPLE_vertex_array_range` or `GL_ARB_vertex_buffer_object` extensions. (See "Using Extensions to Improve Performance" (page 87).) If you can't use these extensions, then make sure you use vertex arrays and display lists. Avoid using immediate mode. But if your code must use immediate mode, maximize the number of vertices per draw command or within a begin-end code block.
- Use vertex programs to perform computations on vertex data instead of using the CPU to perform the computations.
- If your code must use immediate mode, use CGL macros (for Cocoa or Carbon) or AGL macros (Carbon only). Macros use the function call dispatch table directly, which can dramatically reduce function call overhead. See "Use OpenGL Macros" (page 125).

Using Extensions to Improve Performance

The vertex array range (`GL_APPLE_vertex_array_range`) and vertex buffer object (`GL_ARB_vertex_buffer_object`) extensions were created to help streamline the vertex data path. Although both can improve application performance, the vertex buffer object extension should be your first choice and the vertex array range extension your second. The vertex array range extension provides the GPU with direct access to your data. When your data is dynamic, the burden is on your application to synchronize access to that data. Vertex buffer objects, on the other hand, don't require your application to synchronize data access, which is the primary reason why they are preferred. You'll read more about each extension in the sections that follow.

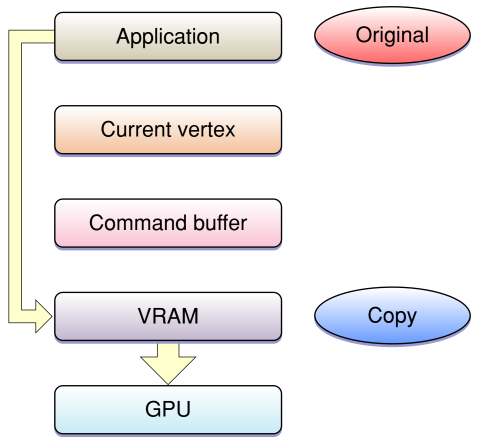
For dynamic vertex array data, these extensions set up DMA from the application to the GPU, as shown in Figure 8-4. Notice that copies of the vertex data are not maintained in VRAM. This means that each time the data is drawn, it gets moved from the application to the GPU. It's important to ensure that this happens asynchronously.

Figure 8-4 Extensions allow dynamic data to use DMA



For static vertex data, you can use these extensions to cache the data in VRAM, which allows the data to utilize the full bandwidth of the graphics processor bus, as shown in Figure 8-5. Data needs to be copied to VRAM only once.

Figure 8-5 Extensions allow static vertex data to use VRAM storage



The next sections describe these extensions in more detail as well as the Apple fence extension (`GL_APPLE_fence`), which is used to synchronize drawing commands.

Vertex Array Range Extension

The vertex array range extension (`APPLE_vertex_array_range`) lets you define a region of memory for your vertex data. This allows the OpenGL driver to optimize memory usage by creating a single memory mapping for your vertex data. You can also provide a hint as to how the data should be stored: cached or shared. The cached option specifies to cache vertex data in video memory. The shared option indicates that data should be mapped into a region of memory that allows the GPU to access the vertex data directly using DMA transfer. This option is best for dynamic data. If you use shared memory, you'll need to double buffer your data. See "Double Buffering Vertex Data" (page 93).

You can set up and use the vertex array range extension by following these steps:

1. Enable the extension by calling `glEnableClientState` and supplying the `GL_VERTEX_ARRAY_RANGE_APPLE` constant.

2. Allocate storage for the vertex data. You are responsible for maintaining storage for the data.

3. Define an array of vertex data by calling a function such as `glVertexPointer`. You need to supply a pointer to your data.

4. Optionally set up a hint about handling the storage of the array data by calling the function `glVertexArrayParameteriAPPLE`.

   ```
   GLvoid glVertexArrayParameteriAPPLE(GLenum pname, GLint param);
   ```

   `pname` must be `VERTEX_ARRAY_STORAGE_HINT_APPLE`.

   `param` is a hint that specifies how your application expects to use the data. OpenGL uses this hint to optimize performance. You can supply either `STORAGE_SHARED_APPLE` or `STORAGE_CACHED_APPLE`. The default value is `STORAGE_SHARED_APPLE`, which indicates that the vertex data is dynamic and that OpenGL should use optimization and flushing techniques suitable for this kind of data. If you expect the data to be static supply, `STORAGE_CACHED_APPLE` so that OpenGL uses VRAM caching and other techniques to optimize memory bandwidth.

5. Call the OpenGL function glVertexArrayRangeAPPLE to establish the data set.

   ```
   void glVertexArrayRangeAPPLE(GLsizei length, GLvoid *pointer);
   ```

   length specifies the length of the vertex array range. The length is typically the number of unsigned bytes.

   *pointer points to the base of the vertex array range.

6. Draw with the vertex data using standard OpenGL vertex array commands.

7. Call glFlushVertexArrayRangeAPPLE.

   ```
   void glFlushVertexArrayRangeAPPLE(GLsizei length, GLvoid *pointer);
   ```

   length specifies the length of the vertex array range, in bytes.

   *pointer points to the base of the vertex array range.

   For dynamic data, each time you change the data, you need to maintain synchronicity by calling glFlushVertexArrayRangeAPPLE. You supply as parameters an array size and a pointer to an array, which can be a subset of the data, as long as it includes all of the data that changed. Contrary to the name of the function, glFlushVertexArrayRangeAPPLE doesn't actually flush data like the OpenGL function glFlush does. It simply makes OpenGL aware that the data has changed.

   To make sure that your data is fully coherent, in addition to calling glFlushVertexArrayRangeAPPLE after drawing and prior to modifying the data, you need either to call glFinish or to set up a fence. The APPLE_fence extension lets you set up selective synchronization. See "Fence Extension" (page 92) and "Double Buffering Vertex Data" (page 93).

Listing 8-1 shows code that sets up and uses the vertex array range extension with dynamic data. It overwrites all of the vertex data during each iteration through the drawing loop. The call to the glFinishFenceAPPLE command guarantees that the CPU and the GPU don't access the data at the same time. Although this example calls the glFinishFenceAPPLE function almost immediately after setting the fence, in reality you need to separate these calls to allow parallel operation of the GPU and CPU. To see how that's done, read "Double Buffering Vertex Data" (page 93).

Listing 8-1 Using the vertex array range extension with dynamic data

```
//  To set up the vertex array range extension
glVertexArrayParameteriAPPLE(GL_VERTEX_ARRAY_STORAGE_HINT_APPLE,
GL_STORAGE_SHARED_APPLE);
glVertexArrayRangeAPPLE(buffer_size, my_vertex_pointer);
glEnableClientState(GL_VERTEX_ARRAY_RANGE_APPLE);

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, my_vertex_pointer);
glSetFenceAPPLE(my_fence);

//  When you want to draw using the vertex data
draw_loop {
    glFinishFenceAPPLE(my_fence);
    GenerateMyDynamicVertexData(my_vertex_pointer);
    glFlushVertexArrayRangeAPPLE(buffer_size, my_vertex_pointer);
    PerformDrawing();
    glSetFenceAPPLE(my_fence);
```

```
}
```

Listing 8-2 shows code that uses the vertex array range extension with static data. Unlike the setup for dynamic data, the setup for static data includes using the hint for cached data. Because the data is static, it's unnecessary to set a fence.

Listing 8-2 Using the vertex array range extension with static data

```
//  To set up the vertex array range extension
GenerateMyStaticVertexData(my_vertex_pointer);
glVertexArrayParameteriAPPLE(GL_VERTEX_ARRAY_STORAGE_HINT_APPLE,
GL_STORAGE_CACHED_APPLE);
glVertexArrayRangeAPPLE(array_size, my_vertex_pointer);
glEnableClientState(GL_VERTEX_ARRAY_RANGE_APPLE);

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, stride, my_vertex_pointer);

//  When you want to draw using the vertex data
draw_loop {
    PerformDrawing();
}
```

For detailed information on this extension, see the OpenGL specification for the vertex array range extension.

Vertex Buffer Object Extension

The vertex buffer object extension (`GL_ARB_vertex_buffer_object`) can be used along with vertex arrays to improve the throughput of static or dynamic vertex data in your application. A buffer object is a chunk of memory. You can read and write directly to this memory using OpenGL calls such as `glBufferData`, `glBufferSubData`, and `glGetBufferSubData` or you can access memory through a pointer, an operation referred to as *mapping a buffer*.

You can set up and use the vertex buffer object extension by following these steps:

1. Call the function `glBindBufferARB` to bind an unused name to a buffer object. After this call, the newly created buffer object is initialized with a memory buffer of size zero and a default state. (For the default setting, see the OpenGL specification for ARB_vertex_buffer_object.)

   ```
   void glBindBufferARB(GLenum target, GLuint buffer);
   ```

   `target` must be set to `GL_ARRAY_BUFFER_ARB`.

   `buffer` specifies the unique name for the buffer object.

2. So that you can use vertex arrays with the vertex buffer object, enable the vertex array by calling `glEnableClientState` and supplying the `GL_VERTEX_ARRAY` constant.

3. Define an array of vertex data by calling a function such as `glVertexPointer`. You need to supply an offset into your data buffer.

4. Create and initialize the data store of the buffer object by calling the function `glBufferDataARB`. Essentially, this call uploads your data to the GPU.

   ```
   void glBufferDataARB(GLenum target, sizeiptrARB size,
   ```

```
                const GLvoid *data, GLenum usage);
```

`target` must be set to `GL_ARRAY_BUFFER_ARB`.

`size` specifies the size of the data store.

`*data` points to the source data. If this is not `NULL`, the source data is copied to the data store of the buffer object. If `NULL`, the contents of the data store are undefined.

`usage` is a constant that provides a hint as to how your application plans to use the data store. You can supply any of nine constants defined by the OpenGL specification. OpenGL uses the hint to optimize performance, not to constrain your use of the data. You'll see two of these constants in the examples that follow: `GL_STREAM_DRAW_ARB`, which indicates that the application plans to draw with the data repeatedly and to modify the data, and `GL_STATIC_DRAW_ARB`, which indicates that the application will define the data once but use it to draw many times.

5. Map the data store of the buffer object to your application address space by calling the function `glMapBufferARB`.

   ```
   void *glMapBufferARB(GLenum target, GLenum access);
   ```

   `target` must be set to `GL_ARRAY_BUFFER_ARB`.

   `access` indicates the operations you plan to perform on the data. You can supply `READ_ONLY_ARB`, `WRITE_ONLY_ARB`, or `READ_WRITE_ARB`.

6. Write the vertex data to its destination.

7. When you no longer need the vertex data, call the function `glUnmapBufferARB`. You must supply `GL_ARRAY_BUFFER_ARB` as the parameter to this function.

Listing 8-3 shows code that uses the vertex buffer object extension for dynamic data. This example overwrites all of the vertex data during every draw operation.

Listing 8-3 Using the vertex buffer object extension with dynamic data

```
//  To set up the vertex buffer object extension
#define BUFFER_OFFSET(i) ((char*)NULL + (i))
glBindBufferARB(GL_ARRAY_BUFFER_ARB, myBufferName);

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, stride, BUFFER_OFFSET(0));

//  When you want to draw using the vertex data
draw_loop {
    glBufferDataARB(GL_ARRAY_BUFFER_ARB, bufferSize, NULL, GL_STREAM_DRAW_ARB);
    my_vertex_pointer = glMapBufferARB(GL_ARRAY_BUFFER_ARB, GL_WRITE_ONLY_ARB);
     GenerateMyDynamicVertexData(my_vertex_pointer);
     glUnmapBufferARB(GL_ARRAY_BUFFER_ARB);
     PerformDrawing();
}
```

Listing 8-4 shows codes that uses the vertex buffer object extension with static data.

Listing 8-4 Using the vertex buffer object extension with static data

```
//  To set up the vertex buffer object extension
#define BUFFER_OFFSET(i) ((char*)NULL + (i))
glBindBufferARB(GL_ARRAY_BUFFER_ARB, myBufferName);
glBufferDataARB(GL_ARRAY_BUFFER_ARB, bufferSize, NULL, GL_STATIC_DRAW_ARB);
GLvoid* my_vertex_pointer = glMapBufferARB(GL_ARRAY_BUFFER_ARB,
GL_WRITE_ONLY_ARB);
GenerateMyStaticVertexData(my_vertex_pointer);
glUnmapBufferARB(GL_ARRAY_BUFFER_ARB);

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, stride, BUFFER_OFFSET(0));

//  When you want to draw using the vertex data
draw_loop {
    PerformDrawing();
}
```

For detailed information on this extension, see the OpenGL specification for the vertex buffer object extension.

Fence Extension

The fence extension (`APPLE_fence`) is designed to synchronize drawing commands with modifications that you make to vertex data. A **fence** is a token used to mark the current point in the command stream. When used correctly, it allows you to ensure that drawing with a range of vertex array data (whether it's the entire set or a subset) is complete before you modify the data. When you use the fence you must synchronize the data.

This extension was created because the OpenGL commands `glFlush` and `glFinish` don't offer the level of granularity that is often needed to synchronize drawing and data modifications. A fence can help you coordinate activity between the CPU and the GPU when they are using the same resources. You'll want to use a fence when you are using the vertex array range extension for dynamic data. You do not need to use a fence for vertex buffer objects, but you do need to use a fence when you use the vertex array range extension and the shared memory hint.

Follow these steps to set up and use a fence:

1. Set up the fence by calling the function `glSetFenceApple`. This function inserts a token into the command stream and sets the fence state to `false`.

   ```
   void glSetFenceAPPLE(GLuint fence);
   ```

   `fence` specifies the token to insert. For example:

   ```
   GLint myFence = 1;
   glSetFenceAPPLE(myFence);
   ```

2. Wait for all OpenGL commands issued prior to the fence to complete by calling the function `glFinishFenceApple`.

   ```
   void glFinishFenceAPPLE(GLuint fence);
   ```

   `fence` specifies the token that was inserted previously. For example:

```
    glFinishFenceAPPLE(myFence);
```

There is an art to determining where to insert a fence in the command stream. If you insert a fence for too few drawing commands, you risk having your application stall while it waits for drawing to complete. You'll want to set a fence so your application operates as asynchronously as possible without stalling.

The fence extension also lets you synchronize buffer updates for objects such as vertex arrays and textures. For that you call the function `glFinishObjectAPPLE`, supplying an object name along with the token.

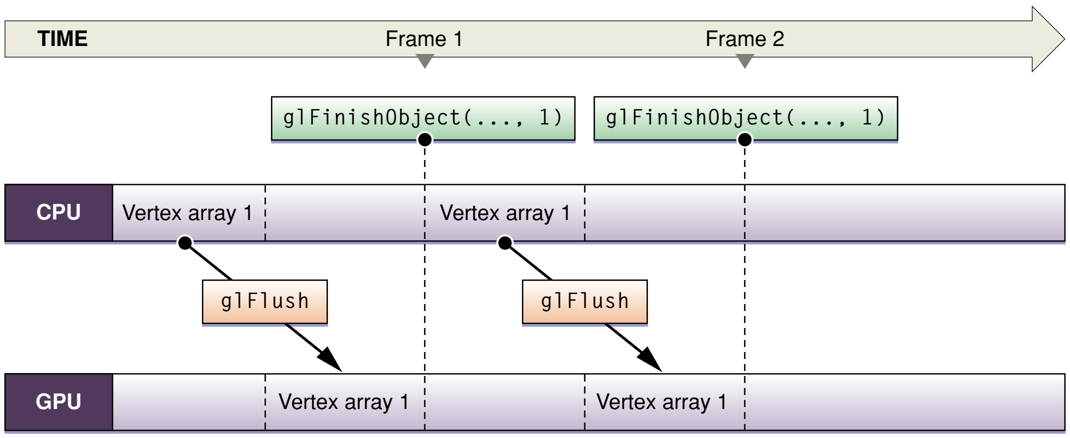
For detailed information on this extension, see the OpenGL specification for the Apple fence extension.

Double Buffering Vertex Data

When you use the vertex array range extension and the shared memory hint, the GPU reads data directly from memory managed by your application. To avoid having the GPU and your application access the data at the same time, you'll need to synchronize access. A simple approach is for your application to operate on the vertex array data, flush it to the GPU, and wait until the GPU is finished before working on the data again. This is what Figure 8-6 shows.

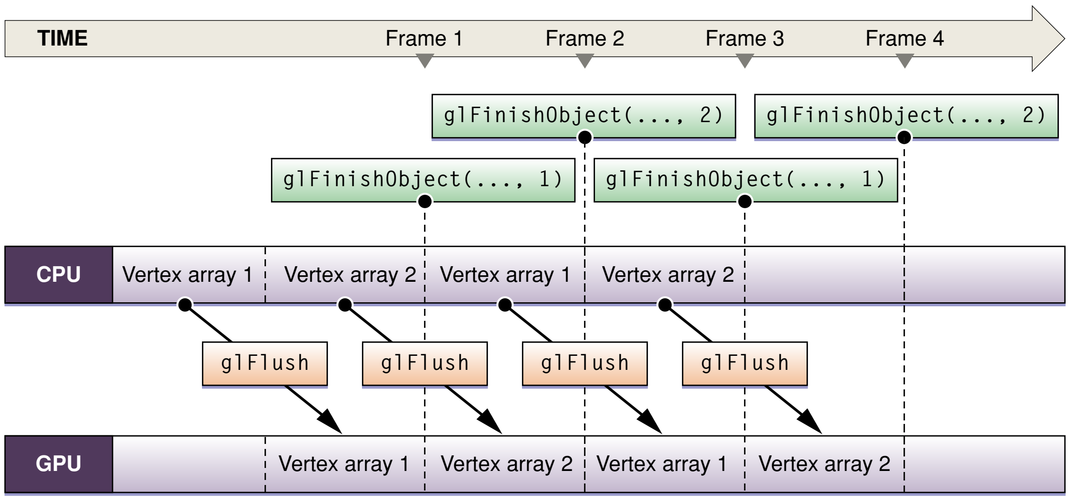
To ensure that the GPU is finished executing commands before the CPU sends more data, you can insert a token into the command stream and use that to determine when the CPU can touch the data again, as described in "Fence Extension" (page 92). Figure 8-6 uses the fence extension command `glFinishObject` to synchronize buffer updates. Notice that when the CPU is processing data, the GPU is waiting. Similarly, when the GPU is processing data, the CPU is waiting. In other words, the application executes synchronously. A more efficient way is for the application to double buffer your data so that you can use the waiting time to process more data.

Figure 8-6 Single-buffered vertex array data



To double buffer your data, you must supply two sets of data to work on. Notice in Figure 8-7 that while the GPU is operating on one set of vertex array data, the CPU is processing the next. After the initial startup, neither processing unit is idle. Using the `glFinishObject` function provided by the fence extension, as shown, ensures that buffer updates are synchronized

Figure 8-7 Double-buffered vertex array data



See Also

OpenGL extension specifications:

- APPLE_vertex_array_range
- ARB_vertex_buffer_object
- APPLE_fence

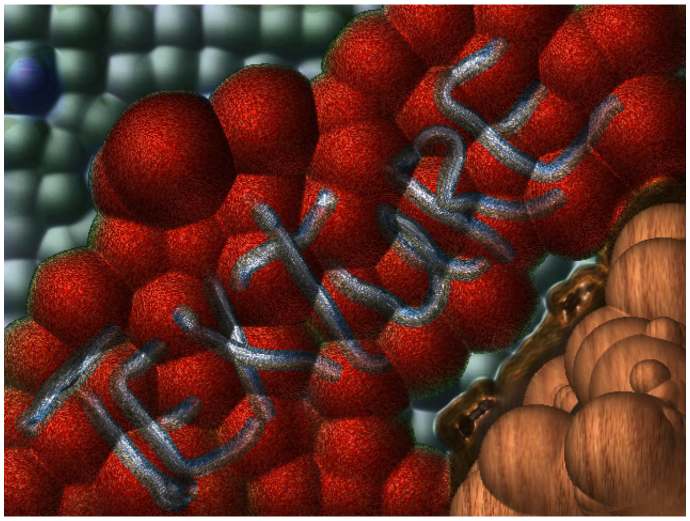
OpenGL sample code projects (Sample Code > Graphics & Imaging > OpenGL):

- *Vertex Optimization* demonstrates different ways to optimize vertex programs.
- *VertexPerformanceTest* shows slow and fast vertex data paths.
- *VertexPerformanceDemo* measures triangle throughput and compares different coding methods.

Techniques for Working with Texture Data

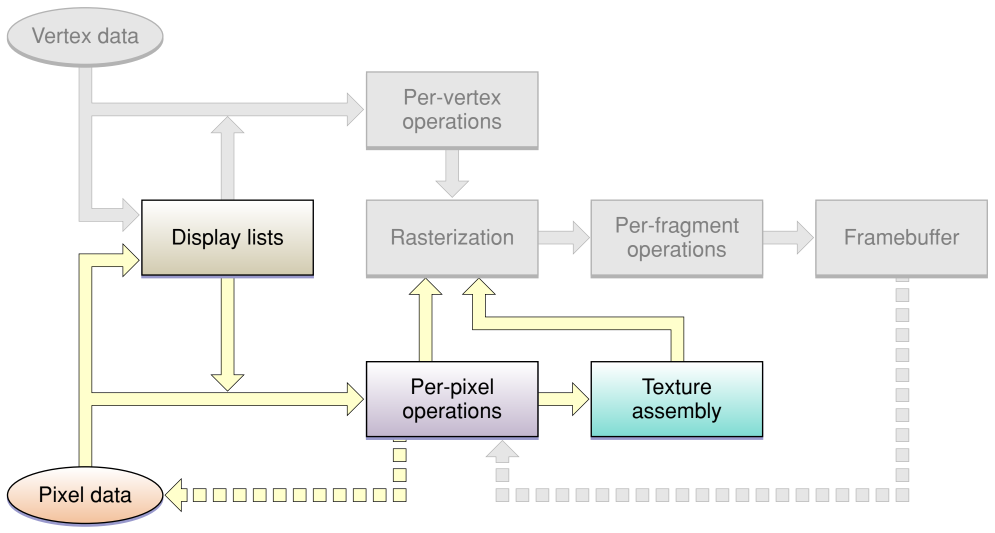
Textures add realism to OpenGL objects. They are what makes the objects defined by vertex data take on the material properties of real-world objects, such as wood, brick, metal, and fur. Texture data can originate from many sources, including images. As with vertex data, there are a variety of techniques you can use to minimize the number of times texture data is copied and converted as it's moved throughout the system.

Figure 9-1 Textures add realism to a scene



Textures start as pixel data that flows through an OpenGL program, as shown in Figure 9-2. As with vertex data you can supply pixel data in two ways. The first way, from pixel data to per-pixel operations, is as part of an OpenGL command sequence that is issued by the application and executed immediately (immediate mode). The second is packaged as a named display list that can be preprocessed ahead of time and used later in the program.

Figure 9-2 Texture data path



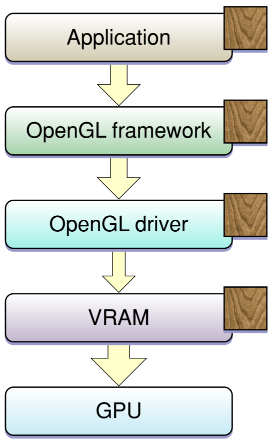
The precise route that texture data takes from your application to its final destination can impact the performance of your application. The purpose of this chapter is to provide techniques you can use to ensure optimal processing of texture data in your application. This chapter

- shows how to use OpenGL extensions to optimize performance
- lists optimal data formats and types
- provides information on working with textures whose dimensions are not a power of two
- describes creating textures from image data
- shows how to download textures
- discusses using double buffers for texture data

Using Extensions to Optimize

Without any optimizations, texture data flows through an OpenGL program as shown in Figure 9-3. Data from your application first goes to the OpenGL framework, which may make a copy of the data before handing it to the driver. If your data is not in a native format for the hardware (see "Optimal Data Formats and Types" (page 101)), the driver may also make a copy of the data to convert it to a hardware-specific format for uploading to video memory. Video memory, in turn, can keep a copy of the data. Theoretically, there could be four copies of your texture data throughout the system.

Figure 9-3 Data copies in an OpenGL program



Data flows at different rates through the system, as shown by the size of the arrows in Figure 9-3. The fastest data transfer happens between VRAM and the GPU. The slowest transfer occurs between the OpenGL driver and VRAM. Data moves between the application and the OpenGL framework, and between the framework and the driver at the same "medium" rate. Eliminating any of the data transfers, but the slowest one in particular, will improve application performance.

There are several extensions you can use to eliminate one or more data copies and control how texture data travels from your application to the GPU:

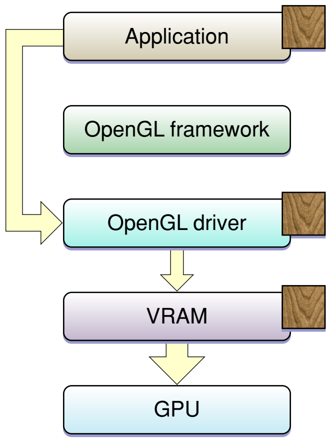
- `GL_APPLE_client_storage`
- `GL_APPLE_texture_range` along with a storage hint, either `GL_STORAGE_CACHED_APPLE` or `GL_STORAGE_SHARED_APPLE`
- `GL_ARB_texture_rectangle`

The sections that follow describe the extensions and show how to use them.

Apple Client Storage

The Apple client storage extension (`APPLE_client_storage`) lets you provide OpenGL with a pointer to memory that your application allocates and maintains. OpenGL retains a pointer to your data but does not copy the data. Because OpenGL references your data, this extension requires that you retain a copy of your texture data until it is no longer needed. By using this extension you can eliminate the OpenGL framework copy as shown in Figure 9-4. Note that a texture width must be a multiple of 32 bytes for OpenGL to bypass the copy operation from the application to the OpenGL framework.

Figure 9-4 The client storage extension eliminates a data copy



The Apple client storage extension defines a pixel storage parameter, `GL_UNPACK_CLIENT_STORAGE_APPLE`, that you pass to the OpenGL function `glPixelStorei` to specify that your application retains storage for textures. The following code sets up client storage:

```
glPixelStorei(GL_UNPACK_CLIENT_STORAGE_APPLE, GL_TRUE);
```

For detailed information, see the OpenGL specification for the Apple client storage extension.

Apple Texture Range and Rectangle Texture

The Apple texture range extension (`APPLE_texture_range`) lets you define a region of memory used for texture data. Typically you specify an address range that encompasses the storage for a set of textures. This allows the OpenGL driver to optimize memory usage by creating a single memory mapping for all of the textures. You can also provide a hint as to how the data should be stored: cached or shared. The cached hint specifies to cache texture data in video memory. This hint is recommended when you have textures that you plan to use multiple times or that use linear filtering. The shared hint indicates that data should be mapped into a region of memory that enables the GPU to access the texture data directly (via DMA) without the need to copy it. This hint is best when you are using large images only once, perform nearest-neighbor filtering, or need to scale down the size of an image.

The texture range extension defines the following routine for making a single memory mapping for all of the textures used by your application:

```
void glTextureRangeAPPLE(GLenum target, GLsizei length, GLvoid *pointer);
```

`target` is a valid texture target, such as `GL_TEXTURE_2D`.

`length` specifies the number of bytes in the address space referred to by the `pointer` parameter.

`*pointer` points to the address space that your application provides for texture storage.

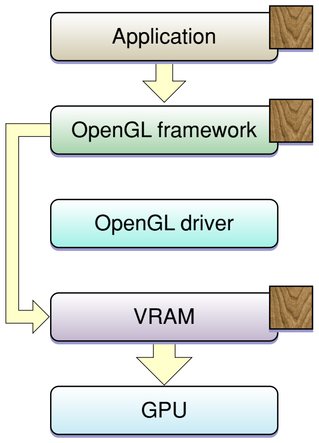
You provide the hint parameter and a parameter value to to the OpenGL function `glTexParameteri`. The possible values for the storage hint parameter (`GL_TEXTURE_STORAGE_HINT_APPLE`) are `GL_STORAGE_CACHED_APPLE` or `GL_STORAGE_SHARED_APPLE`.

Some hardware requires texture dimensions to be a power-of-two before the hardware can upload the data using DMA. The rectangle texture extension (`ARB_texture_rectangle`) was introduced to allow texture targets for textures of any dimensions—that is, rectangle textures (`GL_TEXTURE_RECTANGLE_ARB`). You need to use the rectangle texture extension together with the Apple texture range extension to ensure OpenGL uses DMA to access your texture data. These extensions allow you to bypass the OpenGL driver, as shown in Figure 9-5.

Note that OpenGL does not use DMA for a power-of-two texture target (`GL_TEXTURE_2D`). So, unlike the rectangular texture, the power-of-two texture will incur one additional copy and performance won't be quite as fast. The performance typically isn't an issue because games, which are the applications most likely to use power-of-two textures, load textures at the start of a game or level and don't upload textures in real time as often as applications that use rectangular textures, which usually play video or display images.

The next section has code examples that use the texture range and rectangle textures together with the Apple client storage extension.

Figure 9-5 The texture range extension eliminates a data copy

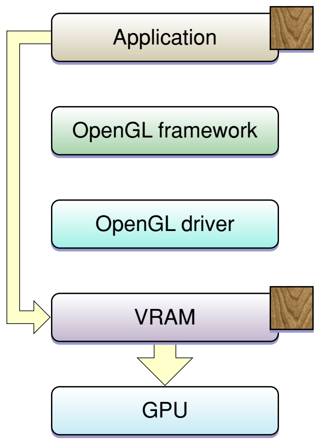


For detailed information on these extensions, see the OpenGL specification for the Apple texture range extension and the OpenGL specification for the ARB texture rectangle extension.

Combining Extensions

You can use the Apple client storage extension along with the Apple texture range extension to streamline the texture data path in your application. When used together, OpenGL moves texture data directly into video memory, as shown in Figure 9-6. The GPU directly accesses your data (via DMA). The set up is slightly different for rectangular and power-of-two textures. The code examples in this section upload textures to the GPU. You can also use these extensions to download textures, see "Downloading Texture Data" (page 107).

Figure 9-6 Combining extensions to eliminate data copies



Listing 9-1 shows how to use the extensions for a rectangular texture. After enabling the texture rectangle extension you need to bind the rectangular texture to a target. Next, set up the storage hint. Call `glPixelStorei` to set up the Apple client storage extension. Finally, call the function `glTexImage2D` with a with a rectangular texture target and a pointer to your texture data.

> **Note:** The texture rectangle extension limits what can be done with rectangular textures. To understand the limitations in detail, read the OpenGL extension for texture rectangles. See "Working with Non–Power-of-Two Textures" (page 101) for an overview of the limitations and an alternative to using this extension.

Listing 9-1 Using texture extensions for a rectangular texture

```
glEnable (GL_TEXTURE_RECTANGLE_ARB);
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, id);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
        GL_TEXTURE_STORAGE_HINT_APPLE,
        GL_STORAGE_CACHED_APPLE);
glPixelStorei(GL_UNPACK_CLIENT_STORAGE_APPLE, GL_TRUE);
glTexImage2D(GL_TEXTURE_RECTANGLE_ARB,
            0, GL_RGBA, sizex, sizey, GL_BGRA,
            GL_UNSIGNED_INT_8_8_8_8_REV,
            myImagePtr);
```

Setting up a power-of-two texture to use these extensions is similar to what's needed to set up a rectangular texture, as you can see by looking at Listing 9-2. The difference is that the `GL_TEXTURE_2D` texture target replaces the `GL_TEXTURE_RECTANGLE_ARB` texture target.

Listing 9-2 Using texture extensions for a power-of-two texture

```
glBindTexture(GL_TEXTURE_2D, myTextureName);

glTexParameteri(GL_TEXTURE_2D,
        GL_TEXTURE_STORAGE_HINT_APPLE,
        GL_STORAGE_CACHED_APPLE);

glPixelStorei(GL_UNPACK_CLIENT_STORAGE_APPLE, GL_TRUE);
```

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA,
             sizex, sizey, GL_BGRA,
             GL_UNSIGNED_INT_8_8_8_8_REV, myImagePtr);
```

Optimal Data Formats and Types

The best format and data type combinations to use for texture data are:

`GL_BGRA`, `GL_UNSIGNED_INT_8_8_8_8_REV`

`GL_BGRA`, `GL_UNSIGNED_SHORT_1_5_5_5_REV`)

`GL_YCBCR_422_APPLE`, `GL_UNSIGNED_SHORT_8_8_REV_APPLE`

The combination `GL_RGBA` and `GL_UNSIGNED_BYTE` needs to be swizzled by many cards when the data is loaded, so it's not recommended.

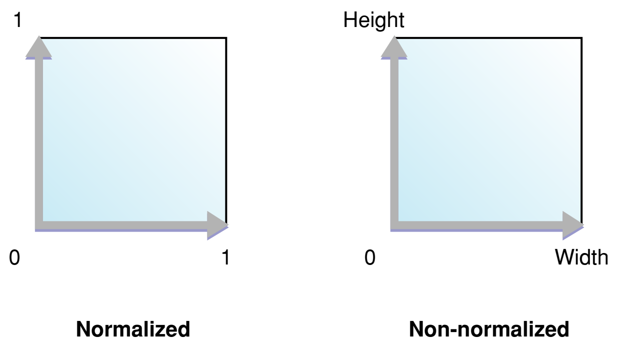
Working with Non–Power-of-Two Textures

With more and more frequency, OpenGL is being used to process video and images, which typically have dimensions that are not a power-of-two. Until OpenGL 2.0, the texture rectangle extension (`ARB_texture_rectangle`) provided the only option for a rectangular texture target. This extension, however, imposes the following restrictions on rectangular textures:

- You can't use mipmap filtering with them.
- You can use only these wrap modes: `GL_CLAMP`, `GL_CLAMP_TO_EDGE`, and `GL_CLAMP_TO_BORDER`.
- The texture cannot have a border.
- The texture uses non-normalized texture coordinates. (See Figure 9-7.)

OpenGL 2.0 adds another option for a rectangular texture target through the `ARB_texture_non_power_of_two` extension, which supports these textures without the limitations of the `ARB_texture_rectangle` extension. Before using it, you must check to make sure the functionality is available. You'll also want to consult the OpenGL specification for the non—power-of-two extension.

Figure 9-7 Normalized and non-normalized coordinates



If your code runs on a system that does not support either the `ARB_texture_rectangle` or `ARB_texture_non_power_of_two` extensions you have these options for working with with rectangular images:

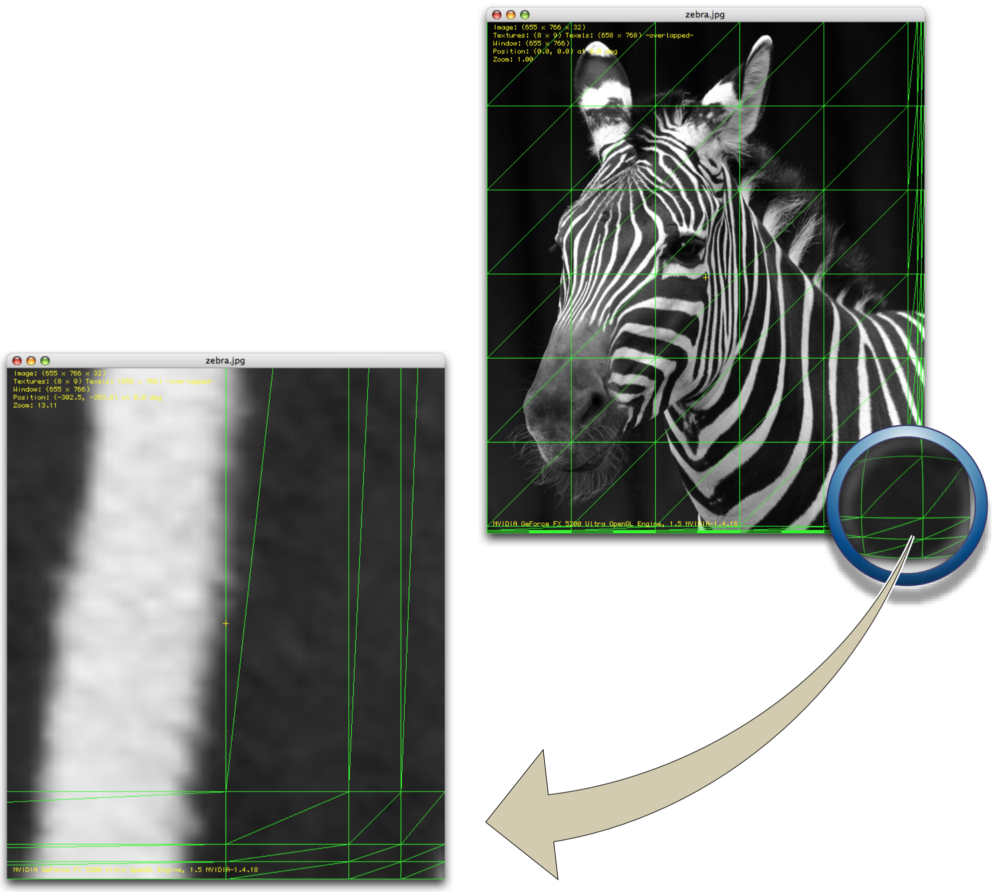
- Use the OpenGL function `gluScaleImage` to scale the image so that it fits in a rectangle whose dimensions are a power of two. The image undoes the scaling effect when you draw the image from the properly sized rectangle back into a polygon that has the correct aspect ratio for the image.

  > **Note:** This option can result in the loss of some data. But if your application runs on hardware that doesn't support the `ARB_texture_rectangle` extension, you may need to use this option.

- Segment the image into power-of-two rectangles, as shown in Figure 9-8 by using one image buffer and different texture pointers. Notice how the sides and corners of the image shown in Figure 9-8 are segmented into increasingly smaller rectangles to ensure that every rectangle has dimensions that are a power of two. Special care may be needed at the borders between each segment to avoid filtering artifacts if the texture is scaled or rotated.

  The *OpenGL Image* sample application available on Sample Code > Graphics & Imaging > OpenGL contains segmentation code and demonstrates other OpenGL features that support high-performance image display.

Figure 9-8 An image segmented into power-of-two tiles



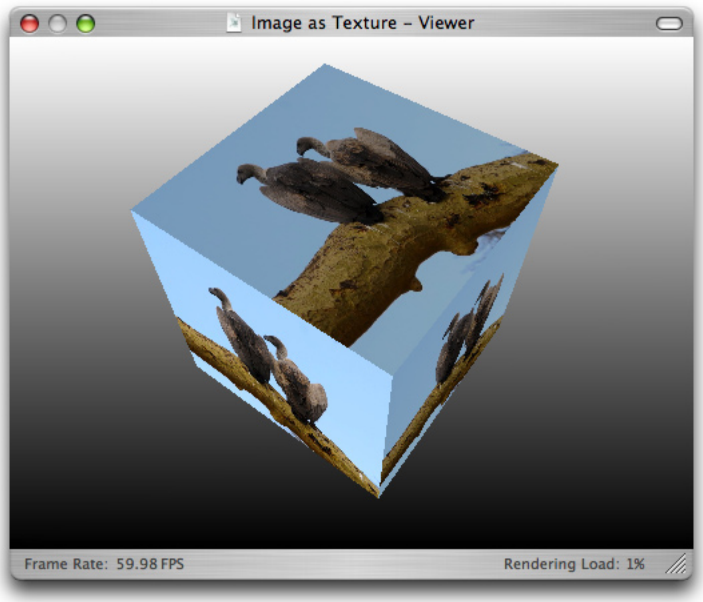
```
Image: (655 x 766 x 32)
Textures: (8 x 9) Texels: (658 x 768) -overlapped-
Window: (655 x 766)
Position: (0.0, 0.0) at 0.0 deg
Zoom: 1.00
NVIDIA GeForce FX 5200 Ultra OpenGL Engine, 1.5 NVIDIA-1.4.18
```

```
Image: (655 x 766 x 32)
Textures: (8 x 9) Texels: (658 x 768) -overlapped-
Window: (655 x 766)
Position: (-302.5, -353.8) at 0.0 deg
Zoom: 13.11
NVIDIA [illegible] Engine, 1.5 NVIDIA-1.4.18
```

Creating Textures from Image Data

OpenGL on the Macintosh provides several options for creating high-quality textures from image data. Mac OS X supports floating-point pixel values, multiple image file formats, and a variety of color spaces. You can import a floating-point image into a floating-point texture. Figure 9-9 shows an image used to texture a cube.

Figure 9-9 Using an image as a texture for a cube



For Cocoa, you need to provide a bitmap representation. You can create an `NSBitmapImageRep` object from the contents of an `NSView` object. For either Cocoa or Carbon, you can use the Image I/O framework (see *CGImageSource Reference*). This framework has support for many different file formats, floating-point data, and a variety of color spaces. Furthermore, it is easy to use. You can import image data as a texture simply by supplying a `CFURL` object that specifies the location of the texture. There is no need for you to convert the image to an intermediate integer RGB format.

Creating a Texture from a Cocoa View

You can use the `NSView` class or a subclass of it for texturing in OpenGL. The process is to first store the image data from an `NSView` object in an `NSBitmapImageRep` object so that the image data is in a format that can be readily used as texture data by OpenGL. Then, after setting up the texture target, you supply the bitmap data to the OpenGL function `glTexImage2D`. Note that you must have a valid, current OpenGL context set up.

Note: You can't create an OpenGL texture from image data that's provided by a view created from the following classes: `NSProgressIndicator`, `NSMovieView`, and `NSOpenGLView`. This is because these views do not use the window backing store, which is what the method `initWithFocusedViewRect:` reads from.

Listing 9-3 shows a routine that uses this process to create a texture from the contents of an `NSView` object. A detailed explanation for each numbered line of code appears following the listing.

Listing 9-3 Building an OpenGL texture from an `NSView` object

```
-(void)myTextureFromView:(NSView*)theView
                textureName:(GLuint*)texName
{
    NSBitmapImageRep * bitmap = [NSBitmapImageRep alloc];                   // 1
    int samplesPerPixel = 0;

    [theView lockFocus];                                                    // 2
    [bitmap initWithFocusedViewRect:[theView bounds]];                      // 3
    [theView unlockFocus];
    glPixelStorei(GL_UNPACK_ROW_LENGTH, [bitmap pixelsWide]);               // 4
    glPixelStorei (GL_UNPACK_ALIGNMENT, 1);                                 // 5
    if (*texName == 0)                                                      // 6
            glGenTextures (1, texName);
    glBindTexture (GL_TEXTURE_RECTANGLE_ARB, *texName);                     // 7
    glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
                    GL_TEXTURE_MIN_FILTER, GL_LINEAR);                      // 8
    samplesPerPixel = [bitmap samplesPerPixel];                             // 9

   if(![bitmap isPlanar] &&
       (samplesPerPixel == 3 || samplesPerPixel == 4)) {                    // 10
        glTexImage2D(GL_TEXTURE_RECTANGLE_ARB,
                     0,
                     samplesPerPixel == 4 ? GL_RGBA8 : GL_RGB8,
                     [bitmap pixelsWide],
                     [bitmap pixelsHigh],
                     0,
                     samplesPerPixel == 4 ? GL_RGBA : GL_RGB,
                     GL_UNSIGNED_BYTE,
                    [bitmap bitmapData]);
    } else {
       // Your code to report unsupported bitmap data
    }
   [bitmap release];                                                        // 11
}
```

Here's what the code does:

1. Allocates an `NSBitmapImageRep` object.

2. Locks the focus on the the `NSView` object so that subsequent commands take effect in coordinate system of the `NSView` object. You must invoke `lockFocus` before invoking methods that send commands to the window server, which is the case with the next line of code. Later, you must balance a `lockFocus` message with an `unlockFocus` message.

3. Initializes the `NSBitmapImageRep` object with bitmap data from the current view using the bounds returned by the `NSView` object passed to the `myTextureFromView:textureName` routine.

4. Sets the appropriate unpacking row length for the bitmap.
5. Sets the byte-aligned unpacking that's needed for bitmaps that are 3 bytes per pixel.
6. If a texture object is not passed in, generates a new texture object.
7. Binds the texture name to the texture target.
8. Sets filtering so that it does not use a mipmap, which would be redundant for the texture rectangle extension.
9. Gets the number of samples per pixel.
10. Checks to see if the bitmap is nonplanar and is either a 24-bit RGB bitmap or a 32-bit RGBA bitmap. If so, retrieves the pixel data using the `bitmapData` method, passing it along with other appropriate parameters to the OpenGL function for specifying a 2D texture image.
11. Releases the `NSBitmapImageRep` object when it is no longer needed.

Creating a Texture from a Quartz Image Source

Quartz images (`CGImageRef` data type) are defined in the Core Graphics framework (`ApplicationServices/CoreGraphics.framework/CGImage.h`) while the image source data type for reading image data and creating Quartz images from an image source is declared in the Image I/O framework (`ApplicationServices/ImageIO.framework/CGImageSource.h`). Quartz provides routines that read a wide variety of image data.

To use a Quartz image as a texture source, follow these steps:

1. Create a Quartz image source by supplying a `CFURL` object to the function `CGImageSourceCreateWithURL`.
2. Create a Quartz image by extracting an image from the image source, using the function `CGImageSourceCreateImageAtIndex`.
3. Extract the image dimensions using the function `CGImageGetWidth` and `CGImageGetHeight`. You'll need these to calculate the storage required for the texture.
4. Allocate storage for the texture.
5. Create a color space for the image data.
6. Create a Quartz bitmap graphics context for drawing. Make sure to set up the context for pre-multiplied alpha.
7. Draw the image to the bitmap context.
8. Release the bitmap context.
9. Set the pixel storage mode by calling the function `glPixelStorei`.
10. Create and bind the texture.
11. Set up the appropriate texture parameters.

12. Call `glTexImage2D`, supplying the image data.

13. Free the image data.

Listing 9-4 shows a code fragment that performs these steps. Note that you must have a valid, current OpenGL context.

Listing 9-4 Using a Quartz image as a texture source

```
CGImageSourceRef myImageSourceRef = CGImageSourceCreateWithURL(url, NULL);
CGImageRef myImageRef = CGImageSourceCreateImageAtIndex (myImageSourceRef, 0,
NULL);
GLint myTextureName;
size_t width = CGImageGetWidth(myImageRef);
size_t height = CGImageGetHeight(myImageRef);
CGRect rect = {{0, 0}, {width, height}};
void * myData = calloc(width * 4, height);
CGColorSpaceRef space = CGColorSpaceCreateDeviceRGB();
CGContextRef myBitmapContext = CGBitmapContextCreate (myData,
                        width, height, 8,
                        width*4, space,
                        kCGBitmapByteOrder32Host |
                          kCGImageAlphaPremultipliedFirst);
CGContextDrawImage(myBitmapContext, rect, myImageRef);
CGContextRelease(myBitmapContext);
glPixelStorei(GL_UNPACK_ROW_LENGTH, width);
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glGenTextures(1, &myTextureName);
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, myTextureName);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
                    GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, GL_RGBA8, width, height,
                    0, GL_BGRA_EXT, GL_UNSIGNED_INT_8_8_8_8_REV, myData);
free(myData);
```

For more information on using Quartz, see *Quartz 2D Programming Guide*, *CGImage Reference*, and *CGImageSource Reference*.

Getting Decompressed Raw Pixel Data from a Source Image

You can use the Image I/O framework together with a Quartz data provider to obtain decompressed raw pixel data from a source image, as shown in Listing 9-5. You can then use the pixel data for your OpenGL texture. The data has the same format as the source image, so you need to make sure that you use a source image that has the layout you need.

Alpha is not premultiplied for the pixel data obtained in Listing 9-5, but alpha is premultiplied for the pixel data you get when using the code described in "Creating a Texture from a Cocoa View" (page 103) and "Creating a Texture from a Quartz Image Source" (page 105).

Listing 9-5 Getting pixel data from a source image

```
CGImageSourceRef myImageSourceRef = CGImageSourceCreateWithURL(url, NULL);
CGImageRef myImageRef = CGImageSourceCreateImageAtIndex (myImageSourceRef, 0,
NULL);
```

```
CFDataRef data = CGDataProviderCopyData(CGImageGetDataProvider(myImageRef));
void *pixelData = CFDataGetBytePtr(data);
```

Downloading Texture Data

A texture download operation uses the same data path as an upload operation except that the data path is reversed. Downloading transfers texture data, using direct memory access (DMA), from VRAM into a texture that can then be accessed directly by your application. You can use the Apple client range, texture range, and texture rectangle extensions for downloading, just as you would for uploading.

To download texture data using the Apple client storage, texture range, and texture rectangle extensions:

- Bind a texture name to a texture target.
- Set up the extensions
- Call the function `glCopyTexSubImage2D` to copy a texture subimage from the specified window coordinates. This call initiates an asynchronous DMA transfer to system memory the next time you call a flush routine. The CPU doesn't wait for this call to complete.
- Call the function `glGetTexImage` to transfer the texture into system memory. Note that the parameters must match the ones that you used to set up the texture when you called the function `glTexImage2D`. This call is the synchronization point; it waits until the transfer is finished.

Listing 9-6 shows a code fragment that downloads a rectangular texture that uses cached memory. Your application processes data between the `glCopyTexSubImage2D` and `glGetTexImage` calls. How much processing? Enough so that your application does not need to wait for the GPU.

Listing 9-6 Code that downloads texture data

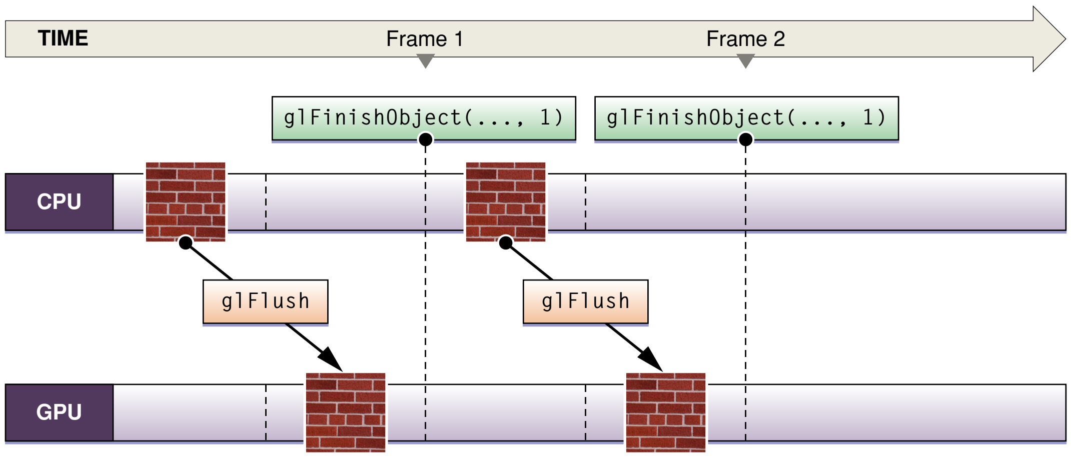
```
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, myTextureName);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_STORAGE_HINT_APPLE,
                GL_STORAGE_SHARED_APPLE);
glPixelStorei(GL_UNPACK_CLIENT_STORAGE_APPLE, GL_TRUE);
glTexImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, GL_RGBA,
                sizex, sizey, GL_BGRA,
                GL_UNSIGNED_INT_8_8_8_8_REV, myImagePtr);

glCopyTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB,
                0, 0, 0, 0, 0, image_width, image_height);
glFlush();
// Do other work processing here, using a double or triple buffer

glGetTexImage(GL_TEXTURE_RECTANGLE_ARB, 0, GL_BGRA,
                GL_UNSIGNED_INT_8_8_8_8_REV, pixels);
```

Double Buffering Texture Data

When you use any technique that allows the GPU to access your texture data directly, such as the texture range extension, it's possible for the GPU and CPU to access the data at the same time. To avoid such a collision, you must synchronize the GPU and the CPU. The simplest way is shown in Figure 9-10. Your application works on the data, flushes it to the GPU and waits until the GPU is finished before working on the data again.

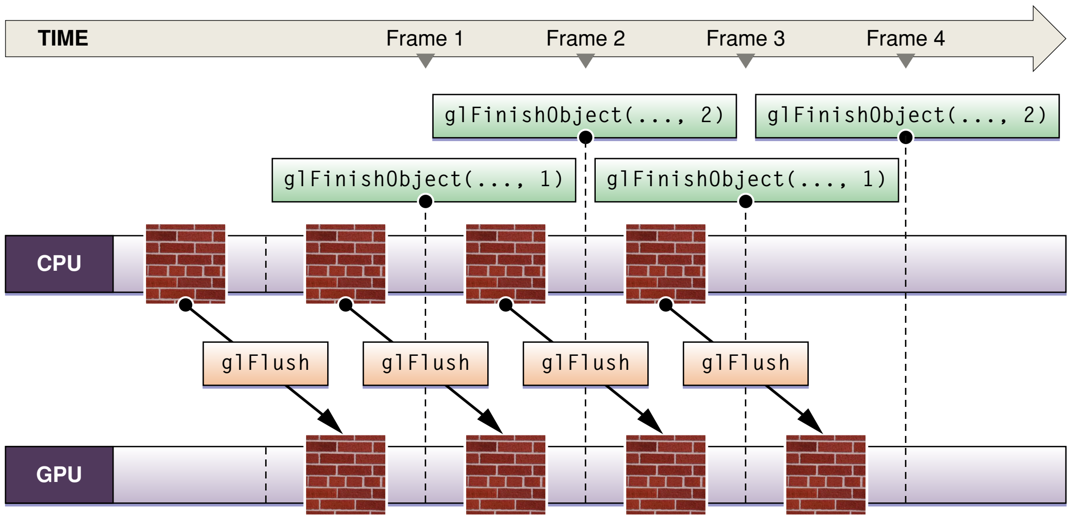
One technique for ensuring that the GPU is finished executing commands before your application sends more data is to insert a token into the command stream and use that to determine when the CPU can touch the data again, as described in "Fence Extension" (page 92). Figure 9-10 uses the fence extension command `glFinishObject` to synchronize buffer updates for a stream of single-buffered texture data. Notice that when the CPU is processing texture data, the GPU is idle. Similarly, when the GPU is processing texture data, the CPU is idle. It's much more efficient for the GPU and CPU to work asynchronously than to work synchronously. Double buffering data is a technique that allows you to process data asynchronously, as shown in Figure 9-11 (page 109).

Figure 9-10 Single-buffered data



To double buffer data, you must supply two sets of data to work on. Note in Figure 9-11 that while the GPU is rendering one frame of data, the CPU processes the next. After the initial startup, neither processing unit is idle. Using the `glFinishObject` function provided by the fence extension ensures that buffer updating is synchronized.

Figure 9-11 Double-buffered data



See Also

OpenGL extension specifications:

- APPLE_client_storage
- APPLE_texture_range
- ARB_texture_rectangle
- ARB_texture_non_power_of_two

OpenGL sample code projects (Sample Code > Graphics & Imaging > OpenGL):

- *OpenGL Image* segments a rectangular image into several power-of-two textures and shows how to use OpenGL for high performance image display.
- *Quartz Composer Texture* shows how to use the `QCRenderer` class to render a Quartz Composer composition into an OpenGL pixel buffer, create a texture from it, and use the texture in an OpenGL scene.
- *TexturePerformanceDemo* provides code that uploads textures using two different ways, one of which is more optimized than the other.
- *TextureRange* shows how to use various OpenGL extensions to optimize uploading texture data.
- *NSGLImage* demonstrates how to use the `NSImage` and `NSBitmapImageRep` classes for texturing.

More information on the Quartz API and how to use Quartz:

- *CGImageSource Reference* describes the `CGImageSourceRef` data type and the functions that operate on it.
- *CGColorSpace Reference* describes the `CGColorSpaceRef` data type and the functions that operate on it.

- *Quartz 2D Programming Guide* describes how to write code that uses all the Quartz data types, including the `CGImageSourceRef` and `CGColorSpaceRef` data types.

Techniques for Scene Anti-Aliasing

Aliasing is the bane of the digital domain. In the early days of the personal computer, jagged edges and blocky graphics were accepted by the user simply because not much could be done to correct them. Now that hardware is faster and displays are higher in resolution, there are several anti-aliasing techniques that can smooth edges to achieve a more realistic scene.

OpenGL supports anti-aliasing that operates at the level of lines and polygons as well as at the level of the full scene. This chapter discusses techniques for full scene anti-aliasing (FSAA). The three anti-aliasing techniques in use today are multisampling, supersampling, and alpha channel blending:

- **Multisampling** defines a technique for sampling pixel content at multiple locations for each pixel. This is a good technique to use if you want to smooth polygon edges.
- **Supersampling** renders at a much higher resolution than what's needed for the display. Prior to drawing the content to the display, OpenGL scales and filters the content to the appropriate resolution. This is a good technique to use when you want to smooth texture interiors in addition to polygon edges.
- **Alpha channel blending** uses the alpha value of a fragment to control how to blend the fragment with the pixel values that are already in the framebuffer. It's a good technique to use when you want to ensure that foreground and background images are composited smoothly.

The `ARB_multisample` extension defines a specification for full scene anti-aliasing. It describes multisampling and alpha channel sampling. The specification does not specifically mention supersampling but its wording doesn't preclude supersampling. The anti-aliasing methods that are available depend on the hardware and the actual implementation depends on the vendor. Some graphics cards support anti-aliasing using a mixture of multisampling and supersampling. The methodology used to select the samples can vary as well. Your best approach is to query the renderer to find out exactly what is supported. OpenGL lets you provide a hint to the renderer as to which anti-aliasing technique you prefer. Hints are available starting in Mac OS X v10.4 as renderer attributes that you supply when you create a pixel format object.

Guidelines

You'll want to keep the following in mind when you set up full scene anti-aliasing:

- Although a system may have enough VRAM to accommodate a multisample buffer, a large buffer can affect the ability of OpenGL to maintain a properly working texture set. Keep in mind that the buffers associated with the rendering context—depth and stencil—increase in size by a factor equal to number of samples per pixel.
- The OpenGL driver allocates the memory needed for the multisample buffer; your application should not allocate this memory.
- Any anti-aliasing algorithm that operates on the full scene requires a fair amount of computing resources. In some cases, there is a tradeoff between performance and quality. For that reason, developers sometimes provide a user interface element that allows the user to enable and disable FSAA, or to choose the level of quality for anti-aliasing.

- The commands `glEnable(GL_MULTISAMPLE)` and `glDisable(GL_MULTISAMPLE)` are ignored on some hardware because some graphics cards have the feature enabled all the time. That doesn't mean that you should not call these commands because you'll certainly need them on hardware that doesn't ignore them.
- A hint as to the variant of sampling you want is a suggestion, not a command. Not all hardware supports all types of anti-aliasing. Other hardware mixes multisampling with supersampling techniques. The driver dictates the type of anti-aliasing that's actually used in your application.
- The best way to find out which sample modes are supported is to call the CGL function `CGLDescribeRenderer` with the renderer property `kCGLRPSampleModes` or `kCGLRPSampleAlpha`.

General Approach

The general approach to setting up full scene anti-aliasing is as follows:

1. Check to see what's supported. Not all hardware is capable of supporting the ARB multisample extension, so you need to check for this functionality (see "Detecting Functionality" (page 59)).

   To find out what type of anti-aliasing a specific renderer supports, call the function `CGLDescribeRenderer`. Supply the renderer property `kCGLRPSampleModes` to find out whether the renderer supports multisampling and supersampling. Supply `kCGLRPSampleAlpha` to see whether the renderer supports alpha sampling.

   You can choose to exclude unsupported hardware from the pixel format search by specifying only the hardware that supports multisample anti-aliasing. Keep in mind that if you exclude unsupported hardware, the unsupported displays will not render anything. If you instead choose to include unsupported hardware, OpenGL uses normal aliased rendering to the unsupported displays and multisampled rendering to supported displays.

2. Include these buffer attributes in the attributes array:

   - The appropriate sample buffer attribute constant (`NSOpenGLPFASampleBuffers`, `AGL_SAMPLE_BUFFERS_ARB`, or `kCGLPFASampleBuffers`) along with the number of multisample buffers. At this time the specification allows only one multisample buffer.
   - The appropriate samples constant (,`NSOpenGLPFASamples`, `AGL_SAMPLES_ARB`, or `kCGLPFASamples`) along with the number of samples per pixel. You can supply 2, 4, 6, or more depending on what the renderer supports and the amount of VRAM available. The value that you supply affects the quality, memory use, and speed of the multisampling operation. For fastest performance, and to use the least amount of video memory, specify 2 samples. When you need more quality, specify 4 or more.
   - The no recovery attribute (`NSOpenGLPFANoRecovery`, `AGL_NO_RECOVERY`, or `kCGLPFANoRecovery`). Although enabling this attribute is not mandatory, it's recommended to prevent OpenGL from using software fallback as a renderer. The software renderer does not support multisample antialiasing prior to Mac OS X v10.4. In versions that the software renderer does support multisampling (4, 9, or 16 samples), antialiasing performance is slow.

3. Optionally provide a hint for the type of anti-aliasing you want—multisampling, supersampling, or alpha sampling. See "Hinting for a Specific Anti-Aliasing Technique" (page 113).

4. Enable multisampling with the following command:

```
glEnable(GL_MULTISAMPLE);
```

Regardless of the enabled state, OpenGL always uses the multisample buffer if you supply the appropriate buffer attributes when you set up the pixel format object. If you haven't supplied the appropriate attributes, enabling multisampling has no effect.

When multisampling is disabled, all coverage values are set to `1`, which gives the appearance of rendering without multisampling.

Some graphics hardware leaves multisampling enabled all the time. However, don't rely on hardware to have multisampling enabled; use `glEnable` to programmatically turn on this feature.

5. Optionally provide hints for the rendering algorithm. You perform this optional step only if you want OpenGL to compute coverage values by a method other than uniformly weighting samples and averaging them.

   Some hardware supports a multisample filter hint through an OpenGL extension—`GL_NV_multisample_filter_hint`. This hint allows an OpenGL implementation to use an alternative method of resolving the color of multisampled pixels.

   You can specify that OpenGL uses faster or nicer rendering by calling the OpenGL function `glHint`, passing the constant `GL_MULTISAMPLE_FILTER_HINT_NV` as the target parameter and `GL_FASTEST` or `GL_NICEST` as the mode parameter. Hints allow the hardware to optimize the output if it can. There is no performance penalty or returned error for issuing a hint that's not supported.

   For more information, see the OpenGL extension registry for NV_multisample_filter_hint.

"Setting Up Full Scene Anti-Aliasing" (page 114) provides specific code examples.

Hinting for a Specific Anti-Aliasing Technique

In Mac OS X v10.4 and later, when you set up your renderer and buffer attributes for full scene antialiasing, you can specify a hint to prefer one anti-aliasing technique over the others. If the underlying renderer does not have sufficient resources to support what you request, OpenGL ignores the hint. If you do not supply the appropriate buffer attributes when you create a pixel format object, then the hint does nothing. Table 10-1 lists the hinting constants available for the `NSOpenGLPixelFormat` class, AGL, and CGL.

Table 10-1 Anti-aliasing hints available starting in Mac OS X v10.4

Multisampling	Supersampling	Alpha blending
`NSOpenGLPFAMultisample`	`NSOpenGLPFASupersample`	`NSOpenGLPFASampleAlpha`
`AGL_MULTISAMPLE`	`AGL_SUPERSAMPLE`	`AGL_SAMPLE_ALPHA`
`kCGLPFAMultisample`	`kCGLPFASupersample`	`kCGLPFASampleAlpha`

Setting Up Full Scene Anti-Aliasing

The code listings in this section show how to set up full scene anti-aliasing using the `NSOpenGLPixelFormat` class, AGL, and CGL. You'll see that the code to set buffer and renderer attributes and to create a context looks similar to what you'd normally use to set up any rendering context. Regardless of the API that you use, you need to specify the appropriate attributes. Although you need to specify the context slightly differently for each of the APIs, the outcome is the same—a pixel format and context that supports full-scene anti-aliased rendering.

Listing 10-1 sets up full scene anti-aliasing using the `NSOpenGLPixelFormat` class, but does not provide a hint, which is optional. A detailed explanation for each numbered line of code appears following the listing.

Listing 10-1 Using `NSOpenGLPixelFormat` to set up full scene anti-aliasing

```
#import <Cocoa/Cocoa.h>
@implementation BasicOpenGLView
+ (NSOpenGLPixelFormat*)defaultPixelFormat
{
    NSOpenGLPixelFormatAttribute attributes [] = {                    // 1
            NSOpenGLPFAWindow,
            NSOpenGLPFADoubleBuffer,
            NSOpenGLPFASampleBuffers, 1,
            NSOpenGLPFASamples, 2,
            NSOpenGLPFANoRecovery,
            (NSOpenGLPixelFormatAttribute)nil
        };
    return [[[NSOpenGLPixelFormat alloc]
            initWithAttributes:attributes] autorelease];              // 2
}
-(id) initWithFrame: (NSRect) frameRect
{
    NSOpenGLPixelFormat *pixelFormat = [BasicOpenGLView
                                defaultPixelFormat];                  // 3
    self = [super initWithFrame: frameRect
                pixelFormat: pixelFormat];                            // 4
   return self;
}
// Define  other class methods  here.
@end
```

Here's what the code in Listing 10-1 does:

1. Sets up attributes for OpenGL to use for choosing the pixel format. The attributes include the two required for multisampling: `NSOpenGLPFASampleBuffers` and `NSOpenGLPFASamples`, along with those to support a Cocoa window, double buffering, and no recovery.
2. Allocates and initializes an `NSOpenGLPixelFormat` object with the requested multisampling (and other) attributes.
3. Creates an `NSOpenGLPixelFormat` object for a custom `NSOpenGLView` class (`BasicOpenGLView`) that was previously created by the application using Interface Builder but is not shown in this example.
4. Initializes the view with the newly created pixel format.

Listing 10-2 sets up full scene anti-aliasing in Carbon and provides a hint for supersampling. A detailed explanation for each numbered line of code appears following the listing.

Listing 10-2 Using AGL to set up full scene anti-aliasing with a hint for supersampling

```
#include <AGL/agl.h>
GLint attribs[] = { AGL_RGBA,                                              // 1
                    AGL_DOUBLEBUFFER,
                    AGL_SAMPLE_BUFFERS_ARB, 1,
                    AGL_SAMPLES_ARB, 2,
                    AGL_SUPERSAMPLE,
                    AGL_NO_RECOVERY,
                    AGL_NONE };

AGLPixelFormat pixelFormat = NULL;
AGLContext context = NULL;
pixelFormat = aglChoosePixelFormat (NULL, 0, attribs);                     // 2
if (pixelFormat) {
    context = aglCreateContext (pixelFormat, NULL);                        // 3
    aglDestroyPixelFormat (pixelFormat);
}
```

Here's what the code in Listing 10-2 does:

1. Sets up attributes for OpenGL to use for creating an AGL pixel format object. The attributes include the two required for multisampling (`AGL_SAMPLE_BUFFERS_ARB` and `AGL_SAMPLES_ARB`) along with those to support RGBA pixels, double buffering, and no recovery.
2. Creates a pixel format object. Prior to this call you can optionally provide a list of `GDHandle` values that specify the supported displays.
3. Creates a rendering context based on the newly created pixel format object that is set up to support full scene antialiasing.

Listing 10-3 sets up full scene anti-aliasing using the CGL API and provides a hint for multisampling. A detailed explanation for each numbered line of code appears following the listing.

Listing 10-3 Using CGL to set up full scene anti-aliasing with a hint for multisampling

```
#include <OpenGL/OpenGL.h>
CGLPixelFormatAttribute attribs[] = {   kCGLPFADisplayMask, 0,             // 1
                                        kCGLPFAFullScreen,
                                        kCGLPFADoubleBuffer,
                                        kCGLPFASampleBuffers, 1,
                                        kCGLPFASamples, 2,
                                        kCGLPFAMultisample
                                        kCGLPFANoRecovery,
                                        0 };

CGLPixelFormatObj pixelFormat = NULL;
CGLContextObj context = NULL;
long numPixelFormats = 0;
attribs[1] = CGDisplayIDToOpenGLDisplayMask (CGMainDisplayID ());          // 2
CGLChoosePixelFormat (attribs, &pixelFormat, &numPixelFormats));           // 3
if (pixelFormat) {
        CGLCreateContext (pixelFormat, NULL, &context);                    // 4
        CGLDestroyPixelFormat (pixelFormat);
}
```

Here's what the code in Listing 10-3 does:

1. Sets up attributes for OpenGL to use for creating a CGL pixel format object. The attributes include the two multisampling attributes: `kCGLPFASampleBuffers` and `kCGLPFASamples`, along with those to support full-screen drawing, double buffering, and no recovery. The associated value for `kCGLPFASamples` is the number of samples per multisample buffer, which in this case is `2`. The associated value for `kCGLPFASampleBuffers` is a nonnegative integer that indicates the number of existing independent sample buffers, which in this case is `1`.

2. Sets the value of the display mask attribute to the main display. (Note that this code example does not capture the main display. See Listing 3-3 (page 41).)

3. Creates a pixel format object with the requested attributes.

4. Creates a context for the pixel format that isn't shared.

See Also

You can find the complete specification for the `GL_ARB_multisample` extension in the OpenGL extensions registry at http://oss.sgi.com/projects/ogl-sample/registry/.

If your application needs point or line anti-aliasing instead of full scene anti-aliasing, use the built in OpenGL point and line anti-aliasing functions. These are described in Section 3.4.2 in the OpenGL Specification.

Multithreading and OpenGL

Each process in Mac OS X is made up of one or more threads. A **thread** is a stream of execution that runs code for the process. You can improve application performance and enhance the perceived responsiveness of the user interface when you set up your application to use multiple threads. On computers with one processor, multithreading can allow a program to execute multiple pieces of code independently. On computers with more than one processor, multithreading can allow a program to execute multiple pieces of code simultaneously.

Multithreading, however, is not the solution for all performance issues. When it is a possible solution, it enhances performance only when it's set up correctly. Getting multithreading to work properly in an OpenGL application requires advanced programming techniques—the OpenGL API is not inherently thread-safe. If you want to make your OpenGL program multithreaded, read this chapter to get started, then roll up your sleeves. Be prepared to undertake a lot of detective work if things go wrong. In threaded applications, the cause of the problem is often difficult to isolate.

Program Design

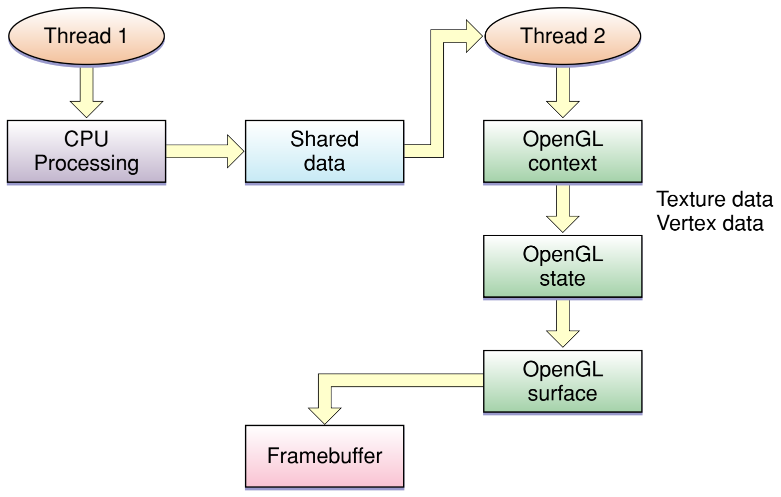
You'll have the best chance of success with multithreading if you design your program with threading in mind. It's difficult, and often risky, to retrofit an existing OpenGL application to use multiple threads. Before you write any threading code, choose a strategy for dividing work among threads.

Consider using one of the following strategies for your OpenGL application:

- Move OpenGL onto a separate thread.
- Split OpenGL texture and vertex processing onto separate threads. You gain performance advantages by applying threads on single processor machines but threads are most efficient on computers with multiple CPUs since each processor can devote itself to a thread, potentially doubling the throughput.
- For contexts on separate threads, share surfaces or OpenGL object state: display lists, textures, vertex and fragment programs, vertex array objects, and so on.

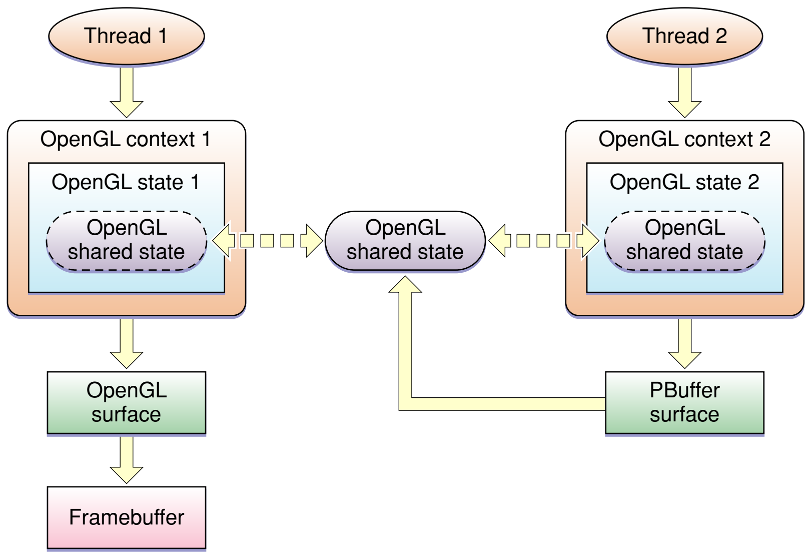
Applications that move OpenGL onto a separate thread are designed as shown in Figure 11-1. The CPU writes its data to a shared space, accessible to OpenGL. This design provides a clear division of labor and is fairly straightforward to implement. You can use this design to load data into your application on one thread, and then draw with the data on the other thread.

Figure 11-1 CPU processing and OpenGL on separate threads



The Apple-specific OpenGL APIs provide the option for sharing data between contexts. You can leverage this feature in a threaded application by creating a separate thread for each of the contexts that share data, as shown in Figure 11-2. Shared resources are automatically set up as mutual exclusion (**mutex**) objects. Notice that Thread 2 draws to a pixel buffer that is linked to the shared state as a texture. Thread 1 can then draw using that texture.

Figure 11-2 Two contexts on separate threads



Guidelines for Threading OpenGL Applications

Follow these guidelines to ensure successful threading in an application that uses OpenGL:

- Use only one thread per context. OpenGL commands for a specific context are not reentrant. You should never have more than one thread accessing a single context simultaneously.

If for some reason you decide to set more than one thread to target the same context, then you must synchronize threads by placing a mutex around all OpenGL calls to the context, such as `gl*` and `CGL*`. You can use one of the APIs listed in "Threading APIs" (page 119) to set up a mutex. OpenGL commands that block—such as `fence` commands—do not synchronize threads.

- Contexts that are on different threads can share object resources. For example, it is acceptable for one context in one thread to modify a texture and a second context in a second thread to modify the same texture. Why? Because the shared object handling provided by the Apple APIs automatically protects against thread errors. And, your application is following the "one thread per context" guideline.
- When you use an `NSOpenGLView` object with OpenGL calls that are issued from a thread other than the main one, you must set up mutex locking. Why? Unless you override the default behavior, the main thread may need to communicate with the view for such things as resizing.

  Applications that use Objective-C with multithreading can lock contexts using the functions `CGLLockContext` and `CGLUnlockContext`, which were introduced in Mac OS X v10.4. If you want to perform rendering in a thread other than the main one, you can lock the context that you want to access and safely execute OpenGL commands. The locking calls must be placed around all of your OpenGL calls in all threads. You can't set up your own mutex in versions of Mac OS X earlier than v10.4.

  `CGLLockContext` blocks the thread it is on until all other threads have unlocked the same context using the function `CGLUnlockContext`. You can use `CGLLockContext` recursively. Context-specific CGL calls by themselves do not require locking, but you can guarantee serial processing for a group of calls by surrounding them with `CGLLockContext` and `CGLUnlockContext`. Keep in mind that calls from the OpenGL API (the API provided by the Architecture Review Board) require locking.
- Keep track of the current context. When switching threads it is easy to switch contexts inadvertently, which causes unforeseen effects on the execution of graphic commands. You must set a current context when switching to a newly created thread.

Note: The guidelines in this section are specific to OpenGL applications. Any threading code that you write also needs to comply with general threading practices. You can find general resources for thread programming in the "See Also" (page 120) section.

When Things Go Wrong

If you don't set up threading correctly, you'll most likely see your application freeze or crash. Things typically go wrong when your application introduces a command to the graphics processor that violates threading practices. The bad command will cause the processor to hang. The CPU blocks against that, causing any drawing onscreen to stop and the spinning wait cursor to appear.

You can use OpenGL Profiler to check thread safety in OpenGL. In the breakpoints window, set the "Break on thread error" option to check whether a problem is due to a thread error.

Threading APIs

The following APIs are available for creating threaded applications in Mac OS X:

- Foundation provides threading support for Cocoa application through the `NSThread` class.

- UNIX provides POSIX threads to support threading for any layer in Mac OS X.
- Carbon provides thread support through the Multiprocessing Services API.

See Also

The OpenGL sample code project *Vertex Optimization* (available from Sample Code > Graphics & Imaging > OpenGL) has an option to run as a multithreaded application.

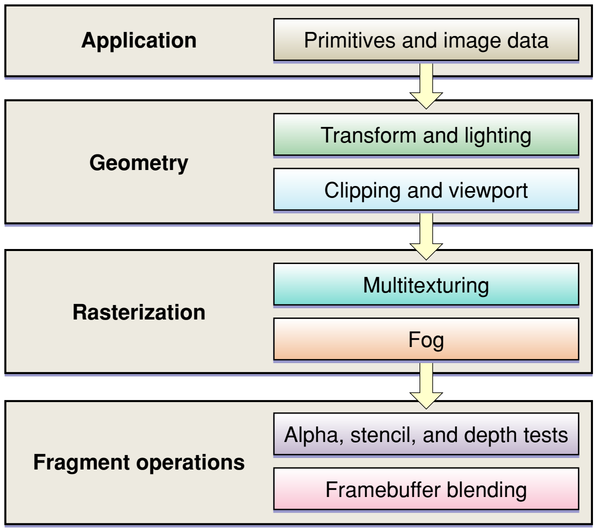
Multithreading programming guides and reference documentation:

- *Threading Programming Guide* explains how to use threads in Cocoa applications.
- *NSThread Class Reference* describes the Foundation threading class and its methods.
- *Multiprocessing Services Programming Guide* explains how to implement preemptive tasks in Carbon applications.
- *Multiprocessing Services Reference* describes the C API for creating preemptively scheduled tasks in Carbon applications.
- "Debugging programs with multiple threads" in the "Running Programs Under GDB" chapter of *Debugging with GDB* provides useful information for any multithreaded application.

Improving Performance

OpenGL performs many complex operations—transformations, lighting, clipping, texturing, environmental effects, and so on—on large data sets. The amount of data and the number of operations can impact performance, making your stellar 3D graphics shine less brightly than you'd like. Unless, of course, you take steps to streamline your application.

Figure 12-1 OpenGL performs complex operations as data flows through a program



Techniques for improving data throughput and increasing program efficiency are discussed throughout this book. This chapter provides additional performance guidelines and discusses some of the tools that you can use to analyze your application.

- "Best Practices" (page 121) summarizes coding tips that can help achieve optimal performance and provides links to other sections, either in this chapter or elsewhere in the book, where you can read more details.
- "Gathering and Analyzing Baseline Performance Data" (page 128) shows how to use `top` and OpenGL Profiler to obtain and interpret baseline performance data.
- "Identifying Bottlenecks with Shark" (page 133) discusses the patterns of usage that the Shark performance tool can make apparent and that indicate places in your code that you may want to improve.

Best Practices

Each of the following sections provides information that can help your application perform optimally:

- "Use Flush and Finish Routines Effectively" (page 122).
- "Be Mindful of OpenGL State Variables" (page 123).
- "Draw Only When Necessary" (page 124).
- "Synchronize with the Screen Refresh Rate" (page 125).
- "Use OpenGL Macros" (page 125).
- "Use the CPU and the GPU Asynchronously" (page 126).
- Adopt "Techniques for Working with Vertex Data" (page 85) and "Techniques for Working with Texture Data" (page 95) to optimize data throughput and decrease memory footprint.
- "Use Appropriate Routines for Images and Pixel Data" (page 127).
- Use threading appropriately. See "Multithreading and OpenGL" (page 117).
- Use performance tools to assess your application and identify areas that can be optimized. See "Gathering and Analyzing Baseline Performance Data" (page 128) and "Identifying Bottlenecks with Shark" (page 133).
- "Retrieve Error Information Only When Debugging" (page 127).
- "Use Optimal Data Types and Formats" (page 127)

Use Flush and Finish Routines Effectively

OpenGL commands are not executed immediately. They are queued to a command buffer and then read and executed by the hardware. The command buffer is used for vertices, normals, texture coordinates, and so forth, but not for textures themselves, which are stored elsewhere.

These functions force OpenGL to submit the command buffer to the hardware for execution.

- The function `glFlush` waits until commands are submitted but does not wait for the commands to finish executing.
- The function `glFinish` waits for the submitted commands to complete executing.

For double-buffered contexts, the current OpenGL command buffer is not sent to the graphics processor until `glFlush` or `glFinish` is called, a buffer swap is requested, or the command buffer is full. This also applies to single-buffered contexts, although executing a buffer swap is really just an implicit call to `glFlush` to submit the queued commands to the renderer. This means that, for single-buffered contexts, `glFlush` and `glFinish` are equivalent to a swap operation, since all rendering is taking place directly in the front buffer.

There are only a few cases that require you to call the `glFlush` function:

- Multithreaded applications. To keep drawing synchronized across the threads and prevent command buffer corruption, as each thread completes its command submissions, it should call `glFlush`.
- A drawable object that changes during rendering. Before you can switch from one drawable object to another, you must call `glFlush` to ensure that all commands written in the command queue for the previous drawable object have been submitted.

- Whenever a drawable surface in modified by one context and then used in another context, you must call `glFlush` to ensure the data is serialized to the video card before you attempt to draw with it on the second context. Conversely, if you have drawn with surface that is about to get modified you should flush the context that draws the surface before the second context attempts to modify it. This applies to framebuffer objects as well as to pixel buffer objects.

When used incorrectly, `glFlush` or `glFinish` can cause your application to stall or slow down, using a higher percentage of the CPU than is necessary. You might also see visual anomalies, such as flickering or tearing. Most of the time you don't need to call `glFlush` or `glFinish` to move image data to the screen.

These are situations for which you don't need to, or should not, call `glFlush`:

- When the scene back buffer is not complete. For best results, keep the back buffer as current as possible with a complete scene. Since the flushing and finishing routines force OpenGL to process queued commands, calling either of these when the scene in the back buffer is incomplete causes the incomplete scene to be rendered.
- When calling a buffer swapping routine (the `flushBuffer` method of the `NSOpenGLContext` class, the `aglSwapBuffers` function, or the `CGLFlushDrawable` function), because such functions implicitly call the OpenGL command `glFlush`. Note that when using the `NSOpenGLContext` class or the CGL API, the term *flush* actually refers to a buffer swapping operation.

Be Mindful of OpenGL State Variables

The hardware has one current state, which is compiled and cached. Switching state is expensive, so it's best to design your application to minimize state switches.

Don't set a state that's already set. Once a feature is enabled, it does not need to be enabled again. Calling an enable function more than once does nothing except waste time because OpenGL does not check the state of a feature when you call `glEnable` or `glDisable`. For instance, if you call `glEnable(GL_LIGHTING)` more than once, OpenGL does not check to see if the lighting state is already enabled. It simply updates the state value even if that value is identical to the current value.

You can avoid setting a state more than necessary by using dedicated setup or shutdown routines rather than putting such calls in a drawing loop. Setup and shutdown routines are also useful for turning on and off features that achieve a specific visual effect—for example, when drawing a wire-frame outline around a textured polygon.

If you are drawing 2D images, disable all irrelevant state variables, similar to what's shown in Listing 12-1.

Listing 12-1 Disabling state variables

```
glDisable(GL_DITHER);
glDisable(GL_ALPHA_TEST);
glDisable(GL_BLEND);
glDisable(GL_STENCIL_TEST);
glDisable(GL_FOG);
glDisable(GL_TEXTURE_2D);
glDisable(GL_DEPTH_TEST);
glPixelZoom(1.0,1.0);
// Disable other state variables as appropriate.
```

Draw Only When Necessary

You can ensure that your application draws only when necessary by following a few simple guidelines:

- Allow the system to regulate drawing. For example, in Cocoa use the `setNeedsDisplay:` method.
- Use a timer effectively.
- Advance an animation only when necessary. To determine when to draw the next frame of an animation, calculate the difference between the current time and the start of the last frame. Use the difference to determine how much to advance the animation. You can use the Core Foundation function `CFAbsoluteTimeGetCurrent` to obtain the current time. Don't simply draw each time the system sends the `drawRect:` method because the view can be redrawn for reasons other than a timer firing—such as when the user resizes the window.

Drawing is typically triggered by a timer that fires at a set interval. Timer intervals that are set to very small values (such as 0.001 to yield 1000 executions per second) degrade application performance because they consume CPU time at a far higher rate than is necessary. In most cases, drawing 30 to 60 frames per second is sufficient (.033 to .0167 seconds). You'll get the best performance if you synchronize drawing to the refresh rate of the screen, which means that you should not set the timer interval to anything faster than the refresh rate.

The code in Listing 12-2 shows how to set up a timer in the rendering loop of a Cocoa application. When using a timer in Cocoa, make sure that you do not invoke the `drawRect:` method from the rendering loop. Instead, allow the system to send the `drawRect:` message when it needs to draw. This way, the system also takes care of locking and unlocking focus on the view.

The timer code shown in Listing 12-2 is in the `awakeFromNib` method to ensure that the timer starts up when the application launches. The timer interval is set to 100 milliseconds, which is 10 frames per second. Note that this is slower than the refresh rate, so that there is no risk of overdriving the animation and degrading performance. The `timerFired:` method is called by the system each time the timer fires. When called, this method signals to the system that the display needs refreshing.

Listing 12-2 Setting up a drawing loop timer

```
-(void)awakeFromNib
{
renderTimer = [[NSTimer scheduledTimerWithTimeInterval:
           0.1
           target:self
           selector:@selector(timerFired:)
           userInfo:nil
           repeats:YES]
           retain];
}

- (void)timerFired:(id)sender
{
    [self setNeedsDisplay:YES];
}
```

Synchronize with the Screen Refresh Rate

Tearing is a visual anomaly caused when part of the current frame overwrites previous frame data in the framebuffer before the current frame is fully rendered on the screen. Applications synchronize with the screen refresh rate (sometimes called VBL, vertical blank, or vsynch) to eliminate frame tearing.

Note: During development, it's best to disable synchronization so that you can more accurately benchmark your application. Enable synchronization when you are ready to deploy your application.

The refresh rate of the display limits how often the screen can be refreshed. The screen can be refreshed at rates that are divisible by integer values. For example, a CRT display that has a refresh rate of 60 Hz can support screen refresh rates of 60 Hz, 30 Hz, 20 Hz, and 15 Hz. LCD displays do not have a vertical retrace in the CRT sense and are typically considered to have a fixed refresh rate of 60 Hz.

OpenGL blocks drawing to the display while waiting for the next vertical retrace. Applications that attempt to draw to the screen during this waiting period waste time that could be spent performing other drawing operations or saving battery life and minimizing fan operation.

Listing 12-3 shows how to use the CGL API to synchronize with the screen refresh rate, but you can use a similar approach if your application uses Cocoa or the AGL API. It assumes that you set up the context for double buffering. The swap interval can be set only to `0` or `1`. If the swap interval is set to `1`, the buffers are swapped only during the vertical retrace. After you set up synchronization, call the function `CGLFlushDrawable` to copy the back buffer to the front buffer during the vertical retrace of the display.

Listing 12-3 Setting up synchronization

```
long swapInterval = 1;
CGLSetParameter (CGLGetCurrentContext(), kCGLCPSwapInterval, &swapInterval);
```

Use OpenGL Macros

OpenGL performs a global context and renderer lookup for each command it executes to ensure that all OpenGL commands are issued to the correct rendering context and renderer. There is significant overhead associated with these lookups; applications that have extremely high call frequencies may find that the overhead measurably affects performance. Mac OS X allows your application to use macros to provide a local context variable and cache the current renderer in that variable. You'll get the most out of using macros when your code makes millions of function calls per second. Then you'll see a noticeable boost in imaging response.

You can use the CGL macro header (`CGL/cglMacro.h`) if your application uses CGL from either a Cocoa or a Carbon application, and the AGL macro header (`AGL/aglMacro.h`) for Carbon applications. You must define the local variable `cgl_ctx` or `agl_ctx` to be equal to the current context. Listing 12-4 shows what's needed to set up macro use for the AGL API. You use a similar approach for the CGL API. First, you need to include the correct macro header. Then, you must set the current context.

Listing 12-4 Using AGL macros

```
#include <AGL/aglMacro.h> // include the header
AGLContext agl_ctx = myContext; // set the current context
glBegin (GL_QUADS);     // This code now uses the macro
    // draw here
```

```
glEnd ();
```

Use the CPU and the GPU Asynchronously

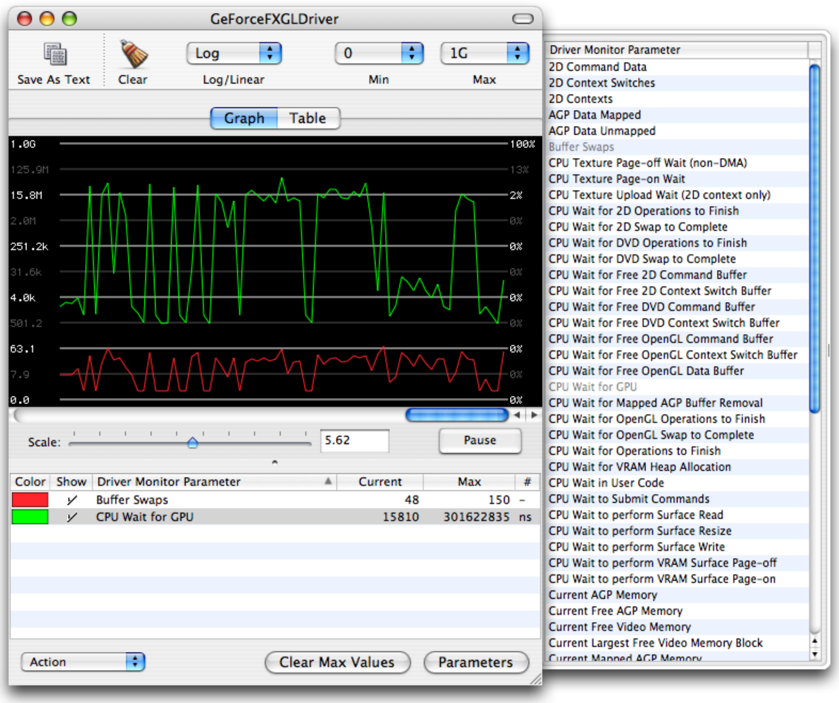
Whenever it’s feasible to do so, it's best to keep both the CPU and GPU busy and working as asynchronously as possible. You'll want to avoid pushing data through the bottleneck between the two units unless it is absolutely necessary.

These tips can help use the CPU and GPU optimally:

- Consider using the GPU to perform intense mathematical computations to take some of the load off the CPU.
- Use double buffering and asynchronous vertex transfer, as described in “Double Buffering Vertex Data” (page 93). If you are using the vertex array range extension, consider experimenting with triple buffers or changing buffer size.
- Use asynchronous texture fetching (see “Downloading Texture Data” (page 107)) rather than calling the function `glReadPixels`. This call is an expensive one because it forces synchronization between the CPU and GPU, which can have the effect of stalling the rendering pipeline. Performance degrades if either the CPU or GPU is waiting for the other processing unit to catch up.
- Upload textures asynchronously using DMA. See “Apple Texture Range and Rectangle Texture” (page 98).

You can use OpenGL Driver Monitor to analyze how long the CPU waits for the GPU, as shown in Figure 12-2. OpenGL Driver Monitor is useful for analyzing other parameters as well. You can choose which parameters to monitor simply by clicking a parameter name from the drawer shown in the figure.

Figure 12-2 The graph view in OpenGL Driver Monitor



Color	Show	Driver Monitor Parameter	Current	Max	#
🟥	✓	Buffer Swaps	48	150	–
🟩	✓	CPU Wait for GPU	15810	301622835	ns

Driver Monitor Parameter
2D Command Data
2D Context Switches
2D Contexts
AGP Data Mapped
AGP Data Unmapped
Buffer Swaps
CPU Texture Page-off Wait (non-DMA)
CPU Texture Page-on Wait
CPU Texture Upload Wait (2D context only)
CPU Wait for 2D Operations to Finish
CPU Wait for 2D Swap to Complete
CPU Wait for DVD Operations to Finish
CPU Wait for DVD Swap to Complete
CPU Wait for Free 2D Command Buffer
CPU Wait for Free 2D Context Switch Buffer
CPU Wait for Free DVD Command Buffer
CPU Wait for Free DVD Context Switch Buffer
CPU Wait for Free OpenGL Command Buffer
CPU Wait for Free OpenGL Context Switch Buffer
CPU Wait for Free OpenGL Data Buffer
CPU Wait for GPU
CPU Wait for Mapped AGP Buffer Removal
CPU Wait for OpenGL Operations to Finish
CPU Wait for OpenGL Swap to Complete
CPU Wait for Operations to Finish
CPU Wait for VRAM Heap Allocation
CPU Wait in User Code
CPU Wait to Submit Commands
CPU Wait to perform Surface Read
CPU Wait to perform Surface Resize
CPU Wait to perform Surface Write
CPU Wait to perform VRAM Surface Page-off
CPU Wait to perform VRAM Surface Page-on
Current AGP Memory
Current Free AGP Memory
Current Free Video Memory
Current Largest Free Video Memory Block
Current Mapped AGP Memory

Use Appropriate Routines for Images and Pixel Data

The size of an image should guide the routine you choose to draw it. Most images you'll draw will have dimensions greater than 128 pixels by 128 pixels. It's best to treat those images as texture data. See the *OpenGL Image* sample application on Sample Code > Graphics & Imaging > OpenGL for an example of high performance image display.

For small images, those whose dimensions are less than or equal to 128 pixels by 128 pixels, use the OpenGL function `glDrawPixels`. See the *Draw Pixels* sample application on Sample Code > Graphics & Imaging > OpenGL for an example of the correct use of `glDrawPixels`.

Copying pixel data from one VRAM location to another VRAM location, for example, to an auxiliary buffer, requires an approach similar to the one used to draw image data. Perform the copy operation using the OpenGL function `glCopyPixels`, as shown in Listing 12-5. If you are using Cocoa, you can use the `NSOpenGLPFAAuxBuffers` and `NSOpenGLPFAAuxDepthStencil` pixel format attributes in conjunction with `glReadBuffer` and`glDrawBuffer` to set up auxiliary buffers for temporary pixel storage.

Using OpenGL for drawing images and copying pixel data can incur a performance cost if the OpenGL state is a complex one. State variables such as dithering, fog, and depth testing don't need to be enabled for 2D drawing. To ensure efficient drawing, first disable irrelevant state variables as shown in Listing 12-1 (page 123).

Listing 12-5 Copying pixels

```
void drawRect:(NSRect) aRect
{
    glDrawBuffer(GL_BACK);
    glReadBuffer(GL_AUX0);

    glCopyPixels(x, y, width, height, GL_COLOR);
}
```

Retrieve Error Information Only When Debugging

When errors occur OpenGL sets an error flag that you can retrieve with the function `glGetError`. During development, it's crucial that your code contains error checking routines, not only for the standard OpenGL calls, but for the Apple-specific functions provided by the AGL and CGL APIs. AGL uses a mechanism for errors that's similar to OpenGL through the functions `aglGetError` and `aglErrorString`. CGL functions return error codes.

If you are developing a performance-critical application, you'll want to retrieve error information in the debugging phase. When you deploy your application you'll want to remove the error-retrieval information for all but the most critical cases. If you retrieve error codes and strings for frequently-called functions, you'll cause performance to slow down.

Use Optimal Data Types and Formats

If you don't use data types and formats that are native to the graphics processor, you'll incur a costly data conversion.

For vertex data, use `GLfloat`, `GLshort`, or `GLubyte` data types. Most graphics processors handle these types natively.

For texture data, you'll get the best performance, regardless of architecture, if you use the following format and data type combination:

```
GL_BGRA, GL_UNSIGNED_INT_8_8_8_8_REV
```

These format and data type combinations also provide acceptable performance:

```
GL_BGRA, GL_UNSIGNED_SHORT_1_5_5_5_REV
GL_YCBCR_422_APPLE, GL_UNSIGNED_SHORT_8_8_REV_APPLE
```

The combination `GL_RGBA` and `GL_UNSIGNED_BYTE` needs to be swizzled by many cards when the data is loaded, so it's not recommended.

Gathering and Analyzing Baseline Performance Data

Analyzing performance is a systematic process that starts with gathering baseline data. Mac OS X provides several applications that you can use to assess baseline performance for an OpenGL application:

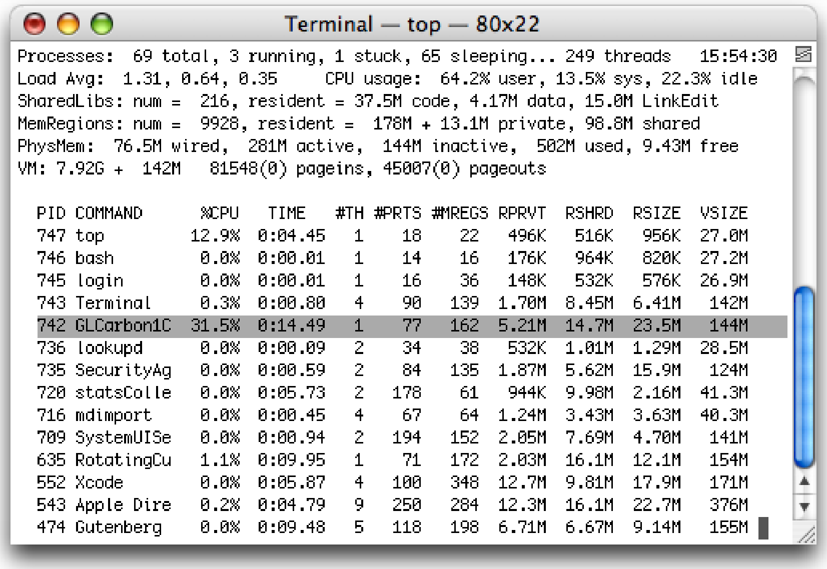
- `top` is a command-line utility that you run in the Terminal window. You can use `top` to assess how much CPU time your application consumes.
- OpenGL Profiler is an application that determines how much time an application spends in OpenGL. It also provides function traces that you can use to look for redundant calls.
- OpenGL Driver Monitor lets you gather real-time data on the operation of the GPU and lets you look at information (OpenGL extensions supported, buffer modes, sample modes, and so forth) for the available renderers. For more information, see OpenGL Tools for Serious Graphics Development and "Use the CPU and the GPU Asynchronously" (page 126).

This section shows how to use `top` along with OpenGL Profiler to analyze where to spend your optimization efforts—in your OpenGL code, your other application code, or in both. You'll see how to gather baseline data and how to determine the relationship of OpenGL performance to overall application performance.

1. Launch your OpenGL application.
2. Open a Terminal window and place it side-by-side with your application window.
3. In the Terminal window, type `top` and press Return. You'll see output similar to that shown in Figure 12-3.

The `top` program indicates the amount of CPU time that an application uses. The CPU time serves as a good baseline value for gauging how much tuning your code needs. Figure 12-3 shows the percentage of CPU time for the OpenGL application GLCarbon1C (highlighted). Note this application utilizes 31.5% of CPU resources.

Figure 12-3 Output produced by the `top` application

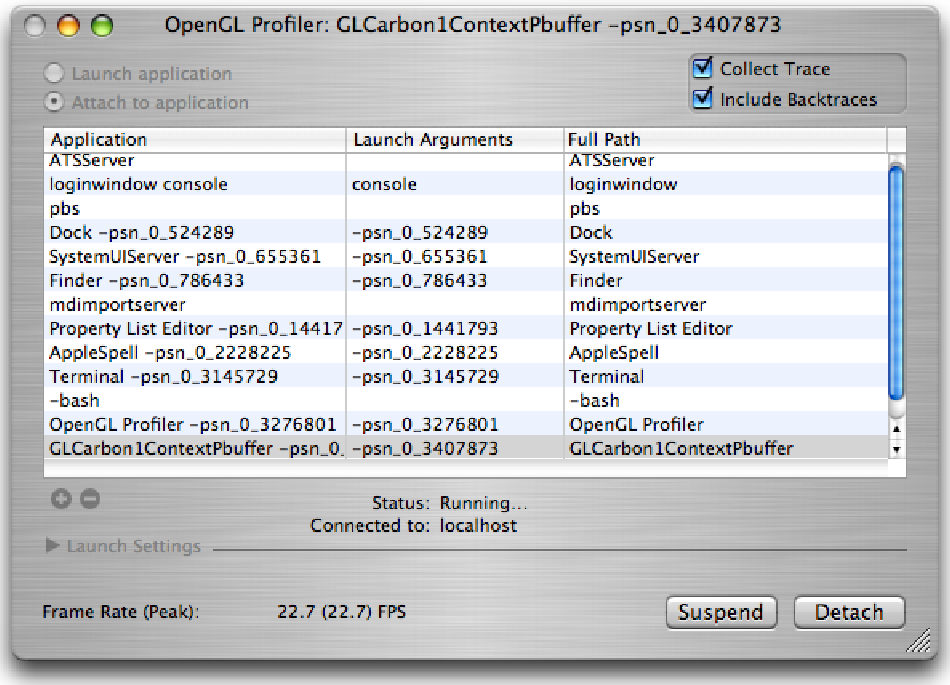


Terminal — top — 80x22

```
Processes:  69 total, 3 running, 1 stuck, 65 sleeping... 249 threads   15:54:30
Load Avg:  1.31, 0.64, 0.35     CPU usage:  64.2% user, 13.5% sys, 22.3% idle
SharedLibs: num =  216, resident = 37.5M code, 4.17M data, 15.0M LinkEdit
MemRegions: num =  9928, resident =  178M + 13.1M private, 98.8M shared
PhysMem:  76.5M wired,  281M active,  144M inactive,  502M used, 9.43M free
VM: 7.92G +  142M   81548(0) pageins, 45007(0) pageouts

  PID COMMAND      %CPU   TIME   #TH #PRTS #MREGS RPRVT  RSHRD  RSIZE  VSIZE
  747 top         12.9% 0:04.45   1    18    22   496K   516K   956K  27.0M
  746 bash         0.0% 0:00.01   1    14    16   176K   964K   820K  27.2M
  745 login        0.0% 0:00.01   1    16    36   148K   532K   576K  26.9M
  743 Terminal     0.3% 0:00.80   4    90   139  1.70M  8.45M  6.41M   142M
  742 GLCarbon1C  31.5% 0:14.49   1    77   162  5.21M  14.7M  23.5M   144M
  736 lookupd      0.0% 0:00.09   2    34    38   532K  1.01M  1.29M  28.5M
  735 SecurityAg   0.0% 0:00.59   2    84   135  1.87M  5.62M  15.9M   124M
  720 statsColle   0.0% 0:05.73   2   178    61   944K  9.98M  2.16M  41.3M
  716 mdimport     0.0% 0:00.45   4    67    64  1.24M  3.43M  3.63M  40.3M
  709 SystemUISe   0.0% 0:00.94   2   194   152  2.05M  7.69M  4.70M   141M
  635 RotatingCu   1.1% 0:09.95   1    71   172  2.03M  16.1M  12.1M   154M
  552 Xcode        0.0% 0:05.87   4   100   348  12.7M  9.81M  17.9M   171M
  543 Apple Dire   0.2% 0:04.79   9   250   284  12.3M  16.1M  22.7M   376M
  474 Gutenberg    0.0% 0:09.48   5   118   198  6.71M  6.67M  9.14M   155M
```

4. Open the OpenGL Profiler application, located in `/Developer/Applications/Graphics Tools/`. In the window that appears, select the options to collect a trace and include backtraces, as shown in Figure 12-4.

Figure 12-4 The OpenGL Profiler window



OpenGL Profiler: GLCarbon1ContextPbuffer -psn_0_3407873

Launch application [✓ Attach to application]

✓ Collect Trace
✓ Include Backtraces

Application	Launch Arguments	Full Path
ATSServer		ATSServer
loginwindow console	console	loginwindow
pbs		pbs
Dock -psn_0_524289	-psn_0_524289	Dock
SystemUIServer -psn_0_655361	-psn_0_655361	SystemUIServer
Finder -psn_0_786433	-psn_0_786433	Finder
mdimportserver		mdimportserver
Property List Editor -psn_0_14417	-psn_0_1441793	Property List Editor
AppleSpell -psn_0_2228225	-psn_0_2228225	AppleSpell
Terminal -psn_0_3145729	-psn_0_3145729	Terminal
-bash		-bash
OpenGL Profiler -psn_0_3276801	-psn_0_3276801	OpenGL Profiler
GLCarbon1ContextPbuffer -psn_0_	-psn_0_3407873	GLCarbon1ContextPbuffer

Status: Running...
Connected to: localhost

Frame Rate (Peak): 22.7 (22.7) FPS

5. Select Attach to application, then select your application from the Application list.

   You may see small pauses or stutters in the application, particularly when OpenGL Profiler is collecting a function trace. This is normal and does not significantly affect the performance statistics. The "glitches" are due to the large amount of data that OpenGL Profiler is writing out.

6. Click Suspend to stop data collection.

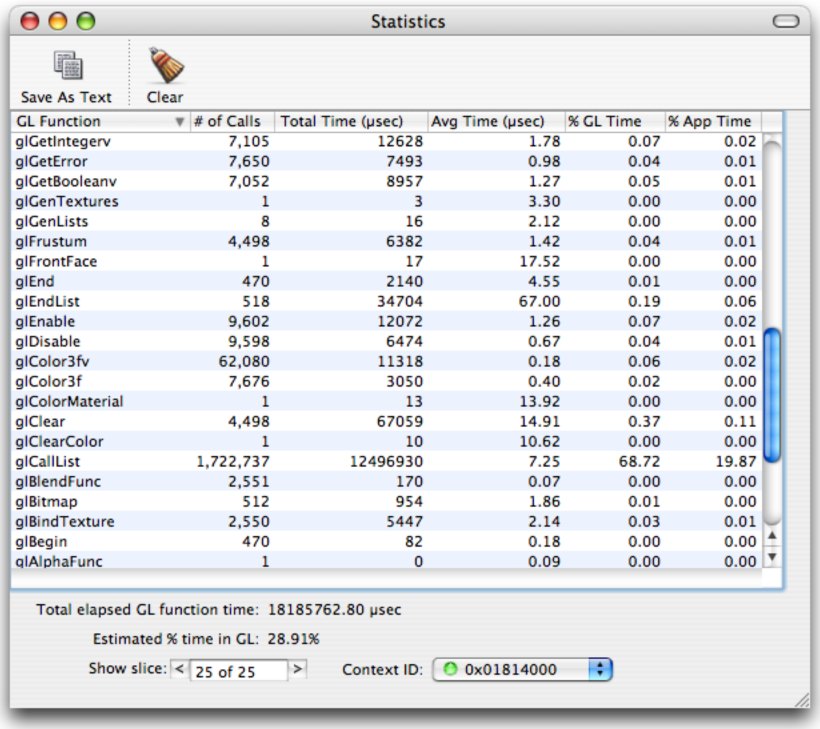
7. Open the Statistics and Trace windows by choosing them from the Views menu.

   Figure 12-5 provides an example of what the Statistics window looks like. Figure 12-6 (page 132) shows a Trace window.

   The estimated percentage of time spent in OpenGL is shown at the bottom of Figure 12-5. Note that for this example, it is 28.91%. The higher this number, the more time the application is spending in OpenGL and the more opportunity there may be to improve application performance by optimizing OpenGL code.

You can use the amount of time spent in OpenGL along with the CPU time to calculate a ratio of the application time versus OpenGL time. This ratio indicates where to spend most of your optimization efforts.

Figure 12-5 A statistics window



GL Function	# of Calls	Total Time (µsec)	Avg Time (µsec)	% GL Time	% App Time
glGetIntegerv	7,105	12628	1.78	0.07	0.02
glGetError	7,650	7493	0.98	0.04	0.01
glGetBooleanv	7,052	8957	1.27	0.05	0.01
glGenTextures	1	3	3.30	0.00	0.00
glGenLists	8	16	2.12	0.00	0.00
glFrustum	4,498	6382	1.42	0.04	0.01
glFrontFace	1	17	17.52	0.00	0.00
glEnd	470	2140	4.55	0.01	0.00
glEndList	518	34704	67.00	0.19	0.06
glEnable	9,602	12072	1.26	0.07	0.02
glDisable	9,598	6474	0.67	0.04	0.01
glColor3fv	62,080	11318	0.18	0.06	0.02
glColor3f	7,676	3050	0.40	0.02	0.00
glColorMaterial	1	13	13.92	0.00	0.00
glClear	4,498	67059	14.91	0.37	0.11
glClearColor	1	10	10.62	0.00	0.00
glCallList	1,722,737	12496930	7.25	68.72	19.87
glBlendFunc	2,551	170	0.07	0.00	0.00
glBitmap	512	954	1.86	0.01	0.00
glBindTexture	2,550	5447	2.14	0.03	0.01
glBegin	470	82	0.18	0.00	0.00
glAlphaFunc	1	0	0.09	0.00	0.00

Total elapsed GL function time: 18185762.80 µsec

Estimated % time in GL: 28.91%

Show slice: < 25 of 25 > Context ID: 0x01814000

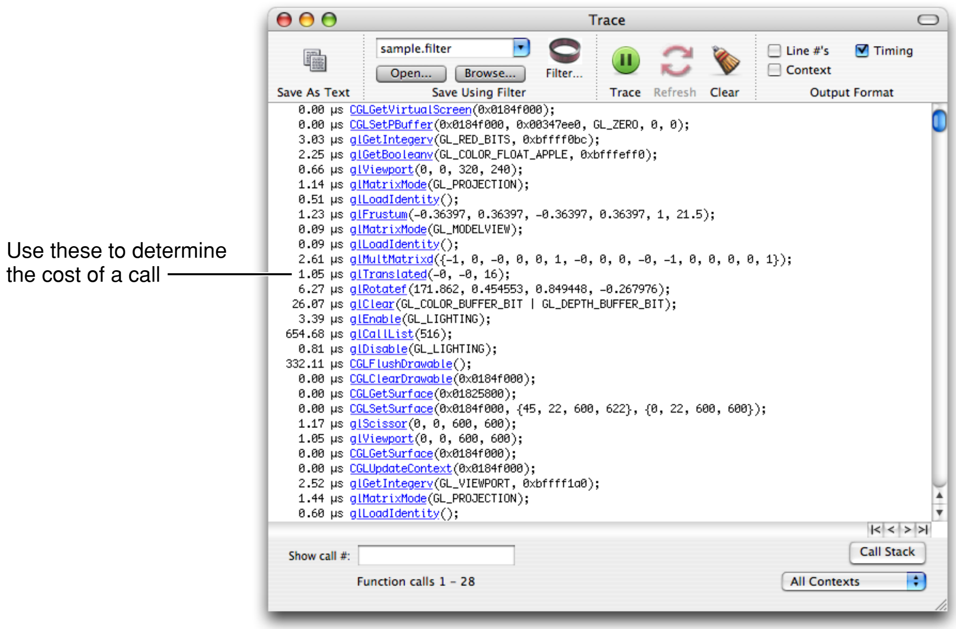
8. In the Trace window, look for duplicate function calls and redundant or unnecessary state changes.

Look for back-to-back function calls with the same or similar data. These are areas that can typically be optimized. Functions that are called more than necessary include `glTexParameter`, `glPixelStore`, `glEnable`, and `glDisable`. For most applications, these functions can be called once from a setup or state modification routine and only called when necessary.

It's generally good practice to keep state changes out of rendering loops (which can be seen in the function trace as the same sequence of state changes and drawing over and over again) as much as possible and use separate routines to adjust state as necessary.

Look at the time value to the left of each function call to determine the cost of the call.

Figure 12-6 A Trace window



```
  0.00 µs CGLGetVirtualScreen(0x0184f000);
  0.00 µs CGLSetPBuffer(0x0184f000, 0x00347ee0, GL_ZERO, 0, 0);
  3.03 µs glGetIntegerv(GL_RED_BITS, 0xbffff0bc);
  2.25 µs glGetBooleanv(GL_COLOR_FLOAT_APPLE, 0xbfffeff0);
  0.66 µs glViewport(0, 0, 320, 240);
  1.14 µs glMatrixMode(GL_PROJECTION);
  0.51 µs glLoadIdentity();
  1.23 µs glFrustum(-0.36397, 0.36397, -0.36397, 0.36397, 1, 21.5);
  0.09 µs glMatrixMode(GL_MODELVIEW);
  0.09 µs glLoadIdentity();
  2.61 µs glMultMatrixd({-1, 0, -0, 0, 0, 1, -0, 0, 0, -0, -1, 0, 0, 0, 0, 1});
  1.05 µs glTranslated(-0, -0, 16);
  6.27 µs glRotatef(171.862, 0.454553, 0.849448, -0.267976);
 26.07 µs glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  3.39 µs glEnable(GL_LIGHTING);
654.68 µs glCallList(516);
  0.81 µs glDisable(GL_LIGHTING);
332.11 µs CGLFlushDrawable();
  0.00 µs CGLClearDrawable(0x0184f000);
  0.00 µs CGLGetSurface(0x01825800);
  0.00 µs CGLSetSurface(0x0184f000, {45, 22, 600, 622}, {0, 22, 600, 600});
  1.17 µs glScissor(0, 0, 600, 600);
  1.05 µs glViewport(0, 0, 600, 600);
  0.00 µs CGLGetSurface(0x0184f000);
  0.00 µs CGLUpdateContext(0x0184f000);
  2.52 µs glGetIntegerv(GL_VIEWPORT, 0xbffff1a0);
  1.44 µs glMatrixMode(GL_PROJECTION);
  0.60 µs glLoadIdentity();
```

9. Determine what the performance gain would be if it were possible to reduce the time to execute all OpenGL calls to zero.

   For example, take the performance data from the GLCarbon1C application used in this section to determine the performance attributable to the OpenGL calls.

   Total Application Time (from `top`) = 31.5%

   Total Time in OpenGL (from OpenGL Profiler) = 28.91%

   At first glance, you might think that optimizing the OpenGL code could improve application performance by almost 29%, thus reducing the total application time by 29%. This isn't the case. Calculate the theoretical performance increase by multiplying the total CPU time by the percentage of time spent in OpenGL. The theoretical performance improvement for this example is:

   ```
   31.5 X .2891 = 9.11%
   ```

   If OpenGL took no time at all to execute, the application would see a 9.11% increase in performance. So, if the application runs at 60 frames per second (FPS), it would perform as follows:

   ```
   New FPS = previous FPS * (1 +(% performance increase)) = 60 fps *(1.0911) =
   65.47 fps
   ```

   The application gains almost 5.5 frames per second by reducing OpenGL from 28.91% to 0%. This shows that the relationship of OpenGL performance to application performance is not linear. Simply reducing the amount of time spent in OpenGL may or may not offer any noticeable benefit in application performance.

Identifying Bottlenecks with Shark

Shark is an extremely useful tool for identifying places in your code that are slow and could benefit from optimization. If you are not familiar with Shark, read some of the documents listed in "See Also" (page 133) that describe Shark in detail and show how to use it. Once you learn the basics, you can use it on your OpenGL applications to identify bottlenecks.

There are three issues to watch out for in Shark when using it to analyze OpenGL performance:

- Costly data conversions. If you notice the `glgProcessPixels` call (in the `libGLImage.dylib` library) showing up in the analysis, it's an indication that the driver is not handling a texture upload optimally. The call is used when your application makes a `glTexImage` or `glTexSubImage` call using data that is in a nonnative format for the driver, which means the data must be converted before the driver can upload it. You can improve performance by changing your data so that it is in a native format for the driver. See "Use Optimal Data Types and Formats" (page 127).

  Note: If your data needs only to be swizzled, `glgProcessPixels` performs the swizzling reasonably fast although not as fast if the data didn't need swizzling. But non-native data formats are converted one byte at a time and will incur a performance cost that is best to avoid.

- Time in the `mach_kernel` library. If you see time spent waiting for a timestamp or waiting for the driver, it indicates that your application is waiting for the GPU to finish processing. You'll see this during a texture upload. See "Double Buffering Texture Data" (page 108) and "Use the CPU and the GPU Asynchronously" (page 126) for ideas on how you might optimize asynchronous behavior between the CPU and the GPU.
- Misleading symbols. You may see a symbol, such as `glgGetString`, that appears to be taking time but shouldn't be taking time in your application. That sometimes happens because the underlying optimizations performed by the system don't have any symbols attached to them on the driver side. Without a symbol to display, Shark shows the last symbol. You need to look for the call that your application made prior to that symbol and focus your attention there. You don't need to concern yourself with the calls that were made "underneath" your call.

See Also

If you are unfamiliar with general performance issues on the Macintosh platform, you will want to read Getting Started with Performance and *Performance Overview*. *Performance Overview* contains general performance tips that are useful to all applications. It also describes most of the performance tools provided with Mac OS X, including:

- Analysis tools—MallocDebug, ObjectAlloc, OpenGL Profiler, Sampler, Saturn, Shark, `heap`, `leaks`, and `vmmap`
- Monitoring tools—BigTop, Quartz Debug, Spin Control, Thread Viewer, `fs_usage`, `sc_usage`, and `top`
- Hardware analysis tools—CacheBasher, MONster, PMC Index, Reggie SE, Skidmarks GT, `acid`, `amber`, `simg4`, and `simg5`
- Assorted command-line tools—`atos`, `c2ph`, `gprof`, `kdump`, `malloc_history`, `nm`, `otool`, `pagestuff`, `pstruct`, `sample`, `vm_stat`

There are two tools other than OpenGL Profiler that are specific for OpenGL development—OpenGL Driver Monitor and OpenGL Shader Builder. OpenGL Driver Monitor collects real-time data from the hardware. OpenGL Shader Builder provides immediate feedback on vertex and fragment programs that you write.

For more information on these tools, see:

- OpenGL Tools for Serious Graphics Development
- Using Shark
- Optimizing with Shark: Big Payoff, Small Effort
- *Shark User Guide*, available by launching Shark and choosing Help > Shark Help.
- CHUD Tools
- OpenGL Profiler
- OpenGL Driver Monitor

The following books contain many techniques for getting the most performance from the GPU:

- *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, Randima Fernando.
- *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Matt Pharr and Randima Fernando.

OpenGL Functionality by Version

The core OpenGL functionality changes with each new version of the OpenGL API. This appendix describes the functionality that was added with each version. See the official OpenGL specification for detailed information.

The functionality for each version is guaranteed to be available through the core OpenGL API even if a particular renderer does not support all of the extensions in a version. For example, a renderer that claims to support OpenGL 1.3 might not export the `GL_ARB_texture_env_combine` or `GL_EXT_texture_env_combine` extensions. It's important that you query both the renderer version and extension string to make sure that the renderer supports any functionality that you want to use.

Note: It's possible for vendor and ARB extensions to provide similar functionality. As particular functionality becomes widely adopted, it can be moved into the core OpenGL API. As a result, functionality that you want to use could be included as an extension, as part of the core API, or both. You should read the extensions and the core OpenGL specifications carefully to see the differences. Furthermore, as an extension is promoted, the API associated with that functionality can change. For more information, see "Determining the OpenGL Capabilities Supported by the Hardware" (page 59).

In the following tables, the extensions describe the feature that the core functionality is based on. The core functionality might not be the same as the extension. For example, compare the core texture crossbar functionality with the extension that it's based on.

Version 1.1

Table A-1 Functionality added in OpenGL 1.1

Functionality	Extension
Copy texture and subtexture	`GL_EXT_copy_texture` and `GL_EXT_subtexture`
Logical operation	`GL_EXT_blend_logic_op`
Polygon offset	`GL_EXT_polygon_offset`
Texture image formats	`GL_EXT_texture`
Texture objects	`GL_EXT_texture_object`
Texture proxies	`GL_EXT_texture`
Texture replace environment	`GL_EXT_texture`
Vertex array	`GL_EXT_vertex_array`

A number of other minor changes outlined in Appendix C section 9 of the OpenGL specification. See http://www.opengl.org.

Version 1.2

Table A-2 Functionality added in OpenGL 1.2

Functionality	Extension
BGRA pixel formats	GL_EXT_bgra
Imaging subset (optional)	GL_SGI_color_table, GL_EXT_color_subtable, GL_EXT_convolution, GL_HP_convolution_border_modes, GL_SGI_color_matrix, GL_EXT_histogram, GL_EXT_blend_minmax, and GL_EXT_blend_subtract
Normal rescaling	GL_EXT_rescale_normal
Packed pixel formats	GL_EXT_packed_pixels
Separate specular color	GL_EXT_separate_specular_color
Texture coordinate edge clamping	GL_SGIS_texture_edge_clamp
Texture level of detail control	GL_SGIS_texture_lod
Three-dimensional texturing	GL_EXT_texture3D
Vertex array draw element range	GL_EXT_draw_range_elements

Note: The imaging subset might not be present on all implementations; you must verify by checking for the ARB_imaging extension.

OpenGL 1.2.1 introduced ARB extensions with no specific core API changes.

Version 1.3

Table A-3 Functionality added in OpenGL 1.3

Functionality	Extension
Compressed textures	GL_ARB_texture_compression
Cube map textures	GL_ARB_texture_cube_map
Multisample	GL_ARB_multisample

Functionality	Extension
Multitexture	GL_ARB_multitexture
Texture add environment mode	GL_ARB_texture_env_add
Texture border clamp	GL_ARB_texture_border_clamp
Texture combine environment mode	GL_ARB_texture_env_combine
Texture dot3 environment mode	GL_ARB_texture_env_dot3
Transpose matrix	GL_ARB_transpose_matrix

Version 1.4

Table A-4 Functionality added in OpenGL 1.4

Functionality	Extension
Automatic mipmap generation	GL_SGIS_generate_mipmap
Blend function separate	GL_ARB_blend_func_separate
Blend squaring	GL_NV_blend_square
Depth textures	GL_ARB_depth_texture
Fog coordinate	GL_EXT_fog_coord
Multiple draw arrays	GL_EXT_multi_draw_arrays
Point parameters	GL_ARB_point_parameters
Secondary color	GL_EXT_secondary_color
Separate blend functions	GL_EXT_blend_func_separate, GL_EXT_blend_color
Shadows	GL_ARB_shadow
Stencil wrap	GL_EXT_stencil_wrap
Texture crossbar environment mode	GL_ARB_texture_env_crossbar
Texture level of detail bias	GL_EXT_texture_lod_bias
Texture mirrored repeat	GL_ARB_texture_mirrored_repeat
Window raster position	GL_ARB_window_pos

Version 1.5

Table A-5 Functionality added in OpenGL 1.5

Functionality	Extension
Buffer objects	GL_ARB_vertex_buffer_object
Occlusion queries	GL_ARB_occlusion_query
Shadow functions	GL_EXT_shadow_funcs

Version 2.0

Table A-6 Functionality added in OpenGL 2.0

Functionality	Extension
Multiple render targets	GL_ARB_draw_buffers
Non–power-of-two textures	GL_ARB_texture_non_power_of_two
Point sprites	GL_ARB_point_sprite
Separate blend equation	GL_EXT_blend_equation_separate
Separate stencil	GL_ATI_separate_stencil GL_EXT_stencil_two_side
Shading language	GL_ARB_shading_language_100
Shader objects	GL_ARB_shader_objects
Shader programs	GL_ARB_fragment_shader GL_ARB_vertex_shader

Setting Up Function Pointers to OpenGL Routines

Function pointers to OpenGL routines allow you to deploy your application across multiple versions of Mac OS X regardless of whether the entry point is supported at link time or runtime. This practice also provides support for code that needs to run cross-platform—in both Mac OS X and Windows.

Note: If you are deploying your application only in Mac OS X v10.4 or later, you do not need to read this chapter. Instead, consider the alternative, which is to set the `gcc` attribute that allows weak linking of symbols. Keep in mind, however, that weak linking may impact your application's performance. For more information, see Frameworks and Weak Linking.

This appendix discusses the tasks needed to set up and use function pointers as entry points to OpenGL routines:

- "Obtaining a Function Pointer to an Arbitrary OpenGL Entry Point" (page 139) shows how to write a generic routine that you can reuse for any OpenGL application on the Macintosh platform.
- "Initializing Entry Points" (page 142) describes how to declare function pointer type definitions and initialize them with the appropriate OpenGL command entry points for your application.

Obtaining a Function Pointer to an Arbitrary OpenGL Entry Point

Getting a pointer to an OpenGL entry point function is fairly straightforward from either Cocoa or Carbon. In either framework in Mac OS X, you can use the Dynamic Loader function `NSLookupAndBindSymbol` to get the address of an OpenGL entry point. The Dynamic Loader is part of the system framework, not part of Cocoa, which is why `NSLookupAndBindSymbol` (declared in `/usr/include/mach-o/dyld.h`) works in Mach-O Carbon applications as well as Cocoa ones. Carbon applications also have the option of using the AGL API, although this approach involves a bit more code. You'll see how to use both approaches in this section.

Keep in mind that getting a valid function pointer means that the entry point is exported by the OpenGL framework; it does not guarantee that a particular routine is supported and valid to call from within your application. You still need to check for OpenGL functionality on a per-renderer basis as described in "Detecting Functionality" (page 59).

Listing B-1 shows how to use `NSLookupAndBindSymbol` from within the function `MyNSGLGetProcAddress`. When provided a symbol name, this application-defined function returns the appropriate function pointer from the global symbol table. A detailed explanation for each numbered line of code appears following the listing.

Listing B-1 Using `NSLookupAndBindSymbol` to obtain a symbol for a symbol name

```
#import <mach-o/dyld.h>
#import <stdlib.h
#import <string.h>
```

```
void * MyNSGLGetProcAddress (const char *name)
{
    NSSymbol symbol;
    char *symbolName;
    symbolName = malloc (strlen (name) + 2);                    // 1
    strcpy(symbolName + 1, name);                               // 2
    symbolName[0] = '_';                                        // 3
    symbol = NULL;
    if (NSIsSymbolNameDefined (symbolName))                     // 4
        symbol = NSLookupAndBindSymbol (symbolName);
    free (symbolName);                                          // 5
    return symbol ? NSAddressOfSymbol (symbol) : NULL;          // 6
}
```

Here's what the code does:

1. Allocates storage for the symbol name plus an underscore character ('_'). The underscore character is part of the UNIX C symbol-mangling convention, so make sure that you provide storage for it.

2. Copies the symbol name into the string variable, starting at the second character, to leave room for prefixing the underscore character.

3. Copies the underscore character into the first character of the symbol name string.

4. Checks to make sure that the symbol name is defined, and if it is, looks up the symbol.

5. Frees the symbol name string because it is no longer needed.

6. Returns the appropriate pointer if successful, or `NULL` if not successful. Before using this pointer, you should make sure that is it valid.

Using the AGL API to obtain a function pointer for an OpenGL entry point requires that you get a Core Foundation bundle reference. As a result, you need to perform a bit of set up work before you make the critical call to the Core Foundation function `CFBundleGetFunctionPointerForName`, as you'll see by looking at the code in Listing B-2. This code requires Carbon and is designed for use with Mach-O and CFM Carbon applications. You would use this approach only if you need to support older versions of your application. If your application runs only in Mac OS X, it should be a Mach-O application. A detailed explanation for each numbered line of code appears following the listing.

Listing B-2 Using AGL to get a function pointer for an entry in the OpenGL framework

```
 #include <Carbon/Carbon.h>
CFBundleRef gBundleRefOpenGL = NULL;

OSStatus MyAGLInitEntryPoints (void)
{
    OSStatus err = noErr;
    const Str255 frameworkName = "\pOpenGL.framework";          // 1
    FSRefParam fileRefParam;
    FSRef fileRef;
    CFURLRef bundleURLOpenGL;

    memset (&fileRefParam, 0, sizeof(fileRefParam));            // 2
    memset (&fileRef, 0, sizeof(fileRef));
    fileRefParam.ioNamePtr  = frameworkName;                    // 3
    fileRefParam.newRef = &fileRef;
    err = FindFolder (kSystemDomain, kFrameworksFolderType, false,
                        &fileRefParam.ioVRefNum,  &fileRefParam.ioDirID); // 4
    if (noErr != err) {
        DebugStr ("\pCould not find frameworks folder");
        return err;
    }
    err = PBMakeFSRefSync (&fileRefParam);                               // 5
    if (noErr != err) {
        DebugStr ("\pCould make FSRef to frameworks folder");
        return err;
    }
   bundleURLOpenGL = CFURLCreateFromFSRef (kCFAllocatorDefault,&fileRef);  // 6
    if (!bundleURLOpenGL) {
        DebugStr ("\pCould not create OpenGL Framework bundle URL");
        return paramErr;
    }
    gBundleRefOpenGL = CFBundleCreate(kCFAllocatorDefault, bundleURLOpenGL); // 7
    if (!gBundleRefOpenGL) {
        DebugStr ("\pCould not create OpenGL Framework bundle");
        return paramErr;
    }
    CFRelease (bundleURLOpenGL);                                         // 8
    if (!CFBundleLoadExecutable (gBundleRefOpenGL)) {                    // 9
        DebugStr ("\pCould not load Mach-O executable");
        return paramErr;
    }
    return err;
}

void MyAGLDeAllocEntryPoints (void)                                      // 10
{
    if (gBundleRefOpenGL != NULL) {
        CFBundleUnloadExecutable (gBundleRefOpenGL);
        CFRelease (gBundleRefOpenGL);
        gBundleRefOpenGL = NULL;
    }
}

void * MyAGLGetProcAddress (char * pszProc)                              // 11
{
    return CFBundleGetFunctionPointerForName (gBundleRefOpenGL,
                CFStringCreateWithCStringNoCopy (NULL,
                         pszProc, CFStringGetSystemEncoding (), NULL) );
}
```

Here's what the code does:

1. Declares a string for the framework name (`OpenGL.framework`), which is where you need to search for OpenGL function.
2. Sets up a buffer for the framework name. The next line does the same for the file reference.
3. Assigns the framework and then the file reference.
4. Finds the OpenGL framework directory; handles the error condition if the folder isn't found.

5. Creates an `FSRef` data structure for the OpenGL framework directory and handles an error condition should one occur.

6. Creates a Core Foundation URL from the `FSRef` data structure and handles an error condition should one occur.

7. Creates a Core Foundation bundle reference to the OpenGL framework and handles an error condition should one occur.

8. Releases the Core Foundation URL because it is no longer needed.

9. Loads the bundle.

10. Performs necessary clean up work.

11. Gets a function pointer for an OpenGL entry point.

Initializing Entry Points

Listing B-3 shows how to use the `MyNSGLGetProcAddress` function from Listing B-1 (page 139) to obtain a few OpenGL entry points. A detailed explanation for each numbered line of code appears following the listing.

Listing B-3 Using `NSGLGetProcAddress` to obtain an OpenGL entry point

```
#import "MyNSGLGetProcAddress.h"                                              // 1
static void InitEntryPoints (void);
static void DeallocEntryPoints (void);

// Function pointer type definitions
typedef void (*glBlendColorProcPtr)(GLclampf red,GLclampf green,
                        GLclampf blue,GLclampf alpha);
typedef void (*glBlendEquationProcPtr)(GLenum mode);
 typedef void (*glDrawRangeElementsProcPtr)(GLenum mode, GLuint start,
                GLuint end,GLsizei count,GLenum type,const GLvoid *indices);

glBlendColorProcPtr pfglBlendColor = NULL;                                    // 2
glBlendEquationProcPtr pfglBlendEquation = NULL;
glDrawRangeElementsProcPtr pfglDrawRangeElements = NULL;

static void InitEntryPoints (void)                                            // 3
{
    pfglBlendColor = (glBlendColorProcPtr) MyNSGLGetProcAddress
                                    ("glBlendColor");
    pfglBlendEquation = (glBlendEquationProcPtr)MyNSGLGetProcAddress
                                ("glBlendEquation");
    pfglDrawRangeElements = (glDrawRangeElementsProcPtr)MyNSGLGetProcAddress
                                ("glDrawRangeElements");
}
// -------------------------
static void DeallocEntryPoints (void)                                         // 4
{
    pfglBlendColor = NULL;
    pfglBlendEquation = NULL;
    pfglDrawRangeElements = NULL;;
}
```

Here's what the code does:

1. Imports the header file that contains the `MyNSGLProcAddress` function from Listing B-1 (page 139).
2. Declares function pointers for the functions of interest. Note that each function pointer uses the prefix `pf` to distinguish it from the function it points to. Although using this prefix is not a requirement, it's best to avoid using the exact function names.
3. Initializes the entry points. This function repeatedly calls the `MyNSGLProcAddress` function to obtain function pointers for each of the functions of interest—`glBlendColor`, `glBlendEquation`, and `glDrawRangeElements`.
4. Sets each of the function pointers to `NULL` when they are no longer needed.

Listing B-4 demonstrates how to use the function `aglGetProcAddress` to obtain a few OpenGL entry points. Note that the approach used by this code is similar to that used in Listing B-3 (page 142). A detailed explanation for each numbered line of code appears following the listing.

Listing B-4 Using AGL to obtain an OpenGL entry point

```
#include "MyAGLGetProcAddress.h"                                      // 1

static OSStatus InitEntryPoints (void);
static void DeallocEntryPoints (void);

typedef void (*glBlendColorProcPtr)(GLclampf red,GLclampf green,
                        GLclampf blue,GLclampf alpha);                // 2
typedef void (*glBlendEquationProcPtr)(GLenum mode);
typedef void (*glDrawRangeElementsProcPtr)(GLenum mode,GLuint start,
                        GLuint end,GLsizei count,GLenum type,
                        const GLvoid *indices);

glBlendColorProcPtr pfglBlendColor = NULL;                            // 3
glBlendEquationProcPtr pfglBlendEquation = NULL;
glDrawRangeElementsProcPtr pfglDrawRangeElements  = NULL;

static OSStatus InitEntryPoints (void)
{
    OSStatus err = MyAGLInitEntryPoints();                            // 4
    if (noErr == err) {                                               // 5
        pfglBlendColor = (glBlendColorProcPtr)
                    MyAGLGetProcAddress ("glBlendColor");
        pfglBlendEquation = (glBlendEquationProcPtr)
                    MyAGLGetProcAddress ("glBlendEquation");
        pfglDrawRangeElements = (glDrawRangeElementsProcPtr)
                MyAGLGetProcAddress("glDrawRangeElements");
    }
    return err;
}

static void DeallocEntryPoints (void)
{
    pfglBlendColor = NULL;                                                    // 6
    pfglBlendEquation = NULL;
    pfglDrawRangeElements = NULL;
    MyAGLDellocEntryPoints ();                                                // 7
}
```

Here's what the code does:

1. Imports the header file that contains the `MyAGLGetProcAdress` function from Listing B-2 (page 140).
2. Declares function pointers for the functions of interest.
3. Initializes each function pointer to `NULL`.
4. Calls the initialization function defined in Listing B-2 (page 140).
5. After checking for an error condition, obtains the function pointers for the functions of interest by calling the `MyAGLGetProcAddress` function defined in Listing B-2 (page 140).
6. Sets each of the function pointers to `NULL` when they are no longer needed.
7. Deallocates the function pointers when they are no longer needed by calling the deallocation function defined in Listing B-2 (page 140).

Quartz Display Services and Full-Screen Mode

The Quartz Display Services API provides functionality that is useful for any Mac OS X application using full-screen mode. In other parts of this programming guide, you've seen how to use Quartz Display Services to obtain exclusive access to the display, query the display, and adjust the resolution, depth, and refresh rate of the display. This appendix shows how to perform some additional tasks that are relevant to full-screen OpenGL applications:

- "Displays and Display Modes" (page 145) shows how to obtain information about the display and switch modes.
- "Fading the Display" (page 147) describes how to fade a display to black and then set it to its original state.
- "Controlling the Pointer" (page 149) discusses how to programmatically manage the cursor and disassociate mouse movement from the cursor.

For more information, see *Quartz Display Services Reference*.

Displays and Display Modes

Quartz Display Services functions allow you to enumerate all displays as well as the supported modes for each display. The Quartz Display Services functions `CGDisplaySwitchToMode` and `CGDisplayBestModeForParameters` use on the Core Foundation `CFDictionary` data type. Each display mode has a dictionary whose key-value pairs you can query. You can use accessor functions to query the properties of the current display mode. You can also use Core Foundation functions to access the dictionary associated with a display mode. See *CFDictionary Reference*.

You can enumerate the display modes for a display by using its display ID. You can obtain an array of display IDs that correspond to all displays in the system by calling the function `CGGetActiveDisplayList`. The first display in the list is always the main display. The main display is also represented by the constant `kCGDirectMainDisplay`.

These functions also obtain an array of display IDs:

- `CGGetDisplaysWithPoint` obtains the display IDs for online displays whose bounds include a specified point.
- `CGGetDisplaysWithRect` obtains the display IDs for online displays whose bounds include a specified rectangle.
- `CGGetDisplaysWithOpenGLDisplayMask` obtains the display IDs for online displays that correspond to the bits set in an OpenGL display mask.

Typically you use the functions `CGGetDisplaysWithPoint` and `CGGetDisplaysWithRect` when tracking user interactions. You choose which display to capture based on where the user places the pointer. After capturing the display, you can the obtain the supported modes by calling the function `CGDisplayAvailableModes`.

Listing C-1 shows how to switch the last display in a display list into its first mode and then print the height and width of the display. A detailed explanation for each numbered line of code appears following the listing.

Listing C-1 Switching modes for a display in a list

```
#define MAX_DISPLAYS 32

CGDirectDisplayID lastDisplay, displayArray[MAX_DISPLAYS] ;
CGDisplayCount numDisplays;

CFArrayRef displayModeArray;
CFDictionaryRef displayMode;

CFNumberRef number;
long height, width;

CGGetActiveDisplayList (MAX_DISPLAYS, displayArray, &numDisplays);              // 1
lastDisplay = displayArray [numDisplays - 1];                                   // 2
CGDisplayCapture (lastDisplay);                                                 // 3
displayModeArray = CGDisplayAvailableModes (lastDisplay);                       // 4
displayMode = (CFDictionaryRef) CFArrayGetValueAtIndex (displayModeArray, 0);  // 5
CGDisplaySwitchToMode (lastDisplay, displayMode);                               // 6
/* Run the event loop. */
CGReleaseAllDisplays();                                                         // 7
```

Here's what the code does:

1. Gets the array of active displays, which are the ones available for drawing.
2. Gets the display ID of the last display in the array. The array is zero-based.
3. Captures the display associated with the last display in the array.
4. Gets all the display modes for the display.
5. Gets the first display mode for the display. Recall that the display mode is stored as a `CFDictionary` object that contains key-value pairs for the attributes of the display mode.
6. Switches the display mode.
7. Before the application quits, releases all displays.

Quartz Display Services provides simple accessor functions for many properties of the current display mode. For these properties, you don't need to call `CFDictionaryGetValue`. Listing C-2 shows how to obtain the properties of the current mode of every display (up to 32) in the system.

Listing C-2 Getting display properties

```
#define MAX_DISPLAYS 32
```

```
CGDirectDisplayID displayArray [MAX_DISPLAYS];
CGDisplayCount numDisplays;
CFNumberRef number;
CFBoolean booleanValue;
long    height, width, refresh, mode,
        bpp, bps, spp, rowBytes, gui, ioflags;
int     i;

CGGetActiveDisplayList (MAX_DISPLAYS, displayArray, &numDisplays);          // 1
printf ("Displays installed: %d\n", numDisplays);                           // 2
for(i = 0; i < numDisplays; i++) {                                          // 3
    width = CGDisplayPixelsWide (displayArray[i]);
    height = CGDisplayPixelsHigh (displayArray[i]);
    bpp = CGDisplayBitsPerPixel (displayArray[i]);
    bps = CGDisplayBitsPerSample (displayArray[i]);
    spp = CGDisplaySamplesPerPixel (displayArray[i]);
    rowBytes = CGDisplayBytesPerRow (displayArray[i]);
    number = CFDictionaryGetValue (CGDisplayCurrentMode (displayArray[i]),
                                   kCGDisplayMode);
    CFNumberGetValue (number, kCFNumberLongType, &mode);
    number = CFDictionaryGetValue (CGDisplayCurrentMode (displayArray[i]),
                                   kCGDisplayRefreshRate);
    CFNumberGetValue (number, kCFNumberLongType, &refresh);
    booleanValue = CFDictionaryGetValue (CGDisplayCurrentMode(displayArray[i]),
                               kCGDisplayModeUsableForDesktopGUI);
    CFNumberGetValue (number, kCFNumberLongType, &gui);
    number = CFDictionaryGetValue (CGDisplayCurrentMode (displayArray[i]),
                               kCGDisplayIOFlags);
    CFNumberGetValue (number, kCFNumberLongType, &ioflags);
}
```

Here's what the code does:

1. Gets the array of displays.

2. Prints out the number of displays.

3. Gets the properties for the current mode of each display. Note that Quartz Display Services provides several functions that obtain properties. For information about the current display mode, you must use the function `CFDictionaryGetValue`, along with the appropriate key, to retrieve a value from the display mode dictionary returned by the function `CGDisplayCurrentMode`.

Fading the Display

Fading a display ensures a smooth transition when entering full-screen mode, especially when switching display modes. There are two options for fading displays:

- Call the function `CGDisplayFade` to fade all displays connected to the system. See Listing C-3.
- Adjust the display gamma value to fade a single display on a system that has more than one display connected. See Listing C-4 (page 148).

Listing C-3 Fading all displays connected to the system

```
CGDisplayFadeReservationToken token;
CGDisplayErr err;

err = CGAcquireDisplayFadeReservation (kCGMaxDisplayReservationInterval, &token);// 1

if (err == kCGErrorSuccess)
{
    err = CGDisplayFade (token, 0.5, kCGDisplayBlendNormal,
                kCGDisplayBlendSolidColor, 0, 0, 0, true);                  // 2
    // Your code to change the display mode and
    // set the full-screen context.
    err = CGDisplayFade (token, 0.5, kCGDisplayBlendSolidColor,
          kCGDisplayBlendNormal, 0, 0, 0, true);                            // 3
    err = CGReleaseDisplayFadeReservation (token);                          // 4
}
```

Here's what the code does:

1. Reserves the display hardware for the maximum amount of time allowable. Your application must perform this step before it can fade the displays. During this time, your application has exclusive rights to use the fade hardware.

2. Fades displays to black over a duration of 0.5 seconds

3. Performs a fade-in for all displays, from black to normal, over a duration of 0.5 seconds

4. Releases the fade reservation and invalidates the fade token.

When you adjust the gamma value to fade a display, you can't assume that the maximum gamma value is 1.0 because the user might have specified a different maximum value in System Preferences. You need to retrieve the current settings and scale them so that they range from `0` to `1`. Listing C-3 shows how to fade the main display to black and back. Note that the code uses a loop is used to obtain a smooth fade. A more robust technique is to use a timer to ensure a fixed fade duration on different systems. A detailed explanation for each numbered line of code appears following the listing.

Listing C-4 Fading a single display on a system with multiple displays

```
#define kMyFadeTime    1.0
#define kMyFadeSteps    100
#define kMyFadeInterval    ( kMyFadeTime/(double) kMyFadeSteps)

int step;
double fadeValue ;
CGGammaValue    redMin, redMax, redGamma,
                greenMin, greenMax, greenGamma,
                blueMin, blueMax, blueGamma;

CGGetDisplayTransferByFormula (kCGDirectMainDisplay,
            &redMin, &redMax, &redGamma,
            &greenMin, &greenMax, &greenGamma,
            &blueMin, &blueMax, &blueGamma );                               // 1

for ( step = 0; step < kMyFadeSteps; ++step ) {
    fadeValue = 1.0 - (step * kMyFadeInterval);
    CGSetDisplayTransferByFormula (kCGDirectMainDisplay,
            redMin, fadeValue*redMax, redGamma,
            greenMin, fadeValue*greenMax, greenGamma,
            blueMin, fadeValue*blueMax, blueGamma );                        // 2
    usleep( (useconds_t)(1000000.0 * kMyFadeInterval) );                    // 3
}
// Your code to change the display mode and
// attach the context to a full-screen drawable object.
// Run the event loop.
for ( step = 0; step < kMyFadeSteps; ++step ) {
    fadeValue = (step * kMyFadeInterval);
    CGSetDisplayTransferByFormula( kCGDirectMainDisplay,
            redMin, fadeValue*redMax, redGamma,
            greenMin, fadeValue*greenMax, greenGamma,
            blueMin, fadeValue*blueMax, blueGamma );                        // 4
    usleep( (useconds_t)(1000000.0 * kMyFadeInterval) );                    // 5
}
CGDisplayRestoreColorSyncSettings()                                         // 6
```

Here's what the code does:

1. Gets the current coefficients of the gamma transfer formula for a display as the starting gamma values.

2. Fades from the current gamma by setting the color gamma function for the display, specified as the coefficients of the gamma transfer formula. Starts with the current gamma (multiplying by a factor of `1.0`) and ends with black (multiplying by a factor of `0.0`).

3. Suspends processing for a short interval. You either need to use a timer or insert a short delay to achieve a fade effect because the call to change the display gamma returns within 100 microseconds or so, and the actual gamma is applied asynchronously during the next vertical blanking period. Without the delay, you'll get what appears as an instantaneous switch to black rather than a fade effect.

4. Fade from black (multiplying by a factor of `0.0`) back to original gamma (multiplying by a factor of `1.0`).

5. Suspends processing for a short interval to achieve a smooth fade-in effect.

6. Finds and applies all ColorSync settings for all attached displays, restoring the gamma tables to the values in the user's ColorSync display profile. It's a good idea to call this function because the operation performed by the function `CGSetDisplayTransferByFormula` can't reproduce precisely the color correction data for all displays, particularly LCD panels.

Controlling the Pointer

When you use full-screen mode, you may want to hide the pointer, programatically move the pointer, or disassociate mouse movement from pointer position. To hide or show the pointer, use the functions `CGDisplayHideCursor` and `CGDisplayShowCursor`. These functions control the pointer visibility on all displays.

Quartz Display Services provides a convenient function for disassociating mouse movement from pointer position while an application is in the foreground. By passing `false` to the function `CGAssociateMouseAndMouseCursorPosition`, you can prevent mouse movement from changing the pointer position. Pass `true` to reverse the effect. You should also hide the menu bar because clicking it can cause the pointer to become visible again, even after capturing the display.

You can move the pointer programatically by calling the function `CGDisplayMoveCursorToPoint`. This function takes two parameters, a display ID and a point. The location of the point is relative to the display origin (the upper-left corner of the display).

Listing C-5 shows how you would hide and move the cursor on the main display, disassociate the cursor from mouse movement, and restore the cursor and mouse when you are done.

Listing C-5 Controlling the pointer programmatically

```
CGDisplayHideCursor (kCGDirectMainDisplay); //Hide cursor
CGDisplayMoveCursorToPoint (kCGDirectMainDisplay,CGPointZero); //Place at display
 origin
CGAssociateMouseAndMouseCursorPosition (FALSE);
// Perform your application's main loop.
//In the mouse movement notification function, get the motion deltas
CGAssociateMouseAndMouseCursorPosition (TRUE);
CGDisplayShowCursor (kCGDirectMainDisplay);
```

See Also

Quartz Display Services Reference which describes the application programming interface that configures and controls the display hardware.

Glossary

This glossary contains terms that are used specifically for the Apple implementation of OpenGL and a few terms that are common in graphics programming. For definitions of additional OpenGL terms, see OpenGL Programming Guide, by the OpenGL Architecture Review Board.

AGL (Apple Graphics Library) framework The Apple framework for using OpenGL graphics in Mac OS X applications written in the Carbon environment.

aliased Graphics whose edges appear jagged; can be remedied by performing anti-aliasing operations.

anti-aliasing In graphics, a technique used to smooth and soften the jagged (or aliased) edges that are sometimes apparent when graphical objects such as text, line art, and images are drawn.

ARB The OpenGL Architecture Review Board, which is the group that oversees the OpenGL specification and extensions to it.

attach An operation that establishes a connection between two existing objects. Compare with bind.

bind An operation that creates a new object and then establishes a connection between that object and a rendering context.

bitmap A rectangular array of bits.

bitplane A rectangular array of pixels.

buffer A block of memory dedicated to storing a specific kind of data, such as depth values, green color values, stencil index values, color index values, and so forth.

CGL (Core OpenGL) framework The Apple framework for using OpenGL graphics in Mac OS X applications (Cocoa or Carbon) that need low-level access to OpenGL.

clipping An operation that identifies the area of drawing. Anything not in the clipping region is not drawn.

clip coordinates The coordinate system used for view-volume clipping. Clip coordinates are applied after applying the projection matrix and prior to perspective division.

color-lookup table A table of values used to map color indexes into actual color values.

completeness A state that indicates whether a framebuffer object meets all the requirements for drawing.

context A set of OpenGL state variables that affect how drawing is performed for a drawable object attached to that context. Also called a rendering context.

culling Eliminating parts of a scene that can't be seen by the observer.

current context The rendering context to which OpenGL routes commands issued by your application.

current matrix A matrix used by OpenGL to transform coordinates in one system to those of another system, such as the modelview matrix, the perspective matrix, and the texture matrix. GL shading language allows user-defined matrices.

depth In OpenGL, refers to the *z* coordinate and specifies how far a pixel lies from the observer.

depth buffer A block of memory used to store a depth value for each pixel. The depth buffer is used to determine whether or not a pixel can be seen by the observer. Those that are hidden are typically removed.

display list A list of OpenGL commands that have an associated name and that are uploaded to the GPU, preprocessed, and then executed at a later time. Display lists are often used for computing-intensive commands.

double buffering The practice of using a front and back color buffer to achieve smooth animation. The back buffer is not displayed, but swapped with the front buffer.

drawable object In Mac OS X, an object allocated outside of OpenGL that can serve as an OpenGL framebuffer. A drawable object can be any of the following: a window, a view, a pixel buffer, offscreen memory, or a full-screen graphics device. See also framebuffer object

extension A feature of OpenGL that's not part of the OpenGL core API and therefore not guaranteed to be supported by every implementation of OpenGL. The naming conventions used for extensions indicate how widely accepted the extension is. The name of an extension supported only by a specific company includes an abbreviation of the company name. If more then one company adopts the extension, the extension name is changed to include `EXT` instead of a company abbreviation. If the OpenGL Architecture Review Board approves an extension, the extension name changes to include `ARB` instead of `EXT` or a company abbreviation.

eye coordinates The coordinate system with the observer at the origin. Eye coordinates are produced by the modelview matrix and passed to the projection matrix.

fence A token used by the `GL_APPLE_fence` extension to determine whether a given command has completed or not.

filtering A process that modifies an image by combining pixels or texels.

fog An effect achieved by fading colors to a background color based on the distance from the observer. Fog provides depth cues to the observer.

fragment The color and depth values for a single pixel; can also include texture coordinate values. A fragment is the result of rasterizing primitives.

framebuffer The collection of buffers associated with a window or a rendering context.

framebuffer attachable image The rendering destination for a framebuffer object.

framebuffer object An OpenGL extension that allows rendering to a destination other than the usual OpenGL buffers or destinations provided by the windowing system. A framebuffer object (FBO) contains state information for the OpenGL framebuffer and its set of images. A framebuffer object is similar to a drawable object, except that a drawable object is a window-system specific object whereas a framebuffer object is a window-agnostic object. The context that's bound to a framebuffer object can be bound to a window-system-provided drawable object for the purpose of displaying the content associated with the framebuffer object.

frustum Defines the region of space that is seen by the observer and that is warped by perspective division.

FSAA (full scene anti-aliasing) A technique that takes multiple samples at a pixel and combines them with coverage values to arrive at a final fragment.

gamma correction A function that changes color intensity values to correct for the nonlinear response of the eye or of a display.

GLU Graphics library utilities.

GL Graphics library.

GLUT Graphics library utilities toolkit, which is independent of the window system. In Mac OS X, GLUT is implemented on top of Cocoa.

GLX An OpenGL extension that supports using OpenGL within a window provided by the X Window system.

image A rectangular array of pixels.

immediate mode The practice of OpenGL executing commands at the time an application issues them. To prevent commands from being issued immediately, an application can use a display list.

interleaved data Arrays of dissimilar data that are grouped together, such as vertex data and texture coordinates. Interleaving can speed data retrieval.

mipmaps A set of texture maps, provided at various resolutions, whose purpose is to minimize artifacts that can occur when a texture is applied to a geometric primitive whose onscreen resolution doesn't match the source texture map. Mipmapping derives from the latin phrase *multum in parvo*, which means "many things in a small place."

modelview matrix A 4 X 4 matrix used by OpenGL to transforms points, lines, polygons, and positions from object coordinates to eye coordinates.

mutex A mutual exclusion object in a multithreaded application.

NURBS (non-uniform rational B-spline) A methodology use to specify parametric curves and surfaces.

packing Converting pixel color components from a buffer into the format needed by an application.

pbuffer See pixel buffer.

pixel A picture element; the smallest element that the graphics hardware can display on the screen. A pixel is made up of all the bits at the location *x*, *y*, in all the bitplanes in the framebuffer.

pixel buffer A type of drawable object that allows the use of offscreen buffers as sources for OpenGL texturing. Pixel buffers, introduced in Mac OS X v10.3, allow hardware-accelerated rendering to a texture.

pixel depth The number of bits per pixel in a pixel image.

pixel format A format used to store pixel data in memory. The format describes the pixel components (that is, red, blue, green, alpha), the number and order of components, and other relevant information, such as whether a pixel contains stencil and depth values.

primitives The simplest elements in OpenGL—points, lines, polygons, bitmaps, and images.

projection matrix A matrix that OpenGL uses to transform points, lines, polygons, and positions from eye coordinates to clip coordinates.

rasterization The process of converting vertex and pixel data to fragments, each of which corresponds to a pixel in the framebuffer.

renderbuffer A rendering destination for a 2D pixel image, used for generalized offscreen rendering, as defined in the OpenGL specification for the `GL_EXT_framebuffer_object` extension.

renderer A combination of hardware and/or software that OpenGL uses to create an image from a view and a model. The hardware portion of a renderer is associated with a particular display device and supports specific capabilities, such as the ability to support a certain color depth or buffering mode. A renderer that uses only software is called a software renderer and is typically used as a fallback.

rendering context A container for state information.

rendering pipeline The order of operations used by OpenGL to transform pixel and vertex data to an image in the framebuffer.

render-to-texture An operation that draws content directly to a texture target.

RGBA Red, green, blue, and alpha color components.

shader A program that computes surface properties.

shading language A high-level language, accessible in C, used to produce advanced imaging effects. The Apple implementation of OpenGL supports `ARB_shading_language_100`.

stencil buffer Memory used specifically for stencil testing. A stencil test is typically used to identify masking regions, identify solid geometry that needs to be capped, and to overlap translucent polygons.

surface The internal representation of a single buffer that OpenGL actually draws to and reads from. For windowed drawable objects, this surface is what the Mac OS X window server uses to composite OpenGL content on the desktop.

tearing A visual anomaly caused when part of the current frame overwrites previous frame data in the framebuffer before the current frame is fully rendered on the screen.

tessellation An operation that reduces a surface to a mesh of polygons or a curve to a sequence of lines.

texel A texture element used to specify the color to apply to a fragment.

texture Image data used to modify the color of rasterized fragments; can be one-, two-, three-dimensional or be a cube map.

texture mapping The process of applying a texture to a primitive.

texture matrix A 4 x 4 matrix that OpenGL uses to transform texture coordinates to the coordinates that are used for interpolation and texture lookup.

texture object An opaque data structure used to store all data related to a texture. A texture object can include such things as an image, a mipmap, and texture parameters (width, height, internal format, resolution, wrapping modes, and so forth).

vertex A three-dimensional point. A set of vertices specify the geometry of a shape. Vertices can have a number of additional attributes such as colors, and texture coordinates. See vertex array.

vertex array A data structure that stores a block of data that specifies such things as vertex coordinates, texture coordinates, surface normals, RGBA colors, color indices, and edge flags.

virtual screen A combination of hardware, renderer, and pixel format that OpenGL selects as suitable for an imaging task. When the current virtual screen changes, the current renderer typically changes.

Document Revision History

This table describes the changes to *OpenGL Programming Guide for Mac OS X*.

Date	Notes
2008-06-09	Updated the Cocoa OpenGL tutorial and made numerous other minor changes.
	Fixed compilation errors in Listing 5-1 (page 60).
	Added "Getting Decompressed Raw Pixel Data from a Source Image" (page 106).
	Updated links to OpenGL extensions.
	Made several minor edits.
2007-12-04	Corrected minor typographical and technical errors.
	Added "Ensuring that Back Buffer Contents Remain the Same" (page 80).
	Revised "Attributes that are not Recommended" (page 79).
	Corrected a typographical error in Listing 3-1 (page 38).
2007-08-07	Fixed several technical issues.
2007-05-29	Fixed a broken link.
2007-05-17	Fixed a few technical inaccuracies in the code listings.
	Changed attribs to attributes in Listing 7-2 (page 82).
	Fixed drawRect method implementation in "Drawing to a Window or View" (page 27).
2006-12-20	Fixed minor errors.
	Added information concerning the Apple client storage extension. Fixed a typographical error.
2006-11-07	Added information about performance issues and processor queries.
	See "Vertex and Fragment Processing" (page 67).
2006-10-03	Added a section on checking for GPU processing.
	Added "Vertex and Fragment Processing" (page 67).
	Fixed a number of minor typos in the code and in the text.

Date	Notes
2006-09-05	Fixed minor technical problems.
2006-07-24	Made minor technical and typograhical changes throughout.
	Added information to "Surface Drawing Order" (page 66).
	Changed `glCopyTexSubImage` to `glCopyTexSubImage2D` in "Downloading Texture Data" (page 107).
	Made minor improvements to Listing 9-6 (page 107).
	Removed information about 1-D textures.
2006-06-28	Made several minor technical corrections.
	Redirected links to the OpenGL specification for the framebuffer object extension so that they point to the SGI Open Source website, which hosts the most up-to-date version of this specification.
	Removed the logic operation blending entry from Table A-6 (page 138) because this functionality is not available in OpenGL 2.0.
2006-05-23	First version.
	This document replaces *Macintosh OpenGL Programming Guide* and *AGL Programming Guide*.
	This document incorporates information from the following Technical Notes:
	TN2007 "The CGDirectDisplay API"
	TN2014 "Insights on OpenGL"
	TN2080 "Understanding and Detecting OpenGL Functionality"
	TN2093 "OpenGL Performance Optimization: The Basics"
	This document incorporates information from the following Technical Q&As:
	Technical Q&A OGL01 "aglChoosePixelFormat, The Inside Scoop"
	Technical Q&A OGL02 "Correct Setup of an AGLDrawable"
	Technical Q&A QA1158 "glFlush() vs. glFinish()"
	Technical Q&A QA1167 "Using Interface Builder's NSOpenGLView or Custom View objects for an OpenGL application"
	Technical Q&A QA1188 "GetProcAdress and OpenGL Entry Points"
	Technical Q&A QA1209 "Updating OpenGL Contexts"
	Technical Q&A QA1248 "Context Sharing Tips"

Date	Notes
	Technical Q&A QA1268 “Sharpening Full Scene Anti-Aliasing Details”
	Technical Q&A QA1269 “Mac OS X OpenGL Interfaces”
	Technical Q&A QA1325 “Creating an OpenGL texture from an NSView”