OpenGL Programming Guide for Mac

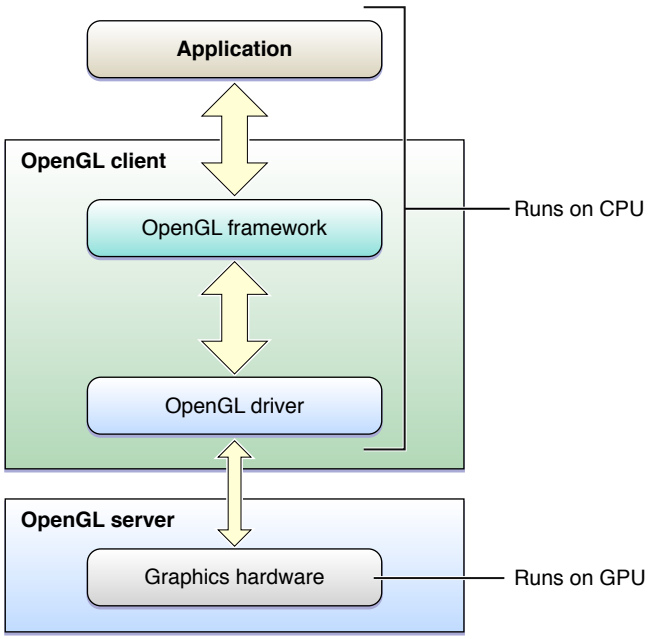
Developer

Contents

Figures, Tables, and Listings

About OpenGL for OS X

OpenGL is an open, cross-platform 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.



At a Glance

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

- **Reliable Implementation**. The OpenGL client-server model abstracts hardware details and guarantees consistent presentation on any compliant hardware and software configuration. Every implementation of OpenGL adheres to the OpenGL specification and must pass a set of conformance tests.
- **Performance**. Applications can harness the considerable power of the graphics hardware to improve rendering speeds and quality.

- **Industry acceptance**. The specification for OpenGL is controlled by the Khronos Group, an industry consortium whose members include many of the major companies in the computer graphics industry, including Apple. In addition to OpenGL for OS X, there are OpenGL implementations for Windows, Linux, Irix, Solaris, and many game consoles.

OpenGL Is a C-based, Platform-Neutral API

Because OpenGL is a C-based API, it is extremely portable and widely supported. As a C API, it integrates seamlessly with Objective-C based Cocoa applications. OpenGL provides functions your application uses to generate 2D or 3D images. Your application presents the rendered images to the screen or copies them back to its own memory.

The OpenGL specification does not provide a windowing layer of its own. It relies on functions defined by OS X to integrate OpenGL drawing with the windowing system. Your application creates an OS X OpenGL **rendering context** and attaches a rendering target to it (known as a **drawable object**). The rendering context manages OpenGL state changes and objects created by calls to the OpenGL API. The drawable object is the final destination for OpenGL drawing commands and is typically associated with a Cocoa window or view.

Relevant Chapters: "OpenGL on the Mac Platform" (page 17)

Different Rendering Destinations Require Different Setup Commands

Depending on whether your application intends to draw OpenGL content to a window, to draw to the entire screen, or to perform offscreen image processing, it takes different steps to create the rendering context and associate it with a drawable object.

Relevant Chapters: "Drawing to a Window or View" (page 35), "Drawing to the Full Screen" (page 50) and "Drawing Offscreen" (page 53)

OpenGL on Macs Exists in a Heterogenous Environment

Macs support different types of graphics processors, each with different rendering capabilities, supporting versions of OpenGL from 1.x through OpenGL 3.2. When creating a rendering context, your application can accept a broad range of renderers or it can restrict itself to devices with specific capabilities. Once you have a context, you can configure how that context executes OpenGL commands.

OpenGL on the Mac is not only a heterogenous environment, but it is also a *dynamic* environment. Users can add or remove displays, or take a laptop running on battery power and plug it into a wall. When the graphics environment on the Mac changes, the renderer associated with the context may change. Your application must handle these changes and adjust how it uses OpenGL.

Relevant Chapters: "Choosing Renderer and Buffer Attributes" (page 64), "Working with Rendering Contexts" (page 72), and "Determining the OpenGL Capabilities Supported by the Renderer" (page 83)

OpenGL Helps Applications Harness the Power of Graphics Processors

Graphics processors are massively parallelized devices optimized for graphics operations. To access that computing power adds additional overhead because data must move from your application to the GPU over slower internal buses. Accessing the same data simultaneously from both your application and OpenGL is usually restricted. To get great performance in your application, you must carefully design your application to feed data and commands to OpenGL so that the graphics hardware runs in parallel with your application. A poorly tuned application may stall either on the CPU or the GPU waiting for the other to finish processing.

When you are ready to optimize your application's performance, Apple provides both general-purpose and OpenGL-specific profiling tools that make it easy to learn where your application spends its time.

Relevant Chapters: "Optimizing OpenGL for High Resolution" (page 44), "OpenGL on the Mac Platform" (page 17),"OpenGL Application Design Strategies" (page 89), "Best Practices for Working with Vertex Data" (page 104), "Best Practices for Working with Texture Data" (page 118), "Customizing the OpenGL Pipeline with Shaders" (page 139), and "Tuning Your OpenGL Application" (page 155)

Concurrency in OpenGL Applications Requires Additional Effort

Many Macs ship with multiple processors or multiple cores, and future hardware is expected to add more of each. Designing applications to take advantage of multiprocessing is critical. OpenGL places additional restrictions on multithreaded applications. If you intend to add concurrency to an OpenGL application, you must ensure that the application does not access the same context from two different threads at the same time.

Relevant Chapters: "Concurrency and OpenGL" (page 148)

Performance Tuning Allows Your Application to Provide an Exceptional User Experience

Once you've improved the performance of your OpenGL application and taken advantage of concurrency, put some of the freed processing power to work for you. Higher resolution textures, detailed models, and more complex lighting and shading algorithms can improve image quality. Full-scene antialiasing on modern graphics hardware can eliminate many of the "jaggies" common on lower resolution images.

Relevant Chapters: "Customizing the OpenGL Pipeline with Shaders" (page 139),"Techniques for Scene Antialiasing" (page 144)

How to Use This Document

If you have never programmed in OpenGL on the Mac, you should read this book in its entirety, starting with "OpenGL on the Mac Platform" (page 17). Critical Mac terminology is defined in that chapter as well as in the "Glossary" (page 179).

If you already have an OpenGL application running on the Mac, but have not yet updated it for OS X v10.7, read "Choosing Renderer and Buffer Attributes" (page 64) to learn how to choose an OpenGL profile for your application.

To find out how to update an existing OpenGL app for high resolution, see "Optimizing OpenGL for High Resolution" (page 44).

Once you have OpenGL content in your application, read "OpenGL Application Design Strategies" (page 89) to learn fundamental patterns for implementing high-performance OpenGL applications, and the chapters that follow to learn how to apply those patterns to specific OpenGL problems.

Important: Although this guide describes how to create rendering contexts that support OpenGL 3.2, most code examples and discussion in the rest of the book describe the earlier legacy versions of OpenGL. See "Updating an Application to Support the OpenGL 3.2 Core Specification" (page 168) for more information on migrating your application to OpenGL 3.2.

Prerequisites

This guide assumes that you have some experience with OpenGL programming, but want to learn how to apply that knowledge to create software for the Mac. Although this guide provides advice on optimizing OpenGL code, it does not provide entry-level information on how to use the OpenGL API. If you are unfamiliar with OpenGL, you should read "OpenGL on the Mac Platform" (page 17) to get an overview of OpenGL on the Mac platform, and then read the following OpenGL programming guide and reference documents:

- OpenGL Programming Guide, by Dave Shreiner and the Khronos OpenGL Working Group; otherwise known as "The Red book."
- *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**).
- OpenGL Reference Pages.

Before reading this document, you should be familiar with Cocoa windows and views as introduced in *Window Programming Guide* and *View Programming Guide*.

See Also

Keep these reference documents handy as you develop your OpenGL program for 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 integrate OpenGL content into a Cocoa application.
- *CGL Reference* describes low-level functions that can be used to create full-screen OpenGL applications.
- *OpenGL Extensions Guide* provides information about OpenGL extensions supported in OS X.

The OpenGL Foundation website, http://www.opengl.org, provides information on OpenGL commands, the Khronos OpenGL Working Group, 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 Specification* provides detailed information on how an OpenGL implementation is expected to handle each OpenGL command.
- *OpenGL Reference* describes the main OpenGL library.
- *OpenGL GLU Reference* describes the OpenGL Utility Library, which contains convenience functions implemented on top of the OpenGL API.
- *OpenGL GLUT Reference* describes the OpenGL Utility Toolkit, a cross-platform windowing API.
- 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).

OpenGL on the Mac Platform

You can tell that Apple has an implementation of OpenGL on its platform by looking at the user interface for many of the applications that are installed with 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 other operations in OS X are due to the use of OpenGL. OpenGL is available to all Macintosh applications.

OpenGL for 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 the underlying graphics hardware. OS X provides a set of application programming interfaces (APIs) that Cocoa applications can use to support OpenGL drawing.

Figure 1-1 OpenGL provides the reflections in iChat

This chapter provides an overview of OpenGL and the interfaces your application uses on the Mac platform to tap into it.

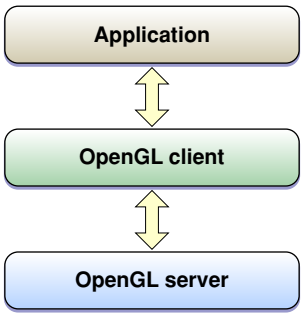
OpenGL Concepts

To understand how OpenGL fits into OS X and your application, you should first understand how OpenGL is designed.

OpenGL Implements a Client-Server Model

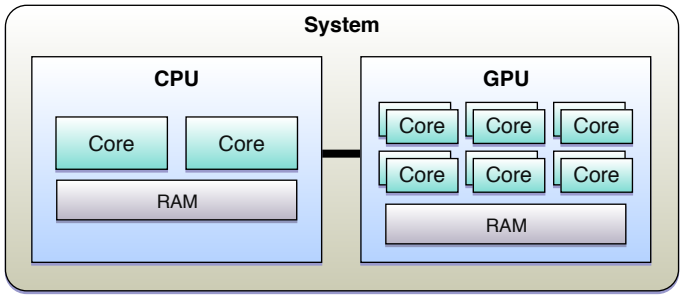
OpenGL uses a client-server model, as shown in Figure 1-2. When your application calls an OpenGL function, it talks to an OpenGL client. The client delivers drawing commands to an OpenGL server. The nature of the client, the server, and the communication path between them is specific to each implementation of OpenGL. For example, the server and clients could be on different computers, or they could be different processes on the same computer.

Figure 1-2 OpenGL client-server model



A client-server model allows the graphics workload to be divided between the client and the server. For example, all Macintosh computers ship with dedicated graphics hardware that is optimized to perform graphics calculations in parallel. Figure 1-3 shows a common arrangement of CPUs and GPUs. With this hardware configuration, the OpenGL client executes on the CPU and the server executes on the GPU.

Figure 1-3 Graphics platform model



OpenGL Commands Can Be Executed Asynchronously

A benefit of the OpenGL client-server model is that the client can return control to the application before the command has finished executing. An OpenGL client may also buffer or delay execution of OpenGL commands. If OpenGL required all commands to complete before returning control to the application, then either the CPU or the GPU would be idle waiting for the other to provide it data, resulting in reduced performance.

Some OpenGL commands implicitly or explicitly require the client to wait until some or all previously submitted commands have completed. OpenGL applications should be designed to reduce the frequency of client-server synchronizations. See "OpenGL Application Design Strategies" (page 89) for more information on how to design your OpenGL application.

OpenGL Commands Are Executed In Order

OpenGL guarantees that commands are executed in the order they are received by OpenGL.

OpenGL Copies Client Data at Call-Time

When an application calls an OpenGL function, the OpenGL client copies any data provided in the parameters before returning control to the application. For example, if a parameter points at an array of vertex data stored in application memory, OpenGL must copy that data before returning. Therefore, an application is free to change memory it owns regardless of calls it makes to OpenGL.

The data that the client copies is often reformatted before it is transmitted to the server. Copying, modifying, and transmitting parameters to the server adds overhead to calling OpenGL. Applications should be designed to minimize copy overhead.

OpenGL Relies on Platform-Specific Libraries For Critical Functionality

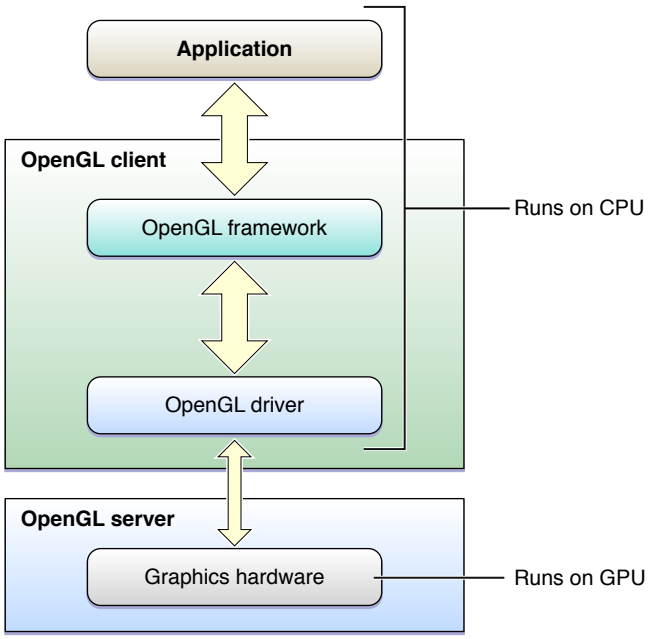
OpenGL provides a rich set of cross-platform drawing commands, but does not define functions to interact with an operating system's graphics subsystem. Instead, OpenGL expects each implementation to define an interface to create rendering contexts and associate them with the graphics subsystem. A rendering context holds all of the data stored in the OpenGL state machine. Allowing multiple contexts allows the state in one machine to be changed by an application without affecting other contexts.

Associating OpenGL with the graphic subsystem usually means allowing OpenGL content to be rendered to a specific window. When content is associated with a window, the implementation creates whatever resources are required to allow OpenGL to render and display images.

OpenGL in OS X

OpenGL in OS X implements the OpenGL client-server model using a common OpenGL framework and plug-in drivers. The framework and driver combine to implement the client portion of OpenGL, as shown in Figure 1-4. Dedicated graphics hardware provides the server. Although this is the common scenario, Apple also provides a software renderer implemented entirely on the CPU.

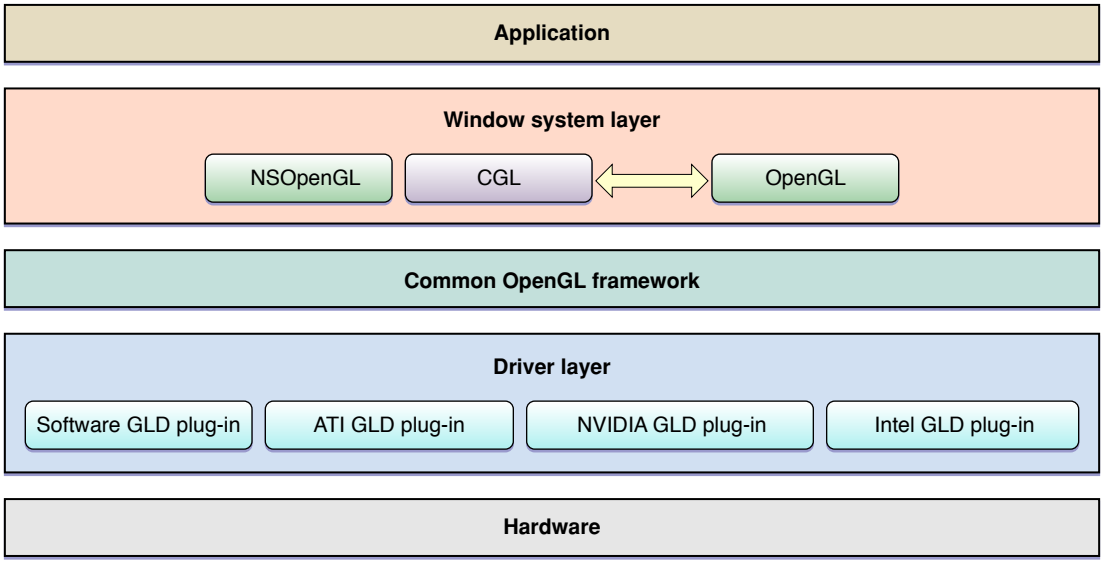
Figure 1-4 MacOS X OpenGL driver model



OS X supports a display space that can include 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 OS X is segmented into well-defined layers: a window system layer,

a framework layer, and a driver layer, as shown in Figure 1-5. 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 1-5 Layers of OpenGL for OS X



The **window system layer** is an OS X–specific layer that your application uses to create OpenGL rendering contexts and associate them with the OS X windowing system. The `NSOpenGL` classes and Core OpenGL (CGL) API also provide some additional controls for how OpenGL operates on that context. See "OpenGL APIs Specific to OS X" (page 22) for more information. Finally, this layer also includes the OpenGL libraries—GL, GLU, and GLUT. (See "Apple-Implemented OpenGL Libraries" (page 23) 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.

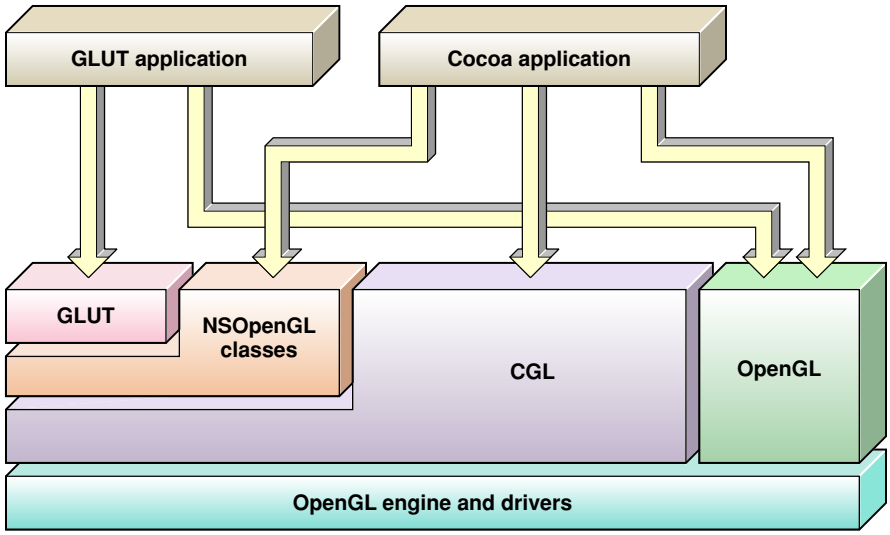
The **driver layer** contains the optional GLD plug-in interface and one or more GLD plug-in drivers, which may have different software and hardware support capabilities. The GLD plug-in interface supports third-party plug-in drivers, allowing third-party hardware vendors to provide drivers optimized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

Accessing OpenGL Within Your Application

The programming interfaces that your application calls fall into two categories—those specific to the Macintosh platform and those defined by the OpenGL Working Group. The Apple-specific programming interfaces are what Cocoa applications use to communicate with the OS X windowing system. These APIs don't create OpenGL content, they manage content, direct it to a drawing destination, and control various aspects of the rendering

operation. Your application calls the OpenGL APIs to create content. OpenGL routines accept vertex, pixel, and texture data and assemble the data to create an image. The final image resides in a framebuffer, which is presented to the user through the windowing-system specific API.

Figure 1-6 The programing interfaces used for OpenGL content



OpenGL APIs Specific to OS X

OS X offers two easy-to-use APIs that are specific to the Macintosh platform: the `NSOpenGL` classes and the CGL API. Throughout this document, these APIs are referred to as the Apple-specific OpenGL APIs.

Cocoa provides many classes specifically for OpenGL:

- The `NSOpenGLContext` class implements a standard OpenGL rendering context.
- The `NSOpenGLPixelFormat` class is used by an application to specify the parameters used to create the OpenGL context.
- The `NSOpenGLView` class is a subclass of `NSView` that uses `NSOpenGLContext` and `NSOpenGLPixelFormat` to display OpenGL content in a view. Applications that subclass `NSOpenGLView` do not need to directly subclass `NSOpenGLPixelFormat` or `NSOpenGLContext`. Applications that need customization or flexibility, can subclass `NSView` and create `NSOpenGLPixelFormat` and `NSOpenGLContext` objects manually.
- The `NSOpenGLLayer` class allows your application to integrate OpenGL drawing with Core Animation.
- The `NSOpenGLPixelBuffer` class provides hardware-accelerated offscreen drawing.

The Core OpenGL API (**CGL**) resides in the OpenGL framework and is used to implement the `NSOpenGL` classes. 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 Reference* provides a complete description of this API.

Apple-Implemented OpenGL Libraries

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 defined 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. 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 document does not show how to use GLUT. The *GLUTBasics* sample project shows you how to get started with GLUT.

GLX is an OpenGL extension that supports using OpenGL within a window provided by the X Window system. X11 for OS X is available as an optional installation. (It's not shown in Figure 1-6 (page 22).) 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 Red book"—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 execute OpenGL commands. The characteristics of the final image depend on the capabilities of the graphics hardware associated with the renderer and the device used to display the image. OS X supports graphics accelerator cards with varying capabilities, as well as a software renderer. It is possible for multiple renderers, each with different capabilities or features, to drive a single set of graphics hardware. To learn how to determine the exact features of a renderer, see "Determining the OpenGL Capabilities Supported by the Renderer" (page 83).

Renderer and Buffer Attributes

Your application uses renderer and buffer attributes to communicate renderer and buffer requirements to OpenGL. The Apple implementation of OpenGL dynamically selects the best renderer for the current rendering task and does so transparently to your application. 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 35) provides a simple example that shows how to use renderer and buffer attributes. "Choosing Renderer and Buffer Attributes" (page 64) explains how to choose renderer and buffer attributes to achieve specific rendering goals.

Pixel Format Objects

A **pixel format** describes the format for pixel data storage in memory. The description includes the number and order of components as well as their names (typically red, blue, green and alpha). It also includes other information, such as whether a pixel contains stencil and depth values. A **pixel format object** is an opaque data structure that holds 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 26) for more information on renderer and display devices.

OpenGL Profiles

OpenGL profiles are new in OS X 10.7. An **OpenGL profile** is a renderer attribute used to request a specific version of the OpenGL specification. When your application provides an OpenGL profile as part of its renderer attributes, it only receives renderers that provide the complete feature set promised by that profile. The render can implement a different version of the OpenGL so long as the version it supplies to your application provides the same functionality that your application requested.

Rendering Contexts

A **rendering context**, or simply *context*, contains OpenGL state information and objects for your application. State variables include such things as drawing color, the viewing and projection transformations, lighting characteristics, and material properties. State variables are set per context. When your application creates OpenGL objects (for example, textures), these are also associated with the rendering context.

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.

Drawable Objects

A **drawable object** refers to an object allocated by the windowing system that can serve as an OpenGL framebuffer. A drawable object is the destination for OpenGL drawing operations. The behavior of drawable objects is not part of the OpenGL specification, but is defined by the OS X windowing system.

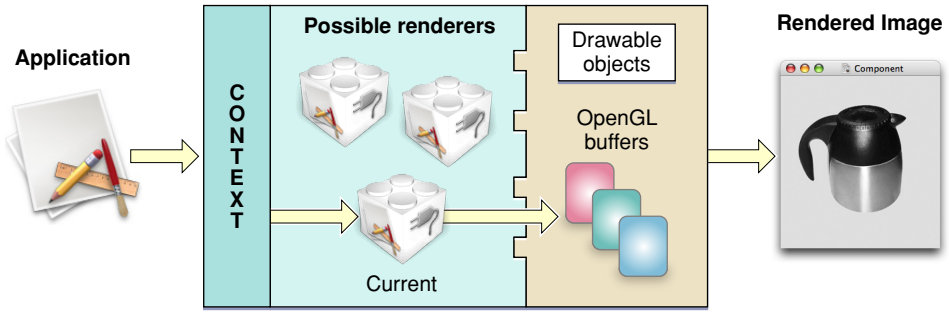
A drawable object can be any of the following: a Cocoa view, offscreen memory, a full-screen graphics device, or a pixel buffer.

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 and then use the pixel buffer as a texture for other OpenGL commands. Although pixel buffers are supported on Apple's implementation of OpenGL, Apple recommends you use framebuffer objects instead. See "Drawing Offscreen" (page 53) for more information on offscreen rendering.

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. Apple's 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 1-7. 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 OS X " (page 31).)

Figure 1-7 Data flow through OpenGL



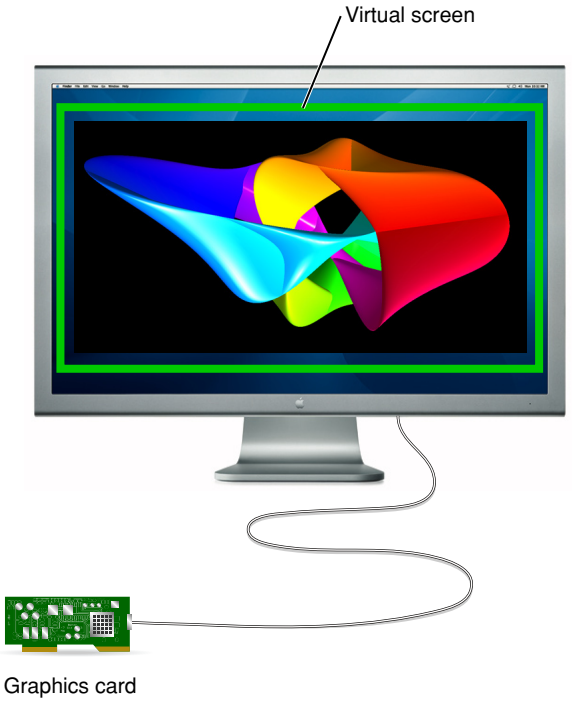
Virtual Screens

The characteristics and quality of the OpenGL content that the user sees depend 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 OpenGL application running on OS X.

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. 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 "Choosing Renderer and Buffer Attributes" (page 64).

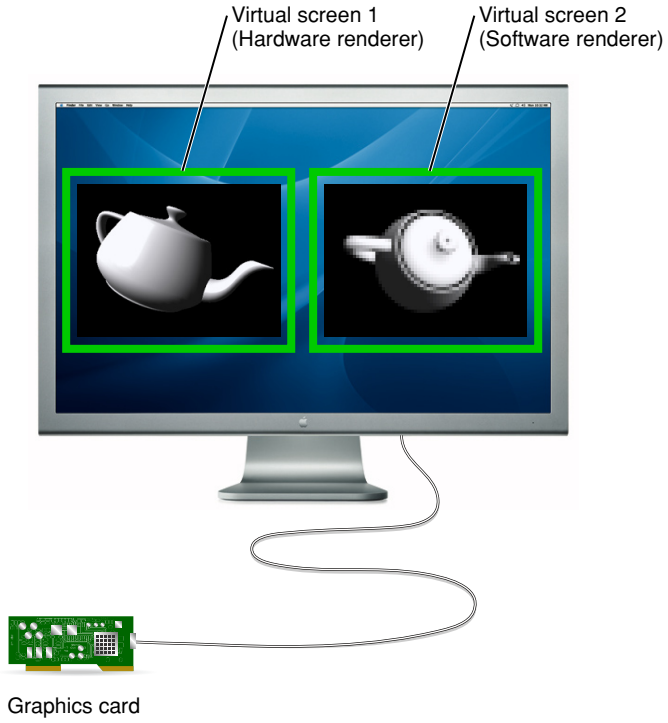
The green rectangle around the OpenGL image in Figure 1-8 surrounds a virtual screen for a system with one graphics card and one display. Note that a virtual screen is not the physical display, which is why the green rectangle is drawn around the application window that shows the OpenGL content. In this case, it is the renderer provided by the graphics card combined with the characteristics of the display.

Figure 1-8 A virtual screen displays what the user sees



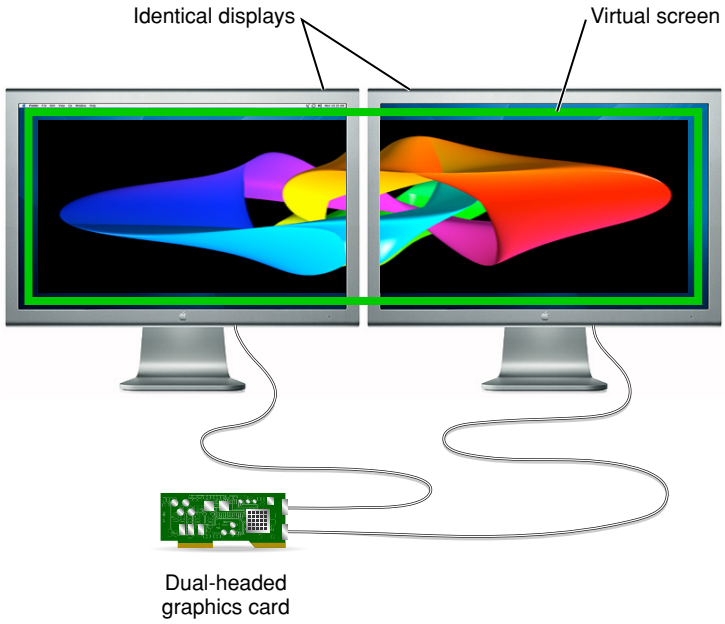
Because a virtual screen is not simply the physical display, a system with one display can use more than one virtual screen at a time, as shown in Figure 1-9. The green rectangles are drawn to point out each virtual screen. Imagine that the virtual screen on the right side uses a software-only renderer and that the one on the left uses a hardware-dependent renderer. Although this is a contrived example, it illustrates the point.

Figure 1-9 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 1-10 is drawn around a virtual screen that spans two physical displays. In this case, the same graphics hardware drives a pair of identical displays. A mirrored display also has a single virtual screen associated with multiple physical displays.

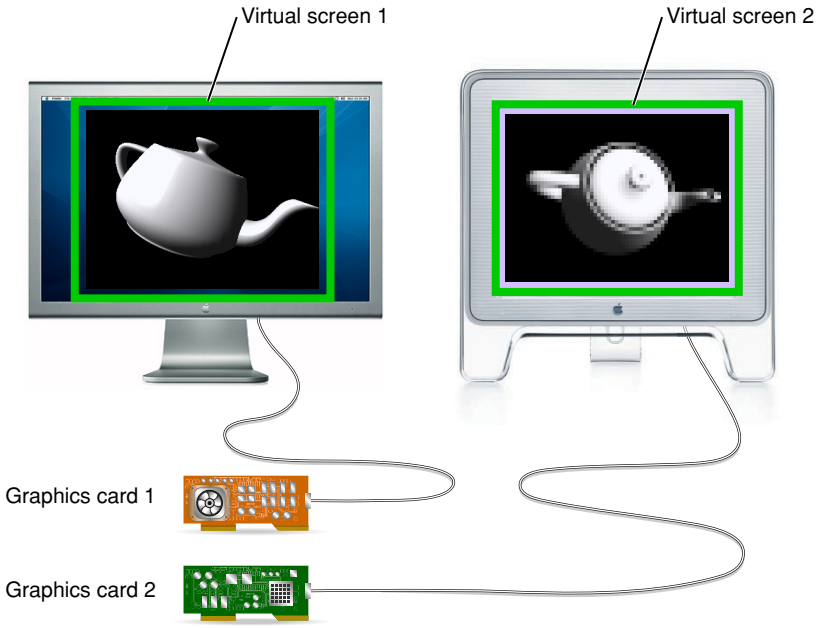
Figure 1-10 A virtual screen can represent more than one physical screen



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 1-10 (page 29), dragging between displays preserves the same virtual screen. However, Figure 1-11 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 1-11, the image quality can be quite different.

Figure 1-11 Two virtual screens and two graphics cards



OpenGL for 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 "Working with Rendering Contexts" (page 72).

Offline Renderer

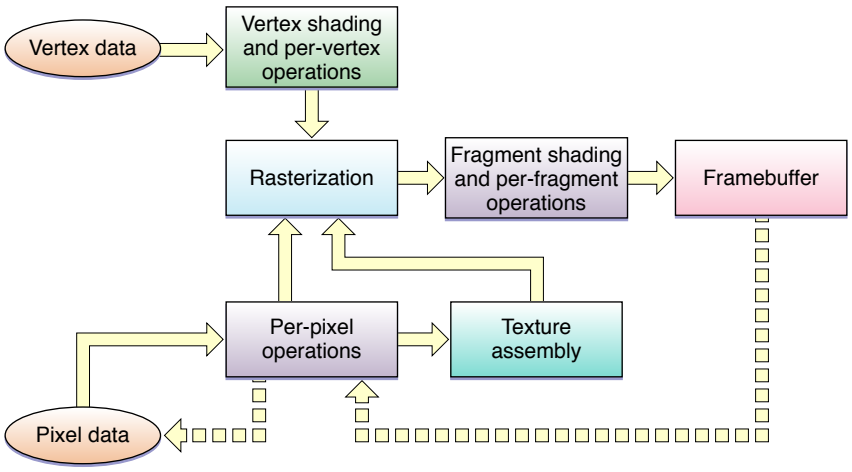
An offline renderer is one that is not currently associated with a display. For example, a graphics processor might be powered down to conserve power, or there might not be a display hooked up to the graphics card. Offline renderers are not normally visible to your application, but your application can enable them by adding the appropriate renderer attribute. Taking advantage of offline renderers is useful because it gives the user a seamless experience when they plug in or remove displays.

For more information about configuring a context to see offline renderers, see "Choosing Renderer and Buffer Attributes" (page 64). To enable your application to switch to a renderer when a display is attached, see "Update the Rendering Context When the Renderer or Geometry Changes" (page 72).

Running an OpenGL Program in OS X

Figure 1-12 shows the flow of data in an OpenGL program, regardless of the platform that the program runs on.

Figure 1-12 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 antialiasing 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 1-12 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 applications provide data to the OpenGL for processing? A Mac 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 24).)
- 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 24) and "Virtual Screens" (page 26).)
- 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 25).)
- 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 25).)
- 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.
- Flush the contents of the rendering context. This causes previously submitted commands to be rendered to the drawable object and displays them to the user.

The tasks described in the first five bullet items are platform-specific. "Drawing to a Window or View" (page 35) 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.

Making Great OpenGL Applications on the Macintosh

OpenGL lets you create applications with outstanding graphics performance as well as a great user experience—but neither of these things come for free. Your application performs best when it works with OpenGL rather than against it. With that in mind, here are guidelines you should follow to create high-performance, future-looking OpenGL applications:

- Ensure your application runs successfully with offline renderers and multiple graphics cards.

  Apple ships many sophisticated hardware configurations. Your application should handle renderer changes seamlessly. You should test your application on a Mac with multiple graphics processors and include tests for attaching and removing displays. For more information on how to implement hot plugging correctly, see "Working with Rendering Contexts" (page 72)

- Avoid finishing and flushing operations.

  Pay particular attention to OpenGL functions that force previously submitted commands to complete. Synchronizing the graphics hardware to the CPU may result in dramatically lower performance. Performance is covered in detail in "OpenGL Application Design Strategies" (page 89).

- Use multithreading to improve the performance of your OpenGL application.

  Many Macs support multiple simultaneous threads of execution. Your application should take advantage of concurrency. Well-behaved applications can take advantage of concurrency in just a few line of code. See "Concurrency and OpenGL" (page 148).

- Use buffer objects to manage your data.

  Vertex buffer objects (VBOs) allow OpenGL to manage your application's vertex data. Using vertex buffer objects gives OpenGL more opportunities to cache vertex data in a format that is friendly to the graphics hardware, improving application performance. For more information see "Best Practices for Working with Vertex Data" (page 104).

  Similarly, pixel buffer objects (PBOs) should be used to manage your image data. See "Best Practices for Working with Texture Data" (page 118)

- Use framebuffer objects (FBOs) when you need to render to offscreen memory.

  Framebuffer objects allow your application to create offscreen rendering targets without many of the limitations of platform-dependent interfaces. See "Rendering to a Framebuffer Object" (page 53).

- Generate objects before binding them.

  Earlier version of OpenGL allowed your applications to create its own object names before binding them. However, you should avoid this. Always use the OpenGL API to generate object names.

- Migrate your OpenGL Applications to OpenGL 3.2

The OpenGL 3.2 Core profile provides a clean break from earlier versions of OpenGL in favor of a simpler shader-based pipeline. For better compatibility with future hardware and OS X releases, migrate your applications away from legacy versions of OpenGL. Many of the recommendations listed above are required when your application uses OpenGL 3.2.

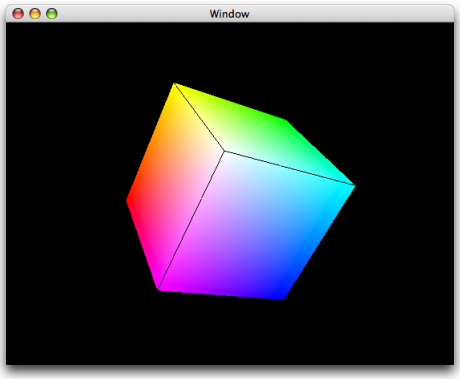
- Harness the power of Apple's development tools.

  Apple provides many tools that help create OpenGL applications and analyze and tune their performance. Learning how to use these tools helps you create fast, reliable applications. "Tuning Your OpenGL Application" (page 155) describes many of these tools.

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.

Figure 2-1 OpenGL content in a Cocoa view



This chapter shows how to display OpenGL drawing onscreen using the APIs provided by OS X. (This chapter does not show how to use GLUT.) The first section describes the overall approach to drawing onscreen and provides an overview of the functions and methods used by each API.

General Approach

To draw your content to a view or a layer, your application uses the `NSOpenGL` classes from within the Cocoa application framework. While the CGL API is used by your applications only to create full-screen content, every `NSOpenGLContext` object contains a CGL context object. This object can be retrieved from the `NSOpenGLContext` when your application needs to reference it directly. To show the similarities between the two, this chapter discusses both the `NSOpenGL` classes and the CGL API.

To draw OpenGL content to a window or view using the `NSOpenGL` classes, 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 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 `kCGLPFAAllRenderers` constant in the CGL 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 26).)

   There may be situations for which you want to ensure that your program uses a specific renderer. "Choosing Renderer and Buffer Attributes" (page 64) discusses how to set up an attributes array that guarantees the system passes back a pixel format object that uses only that renderer.

   If an error occurs, your application may receive a `NULL` pixel format object. Your application must handle this condition.

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. For a windowed context, this is typically a Cocoa view.

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 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. If your application has modest drawing requirements, then you can use the `NSOpenGLView` class. See "Drawing to an NSOpenGLView Class: A Tutorial" (page 37).

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 need to set up a custom `NSView` class. See "Drawing OpenGL Content to a Custom View" (page 40).

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. Create a Cocoa application project named Golden Triangle.
2. Add the OpenGL framework to your project.
3. Add a new file to your project using the Objective-C class template. Name the file `MyOpenGLView.m` and create a header file for it.
4. Open the `MyOpenGLView.h` file and modify the file so that it looks like the code shown in Listing 2-1 to declare the interface.

Listing 2-1 The interface for `MyOpenGLView`

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


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

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

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

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

@implementation MyOpenGLView
@end
```

7. 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();
}
```

8. 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, add the code shown in Listing 2-4. This code draws a 2D, 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

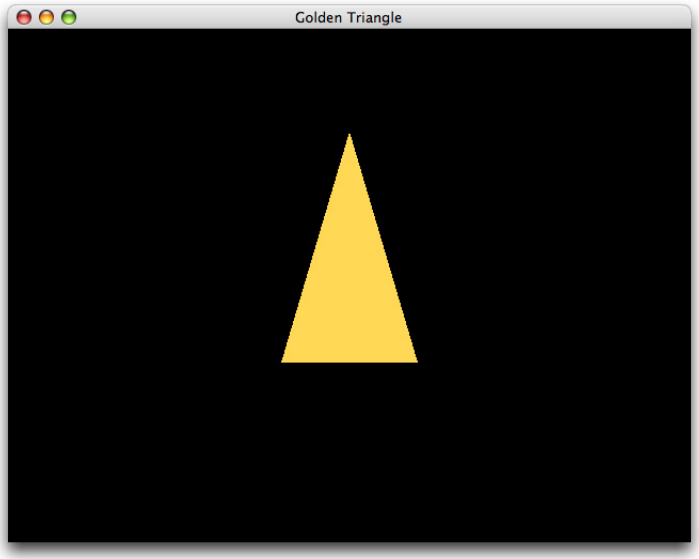
```
static void drawAnObject ()
{
    glColor3f(1.0f, 0.85f, 0.35f);
    glBegin(GL_TRIANGLES);
    {
```

```
        glVertex3f(  0.0,  0.6, 0.0);
        glVertex3f( -0.2, -0.3, 0.0);
        glVertex3f(  0.2, -0.3 ,0.0);
    }
    glEnd();
}
```

9. Open the `MainMenu.xib` in Interface Builder.
10. Change the window's title to `Golden Triangle`.
11. Drag an `NSOpenGLView` object from the Library to the window. Resize the view to fit the window.
12. Change the class of this object to `MyOpenGLView`.
13. 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 need to set them programmatically.
14. Build and run your application. 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:

- Replace the immediate-mode drawing commands with commands that persist your vertex data inside OpenGL. See "OpenGL Application Design Strategies" (page 89).

- 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 needs to provide a timer or support user interaction. For more information about creating animation in your OpenGL application, see "Synchronize with the Screen Refresh Rate" (page 96).

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*.

When you subclass the `NSView` class to create a custom view for OpenGL drawing, you 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 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.

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];
}
```

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 2-7 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];
}
```

Optimizing OpenGL for High Resolution

OpenGL is a pixel-based API so the `NSOpenGLView` class does not provide high-resolution surfaces by default. Because adding more pixels to renderbuffers has performance implications, you must explicitly opt in to support high-resolution screens. It's easy to enable high-resolution backing for an OpenGL view. When you do, you'll want to perform a few additional tasks to ensure the best possible high-resolution experience for your users.

Enable High-Resolution Backing for an OpenGL View

You can opt in to high resolution by calling the method `setWantsBestResolutionOpenGLSurface:` when you initialize the view, and supplying `YES` as an argument:

```
[self  setWantsBestResolutionOpenGLSurface:YES];
```

If you don't opt in, the system magnifies the rendered results.

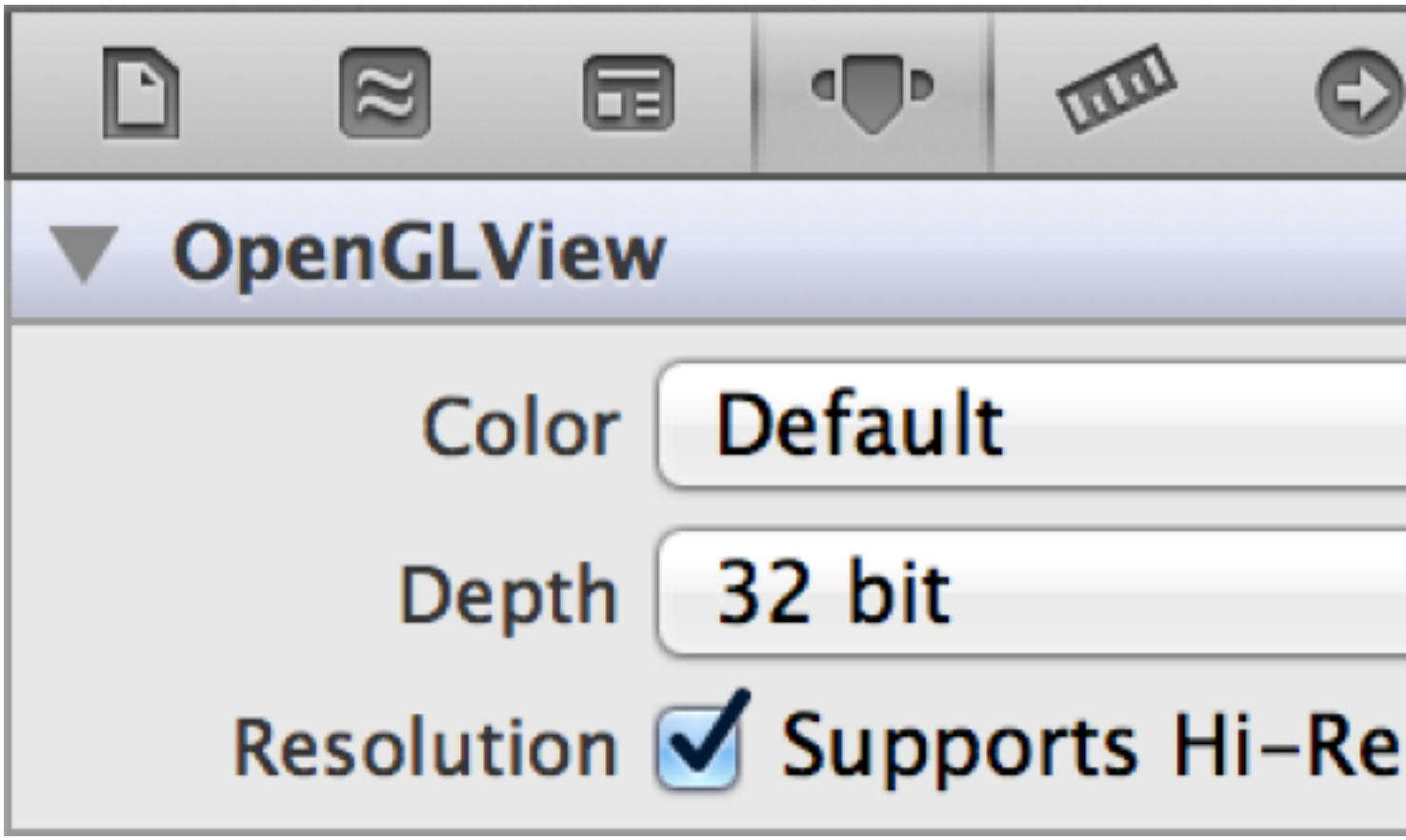
The `wantsBestResolutionOpenGLSurface` property is relevant only for views to which an `NSOpenGLContext` object is bound. Its value does not affect the behavior of other views. For compatibility, `wantsBestResolutionOpenGLSurface` defaults to `NO`, providing a 1-pixel-per-point framebuffer regardless of the backing scale factor for the display the view occupies. Setting this property to `YES` for a given view causes AppKit to allocate a higher-resolution framebuffer when appropriate for the backing scale factor and target display.

To function correctly with `wantsBestResolutionOpenGLSurface` set to `YES`, a view must perform correct conversions between view units (points) and pixel units as needed. For example, the common practice of passing the width and height of `[self bounds]` to `glViewport()` will yield incorrect results at high resolution, because the parameters passed to the `glViewport()` function must be in pixels. As a result, you'll get only partial instead of complete coverage of the render surface. Instead, use the backing store bounds:

```
[self convertRectToBacking:[self bounds]];
```

You can also opt in to high resolution by enabling the Supports Hi-Res Backing setting for the OpenGL view in Xcode, as shown in Figure 3-1.

Figure 3-1 Enabling high-resolution backing for an OpenGL view



Set Up the Viewport to Support High Resolution

The viewport dimensions are in pixels relative to the OpenGL surface. Pass the width and height to `glViewPort` and use 0,0 for the `x` and `y` offsets. Listing 3-1 shows how to get the view dimensions in pixels and take the backing store size into account.

Listing 3-1 Setting up the viewport for drawing

```
- (void)drawRect:(NSRect)rect   // NSOpenGLView subclass
{
    // Get view dimensions in pixels
    NSRect backingBounds = [self convertRectToBacking:[self bounds]];
```

```
    GLsizei backingPixelWidth  = (GLsizei)(backingBounds.size.width),
            backingPixelHeight = (GLsizei)(backingBounds.size.height);

    // Set viewport
    glViewport(0, 0, backingPixelWidth, backingPixelHeight);

    // draw…
}
```

You don't need to perform rendering in pixels, but you do need to be aware of the coordinate system you want to render in. For example, if you want to render in points, this code will work:

```
glOrtho(NSWidth(bounds), NSHeight(bounds),...)
```

Adjust Model and Texture Assets

If you opt in to high-resolution drawing, you also need to adjust the model and texture assets of your app. For example, when running on a high-resolution display, you might want to choose larger models and more detailed textures to take advantage of the increased number of pixels. Conversely, on a standard-resolution display, you can continue to use smaller models and textures.

If you create and cache textures when you initialize your app, you might want to consider a strategy that accommodates changing the texture based on the resolution of the display.

Check for Calls Defined in Pixel Dimensions

These functions use pixel dimensions:

- `glViewport (GLint x, GLint y, GLsizei width, GLsizei height)`
- `glScissor (GLint x, GLint y, GLsizei width, GLsizei height)`
- `glReadPixels (GLint x, GLint y, GLsizei width, GLsizei height, ...)`
- `glLineWidth (GLfloat width)`
- `glRenderbufferStorage (..., GLsizei width, GLsizei height)`
- `glTexImage2D (..., GLsizei width, GLsizei height, ...)`

Tune OpenGL Performance for High Resolution

Performance is an important factor when determining whether to support high-resolution content. The quadrupling of pixels that occurs when you opt in to high resolution requires more work by the fragment processor. If your app performs many per-fragment calculations, the increase in pixels might reduce its frame rate. If your app runs significantly slower at high resolution, consider the following options:

- Optimize fragment shader performance. (See "Tuning Your OpenGL Application" (page 155).)
- Choose a simpler algorithm to implement in your fragment shader. This reduces the quality of each individual pixel to allow for rendering the overall image at a higher resolution.
- Use a fractional scale factor between 1.0 and 2.0. A scale factor of 1.5 provides better quality than a scale factor of 1.0, but it needs to fill fewer pixels than an image scaled to 2.0.
- Multisampling antialiasing can be costly with marginal benefit at high resolution. If you are using it, you might want to reconsider.

The best solution depends on the needs of your OpenGL app; you should test more than one of these options and choose the approach that provides the best balance between performance and image quality.

Use a Layer-Backed View to Overlay Text on OpenGL Content

When you draw standard controls and Cocoa text to a layer-backed view, the system handles scaling the contents of that layer for you. You need to perform only a few steps to set and use the layer. Compare the controls and text in standard and high resolutions, as shown in Figure 3-2. The text looks the same on both without any additional work on your part.

Figure 3-2 A text overlay scales automatically for standard resolution (left) and high resolution (right)

To set up a layer-backed view for OpenGL content

1. Set the `wantsLayer` property of your `NSOpenGLView` subclass to `YES`.

   Enabling the `wantsLayer` property of an `NSOpenGLView` object activates layer-backed rendering of the OpenGL view. Drawing a layer-backed OpenGL view proceeds mostly normally through the view's `drawRect:` method. The layer-backed rendering mode uses its own `NSOpenGLContext` object, which is distinct from the `NSOpenGLContext` that the view uses for drawing in non-layer-backed mode.

   AppKit automatically creates this context and assigns it to the view by invoking the `setOpenGLContext:` method. The view's `openGLContext` accessor will return the layer-backed OpenGL context (rather than the non-layer-backed context) while the view is operating in layer-backed mode.

2. Create the layer content either as a XIB file or programmatically.

   The controls shown in Figure 3-2 were created in a XIB file by subclassing `NSBox` and using static text with a variety of standard controls. Using this approach allows the `NSBox` subclass to ignore mouse events while still allowing the user to interact with the OpenGL content.

3. Add the layer to the OpenGL view by calling the `addSublayer:` method.

Use an Application Window for Fullscreen Operation

For the best user experience, if you want your app to run full screen, create a window that covers the entire screen. This approach offers two advantages:

- The system provides optimized context performance.
- Users will be able to see critical system dialogs above your content.

You should avoid changing the display mode of the system.

Convert the Coordinate Space When Hit Testing

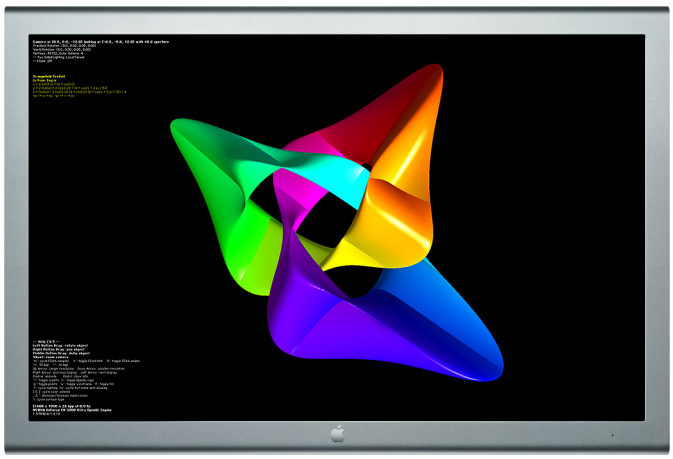
Always convert window event coordinates when performing hit testing in OpenGL. The `locationInWindow` method of the `NSEvent` class returns the receiver's location in the base coordinate system of the window. You then need to call the `convertPoint:fromView:` method to get the local coordinates for the OpenGL view.

```
NSPoint aPoint = [theEvent locationInWindow];
NSPoint localPoint = [myOpenGLView convertPoint:aPoint fromView:nil];
```

Drawing to the Full Screen

In OS X, you have the option to draw to the entire screen. This is a common scenario for games and other immersive applications, and OS X applies additional optimizations to improve the performance of full-screen contexts.

Figure 4-1 Drawing OpenGL content to the full screen



OS X v10.6 and later automatically optimize the performance of screen-sized windows, allowing your application to take complete advantage of the window server environment on OS X. For example, critical operating system dialogs may be displayed over your content when necessary.

For information about high-resolution and full-screen drawing, see "Use an Application Window for Fullscreen Operation" (page 49).

Creating a Full-Screen Application

Creating a full-screen context is very simple. Your application should follow these steps:

1. Create a screen-sized window on the display you want to take over:

```
NSRect mainDisplayRect = [[NSScreen mainScreen] frame];
```

```
NSWindow *fullScreenWindow = [[NSWindow alloc] initWithContentRect:
mainDisplayRect styleMask:NSBorderlessWindowMask
backing:NSBackingStoreBuffered defer:YES];
```

2. Set the window level to be above the menu bar.:

```
[fullScreenWindow setLevel:NSMainMenuWindowLevel+1];
```

3. Perform any other window configuration you desire:

```
[fullScreenWindow setOpaque:YES];
[fullScreenWindow setHidesOnDeactivate:YES];
```

4. Create a view with a double-buffered OpenGL context and attach it to the window:

```
NSOpenGLPixelFormatAttribute attrs[] =
{
    NSOpenGLPFADoubleBuffer,
    0
};
NSOpenGLPixelFormat* pixelFormat = [[NSOpenGLPixelFormat alloc]
initWithAttributes:attrs];


NSRect viewRect = NSMakeRect(0.0, 0.0, mainDisplayRect.size.width,
mainDisplayRect.size.height);
MyOpenGLView *fullScreenView = [[MyOpenGLView alloc] initWithFrame:viewRect
 pixelFormat: pixelFormat];
[fullScreenWindow setContentView: fullScreenView];
```

5. Show the window:

```
[fullScreenWindow makeKeyAndOrderFront:self];
```

That's all you need to do. Your content is in a window that is above most other content, but because it is in a window, OS X can still show critical UI elements above your content when necessary (such as error dialogs). When there is no content above your full-screen window, OS X automatically attempts to optimize this context's performance. For example, when your application calls `flushBuffer` on the `NSOpenGLContext` object, the

system may swap the buffers rather than copying the contents of the back buffer to the front buffer. These performance optimizations are not applied when your application adds the `NSOpenGLPFABackingStore` attribute to the context. Because the system may choose to swap the buffers rather than copy them, your application must completely redraw the scene after every call to `flushBuffer`. For more information on `NSOpenGLPFABackingStore`, see "Ensuring That Back Buffer Contents Remain the Same" (page 66).

Avoid changing the display resolution from that chosen by the user. If your application needs to render data at a lower resolution for performance reasons, you can explicitly create a back buffer at the desired resolution and allow OpenGL to scale those results to the display. See "Controlling the Back Buffer Size" (page 78).

Drawing Offscreen

OpenGL applications may want to use OpenGL to render images without actually displaying them to the user. For example, an image processing application might render the image, then copy that image back to the application and save it to disk. Another useful strategy is to create intermediate images that are used later to render additional content. For example, your application might want to render an image and use it as a texture in a future rendering pass. For best performance, offscreen targets should be managed by OpenGL. Having OpenGL manage offscreen targets allows you to avoid copying pixel data back to your application, except when this is absolutely necessary.

OS X offers two useful options for creating offscreen rendering targets:

- **Framebuffer objects**. The OpenGL framebuffer extension allows your application to create fully supported offscreen OpenGL framebuffers. Framebuffer objects are fully supported as a cross-platform extension, so they are the preferred way to create offscreen rendering targets. See "Rendering to a Framebuffer Object" (page 53).
- **Pixel buffer drawable objects**. Pixel buffer drawable objects are an Apple-specific technology for creating an offscreen target. Each of the Apple-specific OpenGL APIs provides routines to create an offscreen hardware accelerated pixel buffer. Pixel buffers are recommended for use only when framebuffer objects are not available. See "Rendering to a Pixel Buffer" (page 60).

Rendering to a Framebuffer Object

The OpenGL framebuffer extension (`GL_EXT_framebuffer_object`) allows applications to create offscreen rendering targets from within OpenGL. OpenGL manages the memory for these framebuffers.

> **Note:** Extensions are available on a per-renderer basis. Before you use framebuffer objects you must check each renderer to make sure that it supports the extension. See "Detecting Functionality" (page 83) for more information.

A **framebuffer object** (FBO) is similar to a drawable object, except 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. After drawing to a framebuffer object, it is straightforward to read the pixel data to the application, or to use it as source data for other OpenGL commands.

Framebuffer objects offer a number of benefits:

- 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 are associated with 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. Because all commands are rendered by a single context, no additional serialization is required.
- They can share depth buffers; pixel buffers cannot.
- You can use them for 2D pixel images and 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 as the destination for rendering operations and as a 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 other requirements.

Prior to using framebuffer objects, read 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 as either a texture or an image. 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.

Using a Framebuffer Object as a Texture

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 Renderer" (page 83).

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 you can attach to the framebuffer object(`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).

Whenever a framebuffer object is bound, drawing commands are directed to it instead of being directed to the drawable associated with the rendering context.

5. Generate a texture name.

```
void glGenTextures(GLsizei n, GLuint *textures);
```

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

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

6. Bind the texture name to a texture target.

```
void glBindTexture(GLenum target, GLuint texture);
```

`target` is the type of texture to bind.

`texture` is the texture name you just created.

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. If you render to a framebuffer object texture attachment with that same texture currently bound and enabled, the result is undefined.
12. To draw the contents of the texture to a window, 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 5-1 shows code that performs these tasks. This example creates and draws to a single framebuffer object.

Listing 5-1 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);
```

Using a Framebuffer Object as an Image

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.
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`.
4. Bind the framebuffer object name to a framebuffer target by calling the function `glBindFramebufferEXT`.
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. Render content to the renderbuffer.

11. To access the contents of the renderbuffer object, bind the framebuffer object and then use OpenGL functions such as `glReadPixels` or `glCopyTexImage2D`.

12. Delete the framebuffer object with its renderbuffer attachment.

Listing 5-2 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 5-2 Setting up a renderbuffer for drawing images

```
GLuint framebuffer, renderbuffer;
GLenum status;
// Set the width and height appropriately for your image
GLuint imageWidth = 1024,
       imageHeight = 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, imageWidth, imageHeight);
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
// ...
//Your code to use the contents
// ...
// Make the window the target
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
// Delete the renderbuffer attachment
glDeleteRenderbuffersEXT(1, &renderbuffer);
```

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.

Important: Pixel buffers are deprecated starting with OS X v10.7 and are not supported by the OpenGL 3.2 Core profile; use framebuffer objects instead.

You must create a rendering context for each pixel buffer. For example, if you want to use a pixel buffer as a texture source, you create one rendering context attached to the pixel buffer and a second context attached to a window or view.

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 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 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 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:` 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 Red book"—*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 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.

Choosing Renderer and Buffer 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 renderers that meet the requirements defined by those 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 may miss out on future capabilities or it may fail to return renderers on some 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 specify attributes that make your OpenGL content look and run its best, you also need to consider whether your application should run on a less-capable system with less speed or detail. If tradeoffs are acceptable, you need to set the attributes accordingly.

OpenGL Profiles (OS X v10.7)

When your application is running on OS X v10.7, it should always include the `kCGLPFAOpenGLProfile` attribute, followed by a constant for the profile whose functionality your application requires. A profile affects different parts of OpenGL in OS X:

- A profile requires that a specific version of the OpenGL API must provided by the renderer. The renderer may implement a different version of the OpenGL specification only if that version implements the same functions and constants required by the profile; typically, this means a renderer that supports a later version of the OpenGL specification that did not remove or alter behavior specified in the version of the OpenGL specification your application requested.
- The profile alters the list of OpenGL extensions returned by the renderer. For example, extensions whose functionality is provided by the version of the OpenGL specification you requested are not also returned in the list of extensions.
- On OS X, the profile affects what other renderer and buffer attributes may be included in the attributes list.

Follow these guidelines to choose an OpenGL profile:

- If you are developing a new OS X v10.7 application, implement your OpenGL functionality using the OpenGL 3.2 Core profile; include the `kCGLOGLPVersion_3_2_Core` constant.

  The OpenGL 3.2 core profile is defined by Khronos and explicitly removes removes deprecated features described in earlier versions of the OpenGL specification; further the core profile prohibits these functions from being added back into OpenGL using extensions. OpenGL 3.2 core represents a complete break from the fixed function pipeline of OpenGL 1.x in favor of a clean, lean shader-based pipeline.

  When you use the OpenGL 3.2 Core profile on OS X, legacy extensions are removed wherever their functionality is already provided by OpenGL 3.2. Further, pixel and buffer format attributes that are marked as deprecated may not be used in conjunction with the OpenGL 3.2 core profile.

- If you are updating an existing OS X application, include the `kCGLOGLPVersion_Legacy` constant.

  The legacy profile provides the same functionality found in earlier versions of OS X, with no changes. It continues to support older extensions as well as deprecated pixel and buffer format attributes. No new functionality will be added to the legacy profile in future versions of OS X.

- If you want to use OpenGL 3.2 in your application, but also want to support earlier versions of OS X or Macs that lack hardware support for OpenGL 3.2, you must implement multiple OpenGL rendering options in your application. On OS X v10.7, your application should first test to see if OpenGL 3.2 is supported. If OpenGL 3.2 is supported, create a context and provide it to your OpenGL 3.2 rendering path. Otherwise, search for a pixel format using the legacy profile instead.

For more information on migrating an application to OpenGL 3.2, see "Updating an Application to Support the OpenGL 3.2 Core Specification" (page 168).

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` or `kCGLPFAMinimumPolicy`).
- To choose color, depth, and accumulation buffers that are closest to a size you specify, use the closest policy attribute (`NSOpenGLPFAClosestPolicy` or `kCGLPFAClosestPolicy`).
- To choose the largest color, depth, and accumulation buffers available, use the maximum policy attribute (`NSOpenGLPFAMaximumPolicy` or `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.

Ensuring That Back Buffer Contents Remain the Same

When your application uses a double-buffered context, it displays the rendered image by calling a function to flush the image to the screen— the `NSOpenGLContext` class's `flushBuffer` method or the CGL function `CGLFlushDrawable`. When the image is displayed, the contents of the back buffer are not preserved. The next time your application wants to update the back buffer, it must completely redraw the scene.

Your application can add a backing store attribute (`NSOpenGLPFABackingStore` or `kCGLPFABackingStore`) to preserve the contents of the buffer after the back buffer is flushed. Adding this attribute disables some optimizations that the system can perform, which may impact the performance of your application.

Ensuring a Valid Pixel Format Object

The pixel format routines (the `initWithAttributes:` method of the `NSOpenGLPixelFormat` class and the `CGLChoosePixelFormat` function) 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 6-1 illustrates this technique using the CGL API. Notice that the initial attributes list is set up with the supersample attribute last in the list. If the function `CGLChoosePixelFormat` returns `NULL`, it clears the supersample attribute to `NULL` and tries again.

Listing 6-1 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;
GLint 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 want to ensure that you obtain a pixel format that supports a specific renderer type, such as a hardware-accelerated renderer. Table 6-1 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 60).)

Table 6-1 Renderer types and pixel format attributes

Renderer type	CGL	Cocoa
Hardware-accelerated onscreen	`kCGLPFAAccelerated` `kCGLPFANoRecovery`	`NSOpenGLPFAAccelerated` `NSOpenGLPFANoRecovery`

Renderer type	CGL	Cocoa
Software (floating-point)	`kCGLPFARendererID` `kCGLRendererGeneric-FloatID`	`NSOpenGLPFARendererID` `kCGLRendererGenericFloatID`
System memory (not accelerated)	`kCGLPFAOffScreen`	`NSOpenGLPFAOffScreen`
Hardware-accelerated offscreen	`kCGLPFAPBuffer`	`NSOpenGLPFAPixelBuffer`

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. To do this, specify the no-recovery attribute for a windowed drawable object.

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, dragging a windowed drawable object from one display to the other is likely to yield a less than satisfactory result. Either rendering fails, or OpenGL uses the specified renderer and then copies the result 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 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 6-2 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 6-2 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 attribute for selected display
attributes[1] = (NSOpenGLPixelFormatAttribute)
                    CGDisplayIDToOpenGLDisplayMask (display);
return [[(NSOpenGLPixelFormat *)[NSOpenGLPixelFormat alloc]
initWithAttributes:attributes]
                                        autorelease];
}
```

Listing 6-3 shows how to use CGL to set up a context that uses a single renderer. The attribute `kCGLPFANoRecovery` ensures that OpenGL does not provide the fallback option of the software renderer.

Listing 6-3 Setting a CGL context to use a specific display

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

Allowing Offline Renderers

Adding the attribute `NSOpenGLPFAAllowOfflineRenderers` allows OpenGL to include offline renderers in the list of virtual screens returned in the pixel format object. Apple recommends you include this attribute, because it allows your application to work better in environments where renderers come and go, such as when a new display is plugged into a Mac.

If your application includes `NSOpenGLPFAAllowOfflineRenderers` in the list of attributes, your application must also watch for display changes and update its rendering context. See "Update the Rendering Context When the Renderer or Geometry Changes" (page 72).

OpenCL

If your applications uses OpenCL to perform other computations, you may want to find an OpenGL renderer that also supports OpenCL. To do this, add the attribute `NSOpenGLPFAAcceleratedCompute` to the pixel format attribute list. Adding this attribute restricts the list of renderers to those that also support OpenCL.

More information on OpenCL can be found in the *OpenCL Programming Guide for Mac*.

Deprecated Attributes

There are several renderer and buffer attributes that are no longer recommended either because they are too narrowly focused or no longer useful. Your application should move away from using any of these attributes:

- The robust attribute (`NSOpenGLPFARobust` or `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` or `kCGLPFAMultiScreen`) specifies only those renderers that can drive more than one screen at a time.
- The multiprocessing-safe attribute (`kCGLPFAMPSafe`) specifies only those renderers that are thread safe. This attribute is deprecated in 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 "Concurrency and OpenGL" (page 148).
- The compliant attribute (`NSOpenGLPFACompliant` or `kCGLPFACompliant`) specifies only OpenGL-compliant renderers. All OS X renderers are OpenGL-compliant, so this attribute is no longer useful.
- The full screen attribute (`kCGLPFAFullScreen`) requested special full screen contexts. The window screen attribute (`kCGLPFAWindow`) required the context to support windowed contexts. OS X no longer requires a special full screen context to be created, as it automatically provides the same performance benefits with a properly formatted window.
- The offscreen buffer attribute (`kCGLPFAOffScreen`) selects renderers capable of rendering to offscreen memory. Instead, use a frame buffer object as the rendering target and read the final results back to application memory.

- The pixel buffer attributes (`kCGLPFAPBuffer` and `kCGLPFARemotePBuffer` are no longer recommended; use frame buffer objects instead.
- The auxiliary buffers attribute (`kCGLPFAAuxBuffers`) specifies the number of required auxiliary buffers your application requires. Auxiliary buffers are not supported by the OpenGL 3.2 Core profile. Because auxiliary buffers are not supported, the `kCGLPFAAuxDepthStencil` attribute that modifies it is also deprecated.
- The accumulation buffer size attribute (`kCGLPFAAccumSize`) specifies the desired size for the accumulation buffer. Accumulation buffers are not supported by the OpenGL 3.2 Core Profile.

Important: Your application may not use any of the deprecated attributes in conjunction with a profile other than the legacy profile; if you do, pixel format creation fails.

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 that context's state, objects attached to that context, or the drawable object associated with that context. The actual drawing surfaces are never owned by the rendering context but are created, as needed, when the rendering context is actually attached to a drawable object. You can attach multiple rendering contexts to the same drawing surfaces. Each context maintains its own drawing state.

"Drawing to a Window or View" (page 35), "Drawing to the Full Screen" (page 50), and "Drawing Offscreen" (page 53) show how to create a rendering context and attach it to a drawable object. This chapter describes advanced ways to interact with rendering contexts.

Update the Rendering Context When the Renderer or Geometry Changes

A renderer change can occur when the user drags a window from one display to another or when a display is attached or removed. Geometry changes occur when the display mode changes or when a window is resized or moved. If your application uses an `NSOpenGLView` object to maintain the context, it is automatically updated. An application that creates a custom view to hold the rendering context must track the appropriate system events and update the context when the geometry or display changes.

Updating a rendering context notifies it of geometry changes; it doesn't flush content. Calling an update function updates the attached drawable objects and ensures that the renderer is properly updated for any virtual screen changes. If you don't update the rendering context, you may see rendering artifacts.

The routine that you call for updating determines how events related to renderer and geometry changes are handled. For applications that use or subclass `NSOpenGLView`, Cocoa calls the `update` method automatically. Applications that create an `NSOpenGLContext` object manually 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.

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.

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 26)). 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 should check for a virtual screen change and respond to the change to ensure 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 `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.

When the renderer changes, the limits and extensions available to OpenGL may also change. Your application should retest the capabilities of the renderer and use these to choose its rendering algorithms appropriately. See "Determining the OpenGL Capabilities Supported by the Renderer" (page 83).

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 drawing content. Using this approach provides results that are equivalent to using the `reshape` method of the `NSOpenGLView` class.

Listing 7-1 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`, which 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 7-1 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];
}

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

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

@end
```

Context Parameters Alter the Context's Behavior

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 `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*, and *CGL Reference*.

Swap Interval Allows an Application to Synchronize Updates to the Screen Refresh

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 "Synchronize with the Screen Refresh Rate" (page 96).

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

- For Cocoa, use `NSOpenGLCPSwapInterval`.
- If you are using the CGL API, use `kCGLCPSwapInterval` as shown in Listing 7-2.

Listing 7-2 Using CGL to set up synchronization

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

Surface Opacity Specifies How the OpenGL Surface Blends with Surfaces Behind It

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`.
- If you are using the CGL API, use `kCGLCPSurfaceOpacity` as shown in Listing 7-3.

Listing 7-3 Using CGL to set surface opacity

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

Surface Drawing Order 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 OpenGL 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`.
- If you are using the CGL API, use `kCGLCPSurfaceOrder` as shown in Listing 7-4.

Listing 7-4 Using CGL to set surface drawing order

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

Determining Whether Vertex and Fragment Processing Happens on the GPU

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`. Listing 7-5 demonstrates how to use these parameters.

> **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 7-5 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;
```

Controlling the Back Buffer Size

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×pixels, the OpenGL back buffer is sized to match. If the window grows to 1024×768 pixels, for example, then the back buffer is resized as well. 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 7-1). 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.

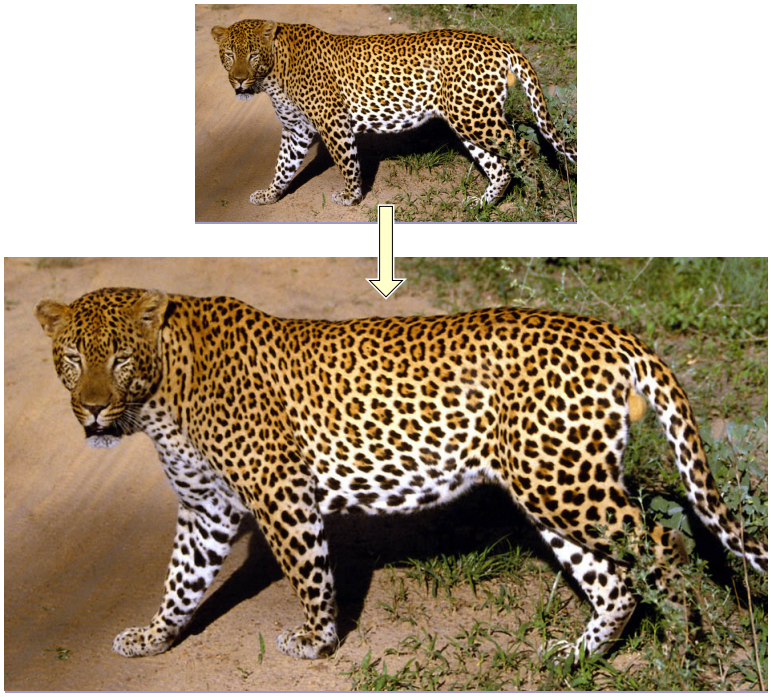
You can use the following constants to specify that you are setting the surface backing size:

- If you are using the CGL API, use `kCGLCPSurfaceBackingSize`, as shown in Listing 7-6.

Listing 7-6 Using CGL to set up back buffer size control

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

Figure 7-1 A fixed size back buffer and variable size front buffer



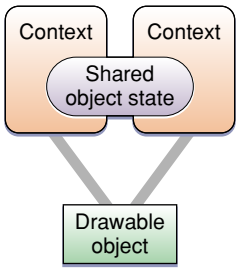
Sharing Rendering Context Resources

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 7-2 (page 80)) or to different drawable objects (see Figure 7-3 (page 80)). 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 is 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 7-2 Shared contexts attached to the same drawable object

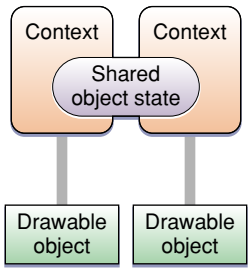


When you create an OpenGL context, you can designate another context whose object resources you want to share. All sharing is peer to peer. Shared resources are reference-counted and thus are maintained until explicitly released or when the last context-sharing resource is released.

Not every context can be shared with every other context. Both contexts must share the same OpenGL profile. You must also ensure that both contexts share the same set of renderers. You meet these requirements 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 7-3 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 7-7 (page 81).
- Use the `share` parameter for the function `CGLCreateContext`. See Listing 7-8 (page 82).

Listing 7-7 ensures the same virtual screen list by using the same pixel format object for each of the shared contexts.

Listing 7-7 Setting up an `NSOpenGLContext` object for sharing

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

- (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 7-8 ensures the same virtual screen list by using the same pixel format object for each of the shared contexts.

Listing 7-8 Setting up a CGL context for sharing

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

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

Determining the OpenGL Capabilities Supported by the Renderer

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 Working Group. Some extensions are promoted from a vendor-specific extension to one shared by more than one vendor, sometimes even being incorporated into the core OpenGL API. Extensions allow OpenGL to embrace innovation, but require you to 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-approved extension that has functionality similar to that of a vendor-specific extension. Your application should prefer core functionality or ARB-approved extensions over those specific to a particular vendor, when both are offered by the same renderer. This makes it easier to transparently support new renderers from other vendors.

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. 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. For example, a software renderer might not support fog effects even though fog effects are available in an OpenGL extension implemented by a hardware vendor on the same system. For this reason, it's important that you check for functionality on a per-renderer basis.

Regardless of what functionality 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 do not allocate a fixed-length string for the return value of the `glGetString` function. 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 OS X. Because extensions are not typically removed, the methodology you use today to check for a feature works in future versions of OS X.

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 the renderer changes. Listing 8-1 shows a code excerpt that checks for a few extensions. A detailed explanation for each line of code appears following the listing.

Listing 8-1 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 is 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 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.

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.

The code in Listing 8-2 shows one way to query the current renderer. It uses the CGL API, which can be called from Cocoa 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 your functionality snapshot each time the list of displays or display configuration changes.

Listing 8-2 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,
                            0};                                                  // 3
    CGLPixelFormatObj pixelFormat = NULL;
    GLint 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 8-1 (page 84) 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 ensure that your functionality-checking code is thorough yet efficient.

- Don't rely on what's in a header file. A function 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.
- Make sure that you are aware of whether a feature is being used as part of the Core OpenGL API or as an extension. When a feature is implemented both as part of the core OpenGL API and as an extension, it uses different constants and function names.

OpenGL Renderer Implementation-Dependent Values

The OpenGL specification defines implementation-dependent values that define the limits of what an OpenGL implementation is capable of. For example, the maximum size of a texture and the number of texture units are both common implementation-dependent values that an application is expected to check. Each of these values provides a minimum value that all conforming OpenGL implementations are expected to support. If your application's usage exceeds these minimums, it must check the limit first, and fail gracefully if the implementation cannot provide the limit desired. Your application may need to load smaller textures, disable a rendering feature, or choose a different implementation.

Although the specification provides a comprehensive list of these limitations, a few stand out in most OpenGL applications. Table 8-1 lists values that applications should test if they require more than the minimum values in the specification.

Table 8-1 Common OpenGL renderer limitations

Maximum size of the texture	GL_MAX_TEXTURE_SIZE
Number of depth buffer planes	GL_DEPTH_BITS
Number of stencil buffer planes	GL_STENCIL_BITS

The limit on the size and complexity of your shaders is a key area you need to test. All graphics hardware supports limited memory to pass attributes into the vertex and fragment shaders. Your application must either keep its usage below the minimums as defined in the specification, or it must check the shader limitations documented in Table 8-2 and choose shaders that are within those limits.

Table 8-2 OpenGL shader limitations

Maximum number of vertex attributes	GL_MAX_VERTEX_ATTRIBS
Maximum number of uniform vertex vectors	GL_MAX_VERTEX_UNIFORM_COMPONENTS
Maximum number of uniform fragment vectors	GL_MAX_FRAGMENT_UNIFORM_COMPONENTS
Maximum number of varying vectors	GL_MAX_VARYING_FLOATS
Maximum number of texture units usable in a vertex shader	GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS
Maximum number of texture units usable in a fragment shader	GL_MAX_TEXTURE_IMAGE_UNITS

OpenGL Application Design Strategies

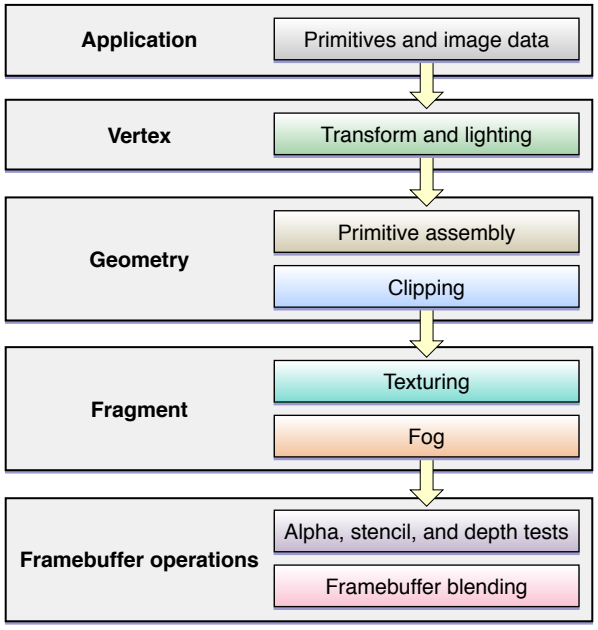
OpenGL performs many complex operations—transformations, lighting, clipping, texturing, environmental effects, and so on—on large data sets. The size of your data and the complexity of the calculations performed on it can impact performance, making your stellar 3D graphics shine less brightly than you'd like. Whether your application is a game using OpenGL to provide immersive real-time images to the user or an image processing application more concerned with image quality, use the information in this chapter to help you design your application.

Visualizing OpenGL

The most common way to visualize OpenGL is as a graphics pipeline, as shown in Figure 9-1 (page 90). Your application sends vertex and image data, configuration and state changes, and rendering commands to OpenGL. Vertices are processed, assembled into primitives, and rasterized into fragments. Each fragment is calculated and merged into the framebuffer. The pipeline model is useful for identifying exactly what work your application must perform to generate the results you want. OpenGL allows you to customize each stage of the graphics pipeline, either through customized shader programs or by configuring a fixed-function pipeline through OpenGL function calls.

In most implementations, each pipeline stage can act in parallel with the others. This is a key point. If any one pipeline stage performs too much work, then the other stages sit idle waiting for it to complete. Your design should balance the work performed in each pipeline stage to the capabilities of the renderer. When you tune your application's performance, the first step is usually to determine which stage the application is bottlenecked in, and why.

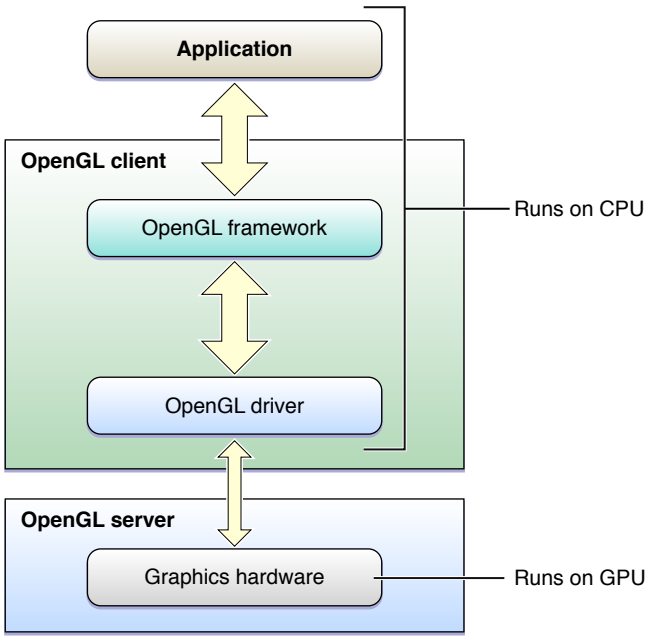
Figure 9-1 OpenGL graphics pipeline



Another way to visualize OpenGL is as a client-server architecture, as shown in Figure 9-2 (page 91). OpenGL state changes, texture and vertex data, and rendering commands must all travel from the application to the OpenGL client. The client transforms these items so that the graphics hardware can understand them, and then forwards them to the GPU. Not only do these transformations add overhead, but the bandwidth between the CPU and the graphics hardware is often lower than other parts of the system.

To achieve great performance, an application must reduce the frequency of calls they make to OpenGL, minimize the transformation overhead, and carefully manage the flow of data between the application and the graphics hardware. For example, OpenGL provides mechanisms that allow some kinds of data to be cached in dedicated graphics memory. Caching reusable data in graphics memory reduces the overhead of transmitting data to the graphics hardware.

Figure 9-2 OpenGL client-server architecture



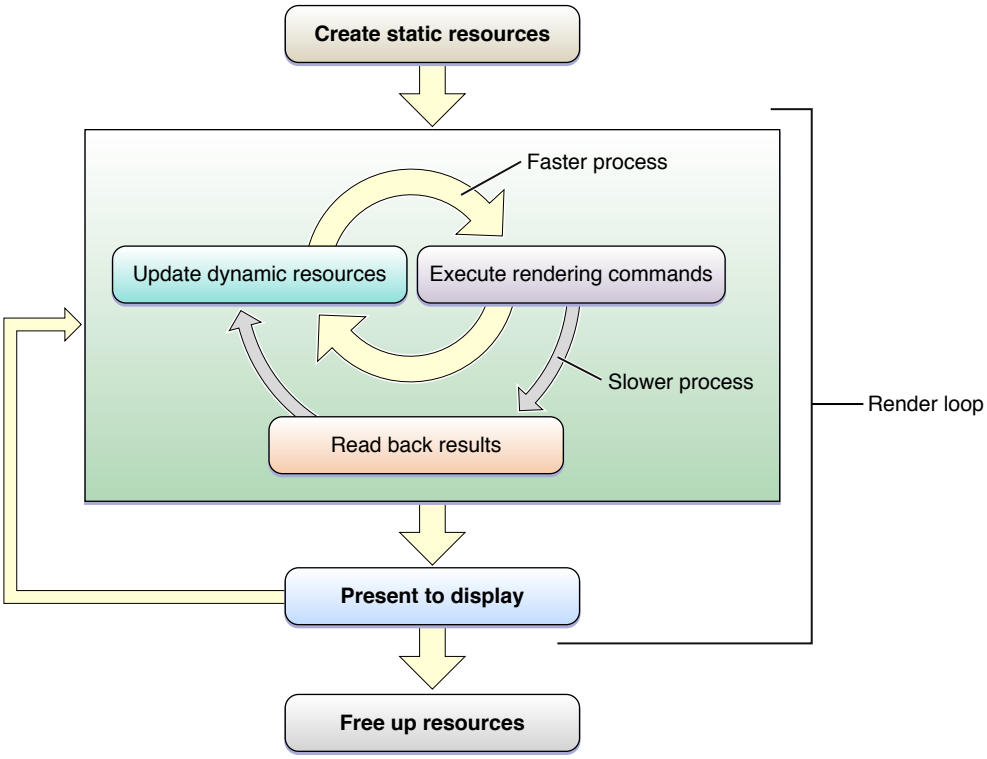
Designing a High-Performance OpenGL Application

To summarize, a well-designed OpenGL application needs to:

- Exploit parallelism in the OpenGL pipeline.
- Manage data flow between the application and the graphics hardware.

Figure 9-3 shows a suggested process flow for an application that uses OpenGL to perform animation to the display.

Figure 9-3 Application model for managing resources



When the application launches, it creates and initializes any static resources it intends to use in the renderer, encapsulating those resources into OpenGL objects where possible. The goal is to create any object that can remain unchanged for the runtime of the application. This trades increased initialization time for better rendering performance. Ideally, complex commands or batches of state changes should be replaced with OpenGL objects that can be switched in with a single function call. For example, configuring the fixed-function pipeline can take dozens of function calls. Replace it with a graphics shader that is compiled at initialization time, and you can switch to a different program with a single function call. In particular, OpenGL objects that are expensive to create or modify should be created as static objects.

The rendering loop processes all of the items you intend to render to the OpenGL context, then swaps the buffers to display the results to the user. In an animated scene, some data needs to be updated for every frame. In the inner rendering loop shown in Figure 9-3, the application alternates between updating rendering resources (possibly creating or modifying OpenGL objects in the process) and submitting rendering commands that use those resources. The goal of this inner loop is to balance the workload so that the CPU and GPU are working in parallel, without blocking each other by using the same resources simultaneously.

A goal for the inner loop is to avoid copying data back from the graphics processor to the CPU. Operations that require the CPU to read results back from the graphics hardware are sometimes necessary, but in general reading back results should be used sparingly. If those results are also used to render the current frame, as shown in the middle rendering loop, this can be very slow. Copying data from the GPU to the CPU often requires that some or all previously submitted drawing commands have completed.

After the application submits all drawing commands needed in the frame, it presents the results to the screen. Alternatively, a non-interactive application might read the final image back to the CPU, but this is also slower than presenting results to the screen. This step should be performed only for results that must be read back to the application. For example, you might copy the image in the back buffer to save it to disk.

Finally, when your application is ready to shut down, it deletes static and dynamic resources to make more hardware resources available to other applications. If your application is moved to the background, releasing resources to other applications is also good practice.

To summarize the important characteristics of this design:

- Create static resources, whenever practical.
- The inner rendering loop alternates between modifying dynamic resources and submitting rendering commands. Enough work should be included in this loop so that when the application needs to read or write to any OpenGL object, the graphics processor has finished processing any commands that used it.
- Avoid reading intermediate rendering results into the application.

The rest of this chapter provides useful OpenGL programming techniques to implement the features of this rendering loop. Later chapters demonstrate how to apply these general techniques to specific areas of OpenGL programming.

- "Update OpenGL Content Only When Your Data Changes" (page 94)
- "Avoid Synchronizing and Flushing Operations" (page 96)
- "Allow OpenGL to Manage Your Resources" (page 99)
- "Use Optimal Data Types and Formats" (page 102)
- "Use Double Buffering to Avoid Resource Conflicts" (page 100)
- "Be Mindful of OpenGL State Variables" (page 101)
- "Use OpenGL Macros" (page 103)
- "Replace State Changes with OpenGL Objects" (page 102)

Update OpenGL Content Only When Your Data Changes

OpenGL applications should avoid recomputing a scene when the data has not changed. This is critical on portable devices, where power conservation is critical to maximizing battery life. You can ensure that your application draws only when necessary by following a few simple guidelines:

- If your application is rendering animation, use a Core Video display link to drive the animation loop. Listing 9-1 (page 94) provides code that allows your application to be notified when a new frame needs to be displayed. This code also synchronizes image updates to the refresh rate of the display. See "Synchronize with the Screen Refresh Rate" (page 96) for more information.
- If your application does not animate its OpenGL content, you should allow the system to regulate drawing. For example, in Cocoa call the `setNeedsDisplay:` method when your data changes.
- If your application does not use a Core Video display link, you should still 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.

Listing 9-1 Setting up a Core Video display link

```
@interface MyView : NSOpenGLView
{
    CVDisplayLinkRef displayLink; //display link for managing rendering thread
}
@end

- (void)prepareOpenGL
{
   // Synchronize buffer swaps with vertical refresh rate
    GLint swapInt = 1;
   [[self openGLContext] setValues:&swapInt forParameter:NSOpenGLCPSwapInterval];

    // Create a display link capable of being used with all active displays
    CVDisplayLinkCreateWithActiveCGDisplays(&displayLink);

    // Set the renderer output callback function
    CVDisplayLinkSetOutputCallback(displayLink, &MyDisplayLinkCallback, self);
```

```
    // Set the display link for the current renderer
    CGLContextObj cglContext = [[self openGLContext] CGLContextObj];
    CGLPixelFormatObj cglPixelFormat = [[self pixelFormat] CGLPixelFormatObj];
    CVDisplayLinkSetCurrentCGDisplayFromOpenGLContext(displayLink, cglContext,
cglPixelFormat);

    // Activate the display link
    CVDisplayLinkStart(displayLink);
}

// This is the renderer output callback function
static CVReturn MyDisplayLinkCallback(CVDisplayLinkRef displayLink, const
CVTimeStamp* now, const CVTimeStamp* outputTime,
CVOptionFlags flagsIn, CVOptionFlags* flagsOut, void* displayLinkContext)
{
    CVReturn result = [(MyView*)displayLinkContext getFrameForTime:outputTime];
    return result;
}

- (CVReturn)getFrameForTime:(const CVTimeStamp*)outputTime
{
    // Add your drawing codes here

    return kCVReturnSuccess;
}

- (void)dealloc
{
    // Release the display link
    CVDisplayLinkRelease(displayLink);

    [super dealloc];
}
```

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. To avoid tearing, applications use a double-buffered context and synchronize buffer swaps 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.

After you tell the context to swap the buffers, OpenGL must defer any rendering commands that follow that swap until after the buffers have successfully been exchanged. 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 9-2 shows how an `NSOpenGLView` object can synchronize with the screen refresh rate; you can use a similar approach if your application uses CGL contexts. 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.

Listing 9-2 Setting up synchronization

```
GLint swapInterval = 1;
[[self openGLContext] setValues:&swapInt forParameter:NSOpenGLCPSwapInterval];
```

Avoid Synchronizing and Flushing Operations

OpenGL is not required to execute most commands immediately. Often, they are queued to a command buffer and read and executed by the hardware at a later time. Usually, OpenGL waits until the application has queued up a significant number of commands before sending the buffer to the hardware—allowing the graphics hardware to execute commands in batches is often more efficient. However, some OpenGL functions must flush the buffer immediately. Other functions not only flush the buffer, but also block until previously submitted commands have completed before returning control to the application. Your application should restrict the

use of flushing and synchronizing commands only to those cases where that behavior is necessary. Excessive use of flushing or synchronizing commands add additional stalls waiting for the hardware to finish rendering. On a single-buffered context, flushing may also cause visual anomalies, such as flickering or tearing.

These situations require 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 all previously submitted commands to complete executing.
- Functions that retrieve OpenGL state (for example, `glGetError`), also wait for submitted commands to complete.
- Buffer swapping routines (the `flushBuffer` method of the `NSOpenGLContext` class or the `CGLFlushDrawable` function) implicitly call `glFlush`. Note that when using the `NSOpenGLContext` class or the CGL API, the term *flush* actually refers to a buffer-swapping operation. For single-buffered contexts, `glFlush` and `glFinish` are equivalent to a swap operation, since all rendering is taking place directly in the front buffer.
- The command buffer is full.

Using glFlush Effectively

Most of the time you don't need to call `glFlush` to move image data to the screen. There are only a few cases that require you to call the `glFlush` function:

- If your application submits rendering commands that use a particular OpenGL object, and it intends to modify that object in the near future. If you attempt to modify an OpenGL object that has pending drawing commands, your application may be forced to wait until those commands have been completed. In this situation, calling `glFlush` ensures that the hardware begins processing commands immediately. After flushing the command buffer, your application should perform work that does not need that resource. It can perform other work (even modifying other OpenGL objects).
- Your application needs to change the drawable object associated with the rendering context. Before you can switch to another drawable object, you must call `glFlush` to ensure that all commands written in the command queue for the previous drawable object have been submitted.
- When two contexts share an OpenGL object. After submitting any OpenGL commands, call `glFlush` before switching to the other context.
- To keep drawing synchronized across multiple threads and prevent command buffer corruption, each thread should submit its rendering commands and then call `glFlush`.

Avoid Querying OpenGL State

Calls to `glGet*()`, including `glGetError()`, may require OpenGL to execute previous commands before retrieving any state variables. This synchronization forces the graphics hardware to run lockstep with the CPU, reducing opportunities for parallelism.

Your application should keep shadow copies of any OpenGL state that you need to query, and maintain these shadow copies as you change the state.

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 CGL API. If you are developing a performance-critical application, retrieve error information only in the debugging phase. Calling `glGetError` excessively in a release build degrades performance.

Use Fences for Finer-Grained Synchronization

Avoid using `glFinish` in your application, because it waits until all previously submitted commands are completed before returning control to your application. Instead, you should use the fence extension (`APPLE_fence`). This extension was created to provide the level of granularity that is not provided by `glFinish`. A **fence** is a token used to mark the current point in the command stream. When used correctly, it allows you to ensure that a specific series of commands has been completed. A fence helps coordinate activity between the CPU and the GPU when they are using the same resources.

Follow these steps to set up and use a fence:

1. At initialization time, create the fence object by calling the function `glGenFencesAPPLE`.

   ```
   GLint myFence;
   glGenFencesAPPLE(1,&myFence);
   ```

2. Call the OpenGL functions that must complete prior to the fence.
3. 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:

   ```
   glSetFenceAPPLE(myFence);
   ```

4. Call `glFlush` to force the commands to be sent to the hardware. This step is optional, but recommended to ensure that the hardware begins processing OpenGL commands.
5. Perform other work in your application.
6. Wait for all OpenGL commands issued prior to the fence to complete by calling the function `glFinishFenceAPPLE`.

```
glFinishFenceAPPLE(myFence);
```

   As an alternative to calling `glFinishFenceAPPLE`, you can call `glTestFenceAPPLE` to determine whether the fence has been reached. The advantage of testing the fence is that your application does not block waiting for the fence to complete. This is useful if your application can continue processing other work while waiting for the fence to trigger.

```
glTestFenceAPPLE(myFence);
```

7. When your application no longer needs the fence, delete it by calling the function `glDeleteFencesAPPLE`.

```
glDeleteFencesAPPLE(1,&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.

Allow OpenGL to Manage Your Resources

OpenGL allows many data types to be stored persistently inside OpenGL. Creating OpenGL objects to store vertex, texture, or other forms of data allows OpenGL to reduce the overhead of transforming the data and sending them to the graphics processor. If data is used more frequently than it is modified, OpenGL can substantially improve the performance of your application.

OpenGL allows your application to hint how it intends to use the data. These hints allow OpenGL to make an informed choice of how to process your data. For example, static data might be placed in high-speed graphics memory directly connected to the graphics processor. Data that changes frequently might be kept in main memory and accessed by the graphics hardware through DMA.

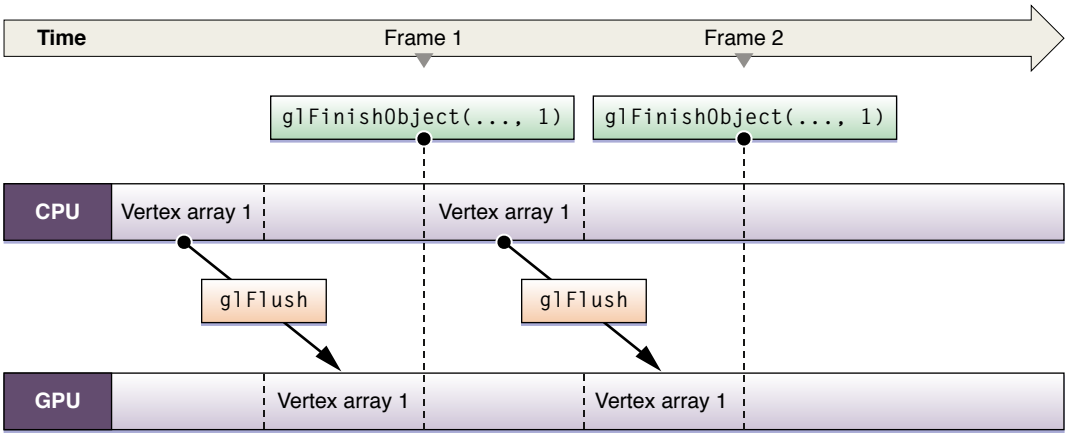
Use Double Buffering to Avoid Resource Conflicts

Resource conflicts occur when your application and OpenGL want to access a resource at the same time. When one participant attempts to modify an OpenGL object being used by the other, one of two problems results:

- The participant that wants to modify the object blocks until it is no longer in use. Then the other participant is not allowed to read from or write to the object until the modifications are complete. This is safe, but these can be hidden bottlenecks in your application.
- Some extensions allow OpenGL to access application memory that can be simultaneously accessed by the application. In this situation, synchronizing between the two participants is left to the application to manage. Your application calls `glFlush` to force OpenGL to execute commands and uses a fence or `glFinish` to ensure that no commands that access that memory are pending.

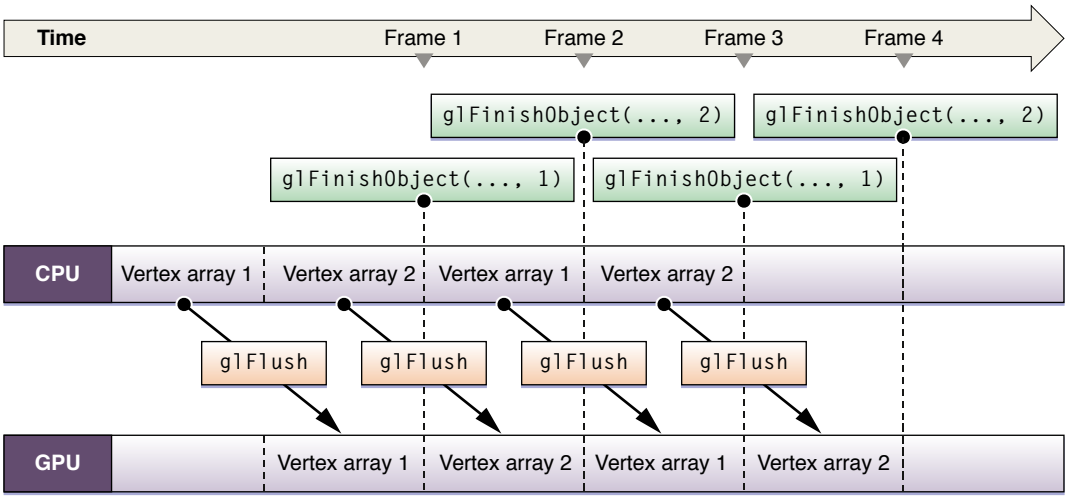
Whether your application relies on OpenGL to synchronize access to a resource, or it manually synchronizes access, resource contention forces one of the participants to wait, rather than allowing them both to execute in parallel. Figure 9-4 demonstrates this problem. There is only a single buffer for vertex data, which both the application and OpenGL want to use and therefore the application must wait until the GPU finishes processing commands before it modifies the data.

Figure 9-4 Single-buffered vertex array data



To solve this problem, your application could fill this idle time with other processing, even other OpenGL processing that does not need the objects in question. If you need to process more OpenGL commands, the solution is to create two of the same resource type and let each participant access a resource. Figure 9-5 illustrates the double-buffered approach. While the GPU operates on one set of vertex array data, the CPU is modifying the other. After the initial startup, neither processing unit is idle. This example uses a fence to ensure that access to each buffer is synchronized.

Figure 9-5 Double-buffered vertex array data



Double buffering is sufficient for most applications, but it requires that both participants finish processing their commands before a swap can occur. For a traditional producer-consumer problem, more than two buffers may prevent a participant from blocking. With triple buffering, the producer and consumer each have a buffer, with a third idle buffer. If the producer finishes before the consumer finishes processing commands, it takes the idle buffer and continues to process commands. In this situation, the producer idles only if the consumer falls badly behind.

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 9-3.

Listing 9-3 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.
```

Replace State Changes with OpenGL Objects

The "Be Mindful of OpenGL State Variables" (page 101) section suggests that reducing the number of state changes can improve performance. Some OpenGL extensions also allow you to create objects that collect multiple OpenGL state changes into an object that can be bound with a single function call. Where such techniques are available, they are recommended. For example, configuring the fixed-function pipeline requires many function calls to change the state of the various operators. Not only does this incur overhead for each function called, but the code is more complex and difficult to manage. Instead, use a shader. A shader, once compiled, can have the same effect but requires only a single call to `glUseProgram`.

Other examples of objects that take the place of multiple state changes include the "Vertex Array Range Extension" (page 113) and "Uniform Buffers" (page 143).

Use Optimal Data Types and Formats

If you don't use data types and formats that are native to the graphics hardware, OpenGL must convert those data types into a format that the graphics hardware understands.

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

For texture data, you'll get the best performance 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.

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. OS X allows your application to use macros to provide a local context variable and cache the current renderer in that variable. You get more benefit from using macros when your code makes millions of function calls per second.

Before implementing this technique, consider carefully whether you can redesign your application to perform less function calls. Frequently changing OpenGL state, pushing or popping matrices, or even submitting one vertex at a time are all examples of techniques that should be replaced with more efficient operations.

You can use the CGL macro header (`CGL/CGLMacro.h`) if your application uses CGL from a Cocoa application. You must define the local variable `cgl_ctx` to be equal to the current context. Listing 9-4 shows what's needed to set up macro use for the CGL API. First, you need to include the correct macro header. Then, you must set the current context.

Listing 9-4 Using CGL macros

```
#include <CGL/CGLMacro.h> // include the header
CGL_MACRO_DECLARE_VARIABLES // set the current context
glBegin (GL_QUADS);     // This code now uses the macro
    // draw here
glEnd ();
```

Best Practices 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.

Figure 10-1 Vertex data sets can be quite large

Applications that use large vertex data sets can adopt one or more of the strategies described in "OpenGL Application Design Strategies" (page 89) to optimize how vertex data is delivered to OpenGL.This chapter expands on those best practices with specific techniques for working with vertex data.

Understand How Vertex Data Flows Through OpenGL

Understanding how vertex data flows through OpenGL is important to choosing strategies for handling the data. Vertex data enters into the vertex stage, where it is processed by either the built-in fixed function vertex stage or a custom vertex.

Figure 10-2 Vertex data path

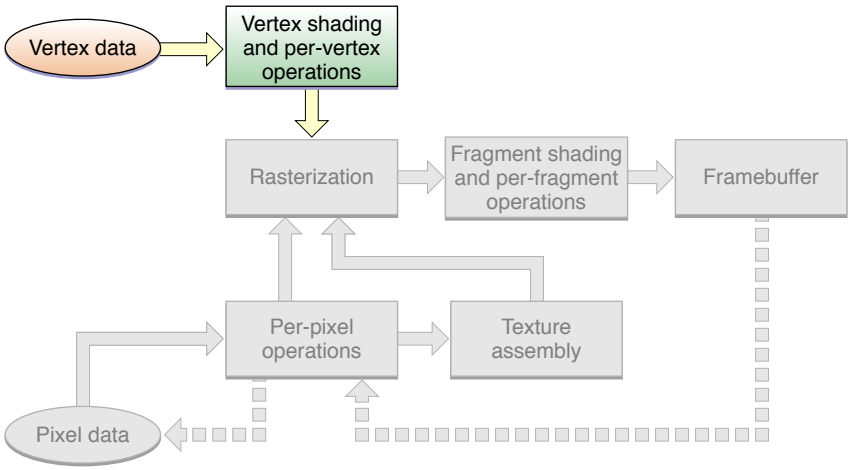
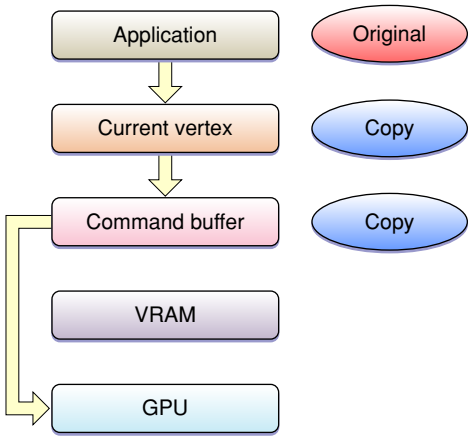


Figure 10-3 takes a closer look at the vertex data path when using immediate mode. Without any optimizations, your vertex data may be copied at various points in the data path. If your application uses immediate mode to each vertex separately, calls to OpenGL first modify the current vertex, which is copied into the command buffer whenever your application makes a `glVertex*` call. This is not only expensive in terms of copy operations, but also in function overhead to specify each vertex.

Figure 10-3 Immediate mode requires a copy of the current vertex data



The OpenGL commands `glDrawRangeElements`, `glDrawElements`, and `glDrawArrays` render multiple geometric primitives from array data, using very few subroutine calls. Listing 10-1 shows a typical implementation. Your application creates a vertex structure that holds all the elements for each vertex. For each element , you enable a client array and provide a pointer and offset to OpenGL so that it knows how to find those elements.

Listing 10-1 Submitting vertex data using `glDrawElements`.

```
typedef struct _vertexStruct
{
    GLfloat position[2];
    GLubyte color[4];
} vertexStruct;

void DrawGeometry()
{
    const vertexStruct vertices[] = {...};
    const GLubyte indices[] = {...};

    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, sizeof(vertexStruct), &vertices[0].position);
    glEnableClientState(GL_COLOR_ARRAY);
   glColorPointer(4, GL_UNSIGNED_BYTE, sizeof(vertexStruct), &vertices[0].color);

    glDrawElements(GL_TRIANGLE_STRIP, sizeof(indices)/sizeof(GLubyte),
GL_UNSIGNED_BYTE, indices);
}
```

Each time you call `glDrawElements`, OpenGL must copy all of the vertex data into the command buffer, which is later copied to the hardware. The copy overhead is still expensive.

Techniques for Handling Vertex Data

Avoiding unnecessary copies of your vertex data is critical to application performance. This section summarizes common techniques for managing your vertex data using either built-in functionality or OpenGL extensions. Before using these techniques, you must ensure that the necessary functions are available to your application. See "Detecting Functionality" (page 83).

- Avoid the use of `glBegin` and `glEnd` to specify your vertex data. The function and copying overhead makes this path useful only for very small data sets. Also, applications written with `glBegin` and `glEnd` are not portable to OpenGL ES on iOS.
- 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 preferred way to manage your vertex data is with vertex buffer objects. Vertex buffer objects are buffers owned by OpenGL that hold your vertex information. These buffers allow OpenGL to place your vertex data into memory that is accessible to the graphics hardware. See "Vertex Buffers" (page 107) for more information.
- If vertex buffer objects are not available, your application can search for the `GL_APPLE_vertex_array_range` and `APPLE_fence` extensions. Vertex array ranges allow you to prevent OpenGL from copying your vertex data into the command buffer. Instead, your application must avoid modifying or deleting the vertex data until OpenGL finishes executing drawing commands. This solution requires more effort from the application, and is not compatible with other platforms, including iOS. See "Vertex Array Range Extension" (page 113) for more information.
- Complex vertex operations require many array pointers to be enabled and set before you call `glDrawElements`. The `GL_APPLE_vertex_array_object` extension allows your application to consolidate a group of array pointers into a single object. Your application switches multiple pointers by binding a single vertex array object, reducing the overhead of changing state. See "Vertex Array Object" (page 116).
- Use double buffering to reduce resource contention between your application and OpenGL. See "Use Double Buffering to Avoid Resource Conflicts" (page 100).
- If you need to compute new vertex information between frames, consider using vertex shaders and buffer objects to perform and store the calculations.

Vertex Buffers

Vertex buffers are available as a core feature starting in OpenGL 1.5, and on earlier versions of OpenGL through the vertex buffer object extension (`GL_ARB_vertex_buffer_object`). Vertex buffers are used to improve the throughput of static or dynamic vertex data in your application.

A buffer object is a chunk of memory owned by OpenGL. Your application reads from or writes to the buffer using OpenGL calls such as `glBufferData`, `glBufferSubData`, and `glGetBufferSubData`. Your application can also gain a pointer to this memory, an operation referred to as *mapping a buffer*. OpenGL prevents your application and itself from simultaneously using the data stored in the buffer. When your application maps a buffer or attempts to modify it, OpenGL may block until previous drawing commands have completed.

Using Vertex Buffers

You can set up and use vertex buffers by following these steps:

1. Call the function `glGenBuffers` to create a new name for a buffer object.

   ```
   void glGenBuffers(sizei n, uint *buffers );
   ```

   `n` is the number of buffers you wish to create identifiers for.

   `buffers` specifies a pointer to memory to store the buffer names.

2. Call the function `glBindBuffer` 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 glBindBuffer(GLenum target, GLuint buffer);
   ```

   `target` must be set to `GL_ARRAY_BUFFER`.

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

3. Fill the buffer object by calling the function `glBufferData`. Essentially, this call uploads your data to the GPU.

   ```
   void glBufferData(GLenum target, sizeiptr size,
               const GLvoid *data, GLenum usage);
   ```

   `target` must be set to `GL_ARRAY_BUFFER`.

   `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 stored 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 stored in the buffer object. These examples use `GL_STREAM_DRAW`, which indicates that the application plans to both modify and draw using the buffer, and `GL_STATIC_DRAW`, which indicates that the application will define the data once but use it to draw many times. For more details on buffer hints, see "Buffer Usage Hints" (page 110)

4. Enable the vertex array by calling `glEnableClientState` and supplying the `GL_VERTEX_ARRAY` constant.
5. Point to the contents of the vertex buffer object by calling a function such as `glVertexPointer`. Instead of providing a pointer, you provide an offset into the vertex buffer object.
6. To update the data in the buffer object, your application calls `glMapBuffer`. Mapping the buffer prevents the GPU from operating on the data, and gives your application a pointer to memory it can use to update the buffer.

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

   `target` must be set to `GL_ARRAY_BUFFER`.

   `access` indicates the operations you plan to perform on the data. You can supply `READ_ONLY`, `WRITE_ONLY`, or `READ_WRITE`.
7. Write pixel data to the pointer received from the call to `glMapBuffer`.
8. When your application has finished modifying the buffer contents, call the function `glUnmapBuffer`. You must supply `GL_ARRAY_BUFFER` as the parameter to this function. Once the buffer is unmapped, the pointer is no longer valid, and the buffer's contents are uploaded again to the GPU.

Listing 10-2 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 10-2 Using the vertex buffer object extension with dynamic data

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

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

//  When you want to draw using the vertex data
draw_loop {
```

```
    glBufferData(GL_ARRAY_BUFFER, bufferSize, NULL, GL_STREAM_DRAW);
    my_vertex_pointer = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
    GenerateMyDynamicVertexData(my_vertex_pointer);
    glUnmapBuffer(GL_ARRAY_BUFFER);
    PerformDrawing();
}
```

Listing 10-3 shows code that uses the vertex buffer object extension with static data.

Listing 10-3 Using the vertex buffer object extension with static data

```
//  To set up the vertex buffer object extension
#define BUFFER_OFFSET(i) ((char*)NULL + (i))
glBindBuffer(GL_ARRAY_BUFFER, myBufferName);
glBufferData(GL_ARRAY_BUFFER, bufferSize, NULL, GL_STATIC_DRAW);
GLvoid* my_vertex_pointer = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
GenerateMyStaticVertexData(my_vertex_pointer);
glUnmapBuffer(GL_ARRAY_BUFFER);

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

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

Buffer Usage Hints

A key advantage of buffer objects is that the application can provide information on how it uses the data stored in each buffer. For example, Listing 10-2 and Listing 10-3 differentiated between cases where the data were expected to never change (`GL_STATIC_DRAW`) and cases where the buffer data might change (`GL_DYNAMIC_DRAW`). The usage parameter allows an OpenGL renderer to alter its strategy for allocating the vertex buffer to improve performance. For example, static buffers may be allocated directly in GPU memory, while dynamic buffers may be stored in main memory and retrieved by the GPU via DMA.

If OpenGL ES compatibility is useful to you, you should limit your usage hints to one of three usage cases:

- `GL_STATIC_DRAW` should be used for vertex data that is specified once and never changed. Your application should create these vertex buffers during initialization and use them repeatedly until your application shuts down.
- `GL_DYNAMIC_DRAW` should be used when the buffer is expected to change after it is created. Your application should still allocate these buffers during initialization and periodically update them by mapping the buffer.
- `GL_STREAM_DRAW` is used when your application needs to create transient geometry that is rendered and then discarded. This is most useful when your application must dynamically change vertex data every frame in a way that cannot be performed in a vertex shader. To use a stream vertex buffer, your application initially fills the buffer using `glBufferData`, then alternates between drawing using the buffer and modifying the buffer.

Other usage constants are detailed in the vertex buffer specification.

If different elements in your vertex format have different usage characteristics, you may want to split the elements into one structure for each usage pattern and allocate a vertex buffer for each. Listing 10-4 shows how to implement this. In this example, position data is expected to be the same in each frame, while color data may be animated in every frame.

Listing 10-4 Geometry with different usage patterns

```
typedef struct _vertexStatic
{
    GLfloat position[2];
} vertexStatic;

typedef struct _vertexDynamic
{
    GLubyte color[4];
} vertexDynamic;

// Separate buffers for static and dynamic data.
GLuint    staticBuffer;
GLuint    dynamicBuffer;
GLuint    indexBuffer;

const vertexStatic staticVertexData[] = {...};
vertexDynamic dynamicVertexData[] = {...};
```

```
const GLubyte indices[] = {...};


void CreateBuffers()
{
    glGenBuffers(1, &staticBuffer);
    glGenBuffers(1, &dynamicBuffer);
    glGenBuffers(1, &indexBuffer);


// Static position data
    glBindBuffer(GL_ARRAY_BUFFER, staticBuffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(staticVertexData), staticVertexData,
GL_STATIC_DRAW);


    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);
   glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);


// Dynamic color data
// While not shown here, the expectation is that the data in this buffer changes
between frames.
    glBindBuffer(GL_ARRAY_BUFFER, dynamicBuffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(dynamicVertexData), dynamicVertexData,
GL_DYNAMIC_DRAW);
}


void DrawUsingVertexBuffers()
{
    glBindBuffer(GL_ARRAY_BUFFER, staticBuffer);
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, sizeof(vertexStatic),
(void*)offsetof(vertexStatic,position));
    glBindBuffer(GL_ARRAY_BUFFER, dynamicBuffer);
    glEnableClientState(GL_COLOR_ARRAY);
    glColorPointer(4, GL_UNSIGNED_BYTE, sizeof(vertexDynamic),
(void*)offsetof(vertexDynamic,color));
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);
```

```
    glDrawElements(GL_TRIANGLE_STRIP, sizeof(indices)/sizeof(GLubyte),
GL_UNSIGNED_BYTE, (void*)0);
}
```

Flush Buffer Range Extension

When your application unmaps a vertex buffer, the OpenGL implementation may copy the full contents of the buffer to the graphics hardware. If your application changes only a subset of a large buffer, this is inefficient. The `APPLE_flush_buffer_range` extension allows your application to tell OpenGL exactly which portions of the buffer were modified, allowing it to send only the changed data to the graphics hardware.

To use the flush buffer range extension, follow these steps:

1. Turn on the flush buffer extension by calling `glBufferParameteriAPPLE`.

   ```
   glBufferParameteriAPPLE(GL_ARRAY_BUFFER,GL_BUFFER_FLUSHING_UNMAP_APPLE,
   GL_FALSE);
   ```

   This disables the normal flushing behavior of OpenGL.

2. Before you unmap a buffer, you must call `glFlushMappedBufferRangeAPPLE` for each range of the buffer that was modified by the application.

   ```
   void glFlushMappedBufferRangeAPPLE(enum target, intptr offset, sizeiptr
   size);
   ```

   `target` is the type of buffer being modified; for vertex data it's `ARRAY_BUFFER`.

   `offset` is the offset into the buffer for the modified data.

   `size` is the length of the modified data in bytes.

3. Call `glUnmapBuffer`. OpenGL unmaps the buffer, but it is required to update only the portions of the buffer your application explicitly marked as changed.

For more information see the APPLE_flush_buffer_range specification.

Vertex Array Range Extension

The vertex array range extension (`APPLE_vertex_array_range`) lets you define a region of memory for your vertex data. The OpenGL driver can 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.

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 supplied data to be static, use `STORAGE_CACHED_APPLE` so that OpenGL can optimize appropriately.
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. If you need to modify the vertex data, set a fence object after you've submitted all the drawing commands. See "Use Fences for Finer-Grained Synchronization" (page 98)
8. Perform other work so that the GPU has time to process the drawing commands that use the vertex array.
9. Call `glFinishFenceAPPLE` to gain access to the vertex array.
10. Modify the data in the vertex array.
11. 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.

Listing 10-5 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 "Use Double Buffering to Avoid Resource Conflicts" (page 100).

Listing 10-5 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 10-6 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 10-6 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 Array Object

Look at the `DrawUsingVertexBuffers` function in Listing 10-4 (page 111). It configures buffer pointers for position, color, and indexing before calling `glDrawElements`. A more complex vertex structure may require additional buffer pointers to be enabled and changed before you can finally draw your geometry. If your application swaps frequently between multiple configurations of elements, changing these parameters adds significant overhead to your application. The `APPLE_vertex_array_object` extension allows you to combine a collection of buffer pointers into a single OpenGL object, allowing you to change all the buffer pointers by binding a different vertex array object.

To use this extension, follow these steps during your application's initialization routines:

1. Generate a vertex array object for a configuration of pointers you wish to use together.

```
void glGenVertexArraysAPPLE(sizei n, const uint *arrays);
```

`n` is the number of arrays you wish to create identifiers for.

`arrays` specifies a pointer to memory to store the array names.

```
glGenVertexArraysAPPLE(1,&myArrayObject);
```

2. Bind the vertex array object you want to configure.

```
void glBindVertexArrayAPPLE(uint array);
```

`array` is the identifier for an array that you received from `glGenVertexArraysAPPLE`.

```
glBindVertexArrayAPPLE(myArrayObject);
```

3. Call the pointer routines (`glColorPointer` and so forth.) that you would normally call inside your rendering loop. When a vertex array object is bound, these calls change the currently bound vertex array object instead of the default OpenGL state.

```
    glBindBuffer(GL_ARRAY_BUFFER, staticBuffer);
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, sizeof(vertexStatic),
(void*)offsetof(vertexStatic,position));
...
```

4. Repeat the previous steps for each configuration of vertex pointers.
5. Inside your rendering loop, replace the calls to configure the array pointers with a call to bind the vertex array object.

```
glBindVertexArrayAPPLE(myArrayObject);
glDrawArrays(...);
```

6. If you need to get back to the default OpenGL behavior, call `glBindVertexArrayAPPLE` and pass in `0`.

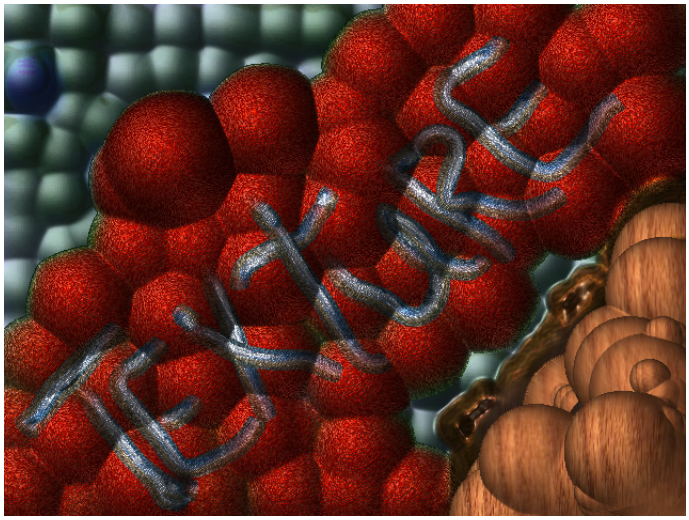
```
glBindVertexArrayAPPLE(0);
```

Best Practices for Working with Texture Data

Textures add realism to OpenGL objects. They help 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.

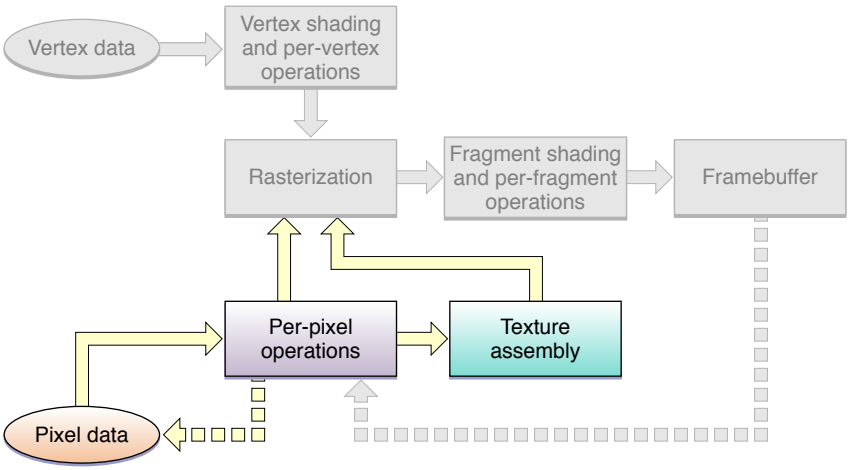
Many of the same techniques your application uses on vertex data can also be used to improve texture performance.

Figure 11-1 Textures add realism to a scene



Textures start as pixel data that flows through an OpenGL program, as shown in Figure 11-2.

Figure 11-2 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

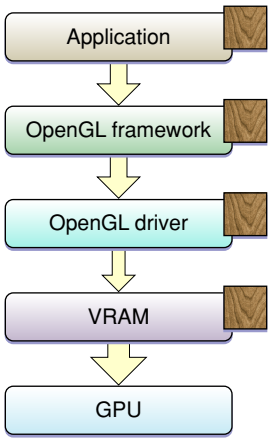
- shows how to use OpenGL extensions to optimize performance
- lists optimal data formats and types
- provides information on working with textures whose dimensions are not a power of two
- describes creating textures from image data
- shows how to download textures
- discusses using double buffers for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

Using Extensions to Improve Texture Performance

Without any optimizations, texture data flows through an OpenGL program as shown in Figure 11-3. 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

128)), 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 11-3 Data copies in an OpenGL program



Data flows at different rates through the system, as shown by the size of the arrows in Figure 11-3. 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_ARB_pixel_buffer_object` allows your application to use OpenGL buffer objects to manage texture and image data. As with vertex buffer objects, they allow your application to hint how a buffer is used and to decide when data is copied to OpenGL.
- `GL_APPLE_client_storage` allows you to prevent OpenGL from copying your texture data into the client. Instead, OpenGL keeps the memory pointer you provided when creating the texture. Your application must keep the texture data at that location until the referencing OpenGL texture is deleted.
- `GL_APPLE_texture_range`, along with a storage hint, either `GL_STORAGE_CACHED_APPLE` or `GL_STORAGE_SHARED_APPLE`, allows you to specify a single block of texture memory and manage it as you see fit.
- `GL_ARB_texture_rectangle` provides support for non-power of-two textures.

Here are some recommendations:

- If your application requires optimal texture upload performance, use `GL_APPLE_client_storage` and `GL_APPLE_texture_range` together to manage your textures.
- If your application requires optimal texture download performance, use pixel buffer objects.
- If your application requires cross-platform techniques, use pixel buffer objects for both texture uploads and texture downloads.
- Use `GL_ARB_texture_rectangle` when your source images are not aligned to a power-of-2 size.

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

Pixel Buffer Objects

Pixel buffer objects are a core feature of OpenGL 2.1 and also available through the `GL_ARB_pixel_buffer_object` extension. The procedure for setting up a pixel buffer object is almost identical to that of vertex buffer objects.

Using Pixel Buffer Objects to Efficiently Load Textures

1. Call the function `glGenBuffers` to create a new name for a buffer object.

```
void glGenBuffers(sizei n, uint *buffers );
```

   `n` is the number of buffers you wish to create identifiers for.

   `buffers` specifies a pointer to memory to store the buffer names.

2. Call the function `glBindBuffer` 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 glBindBuffer(GLenum target, GLuint buffer);
```

   `target` should be be set to `GL_PIXEL_UNPACK_BUFFER` to use the buffer as the source of pixel data.

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

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

```
void glBufferData(GLenum target, sizeiptr size,
            const GLvoid *data, GLenum usage);
```

`target` must be set to `GL_PIXEL_UNPACK_BUFFER`.

`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. For more details on buffer hints, see "Buffer Usage Hints" (page 110)

4. Whenever you call `glDrawPixels`, `glTexSubImage` or similar functions that read pixel data from the application, those functions use the data in the bound pixel buffer object instead.
5. To update the data in the buffer object, your application calls `glMapBuffer`. Mapping the buffer prevents the GPU from operating on the data, and gives your application a pointer to memory it can use to update the buffer.

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

`target` must be set to `PIXEL_UNPACK_BUFFER`.

`access` indicates the operations you plan to perform on the data. You can supply `READ_ONLY`, `WRITE_ONLY`, or `READ_WRITE`.

6. Modify the texture data using the pointer provided by map buffer.
7. When you have finished modifying the texture, call the function `glUnmapBuffer`. You should supply`PIXEL_UNPACK_BUFFER`. Once the buffer is unmapped, your application can no longer access the buffer's data through the pointer, and the buffer's contents are uploaded again to the GPU.

Using Pixel Buffer Objects for Asynchronous Pixel Transfers

`glReadPixels` normally blocks until previous commands have completed, which includes the slow process of copying the pixel data to the application. However, if you call `glReadPixels` while a pixel buffer object is bound, the function returns immediately. It does not block until you actually map the pixel buffer object to read its content.

1. Call the function `glGenBuffers` to create a new name for a buffer object.

```
void glGenBuffers(sizei n, uint *buffers );
```

`n` is the number of buffers you wish to create identifiers for.

`buffers` specifies a pointer to memory to store the buffer names.

2. Call the function `glBindBuffer` 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 glBindBuffer(GLenum target, GLuint buffer);
```

`target` should be be set to `GL_PIXEL_PACK_BUFFER` to use the buffer as the destination for pixel data.

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

3. Create and initialize the data store of the buffer object by calling the function `glBufferData`.

```
void glBufferData(GLenum target, sizeiptr size,
            const GLvoid *data, GLenum usage);
```

`target` must be set to `GL_ARRAY_BUFFER`.

`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. For more details on buffer hints, see "Buffer Usage Hints" (page 110)

4. Call `glReadPixels` or a similar function. The function inserts a command to read the pixel data into the bound pixel buffer object and then returns.
5. To take advantage of asynchronous pixel reads, your application should perform other work.
6. To retrieve the data in the pixel buffer object, your application calls `glMapBuffer`. This blocks OpenGL until the previously queued `glReadPixels` command completes, maps the data, and provides a pointer to your application.

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

`target` must be set to `GL_PIXEL_PACK_BUFFER`.

`access` indicates the operations you plan to perform on the data. You can supply `READ_ONLY`, `WRITE_ONLY`, or `READ_WRITE`.

7. Write vertex data to the pointer provided by map buffer.
8. When you no longer need the vertex data, call the function `glUnmapBuffer`. You should supply `GL_PIXEL_PACK_BUFFER`. Once the buffer is unmapped, the data is no longer accessible to your application.

Using Pixel Buffer Objects to Keep Data on the GPU

There is no difference between a vertex buffer object and a pixel buffer object except for the target to which they are bound. An application can take the results in one buffer and use them as another buffer type. For example, you could use the pixel results from a fragment shader and reinterpret them as vertex data in a future pass, without ever leaving the GPU:

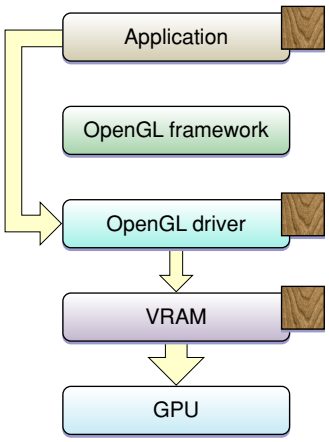
1. Set up your first pass and submit your drawing commands.
2. Bind a pixel buffer object and call `glReadPixels` to fetch the intermediate results into a buffer.
3. Bind the same buffer as a vertex buffer.
4. Set up the second pass of your algorithm and submit your drawing commands.

Keeping your intermediate data inside the GPU when performing multiple passes can result in great performance increases.

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, your application must retain its copy of the data until all referencing textures are deleted. By using this extension you can eliminate the OpenGL framework copy as shown in Figure 11-4. 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 11-4 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.

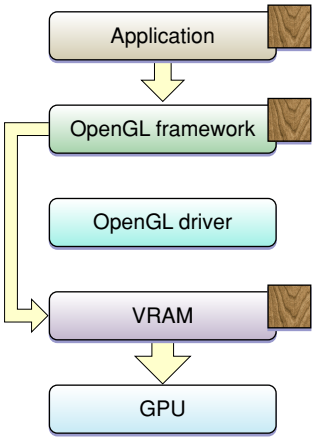
You provide the hint parameter and a parameter value to to the OpenGL function `glTexParameteri`. The possible values for the storage hint parameter (`GL_TEXTURE_STORAGE_HINT_APPLE`) are `GL_STORAGE_CACHED_APPLE` or `GL_STORAGE_SHARED_APPLE`.

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 11-5.

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 11-5 The texture range extension eliminates a data copy

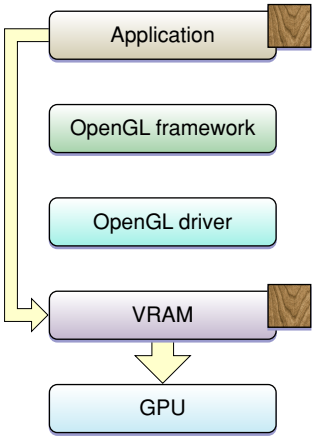


For detailed information on these extensions, see the OpenGL specification for the Apple texture range extension and the OpenGL specification for the ARB texture rectangle extension.

Combining Client Storage with Texture Ranges

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 11-6. 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 136).

Figure 11-6 Combining extensions to eliminate data copies



Listing 11-1 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 129) for an overview of the limitations and an alternative to using this extension.

Listing 11-1 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, 0, 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 11-2. The difference is that the `GL_TEXTURE_2D` texture target replaces the `GL_TEXTURE_RECTANGLE_ARB` texture target.

Listing 11-2 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, 0, 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.

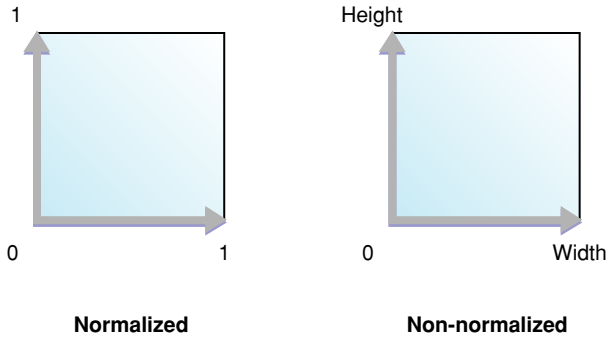
Working with Non–Power-of-Two Textures

OpenGL is often 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 11-7.)

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 11-7 Normalized and non-normalized coordinates



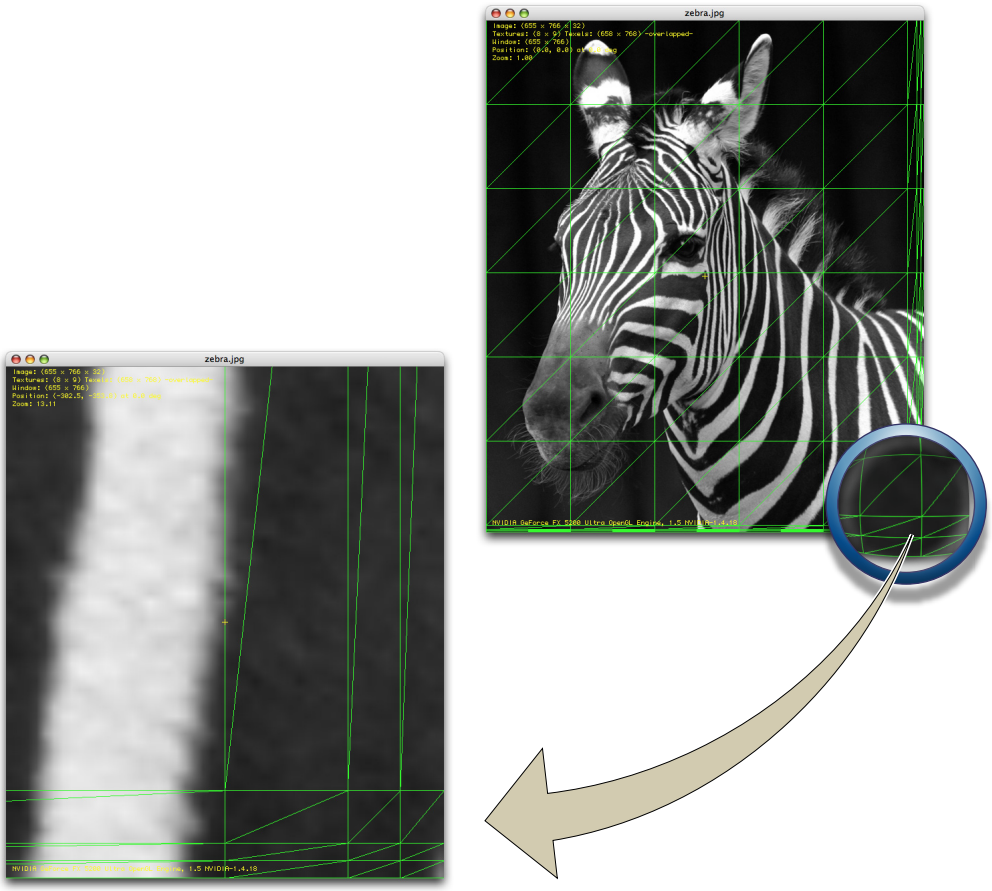
If your code runs on a system that does not support either the `ARB_texture_rectangle` or `ARB_texture_non_power_of_two` extensions you have these options for working with with rectangular images:

- 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 11-8 by using one image buffer and different texture pointers. Notice how the sides and corners of the image shown in Figure 11-8 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.

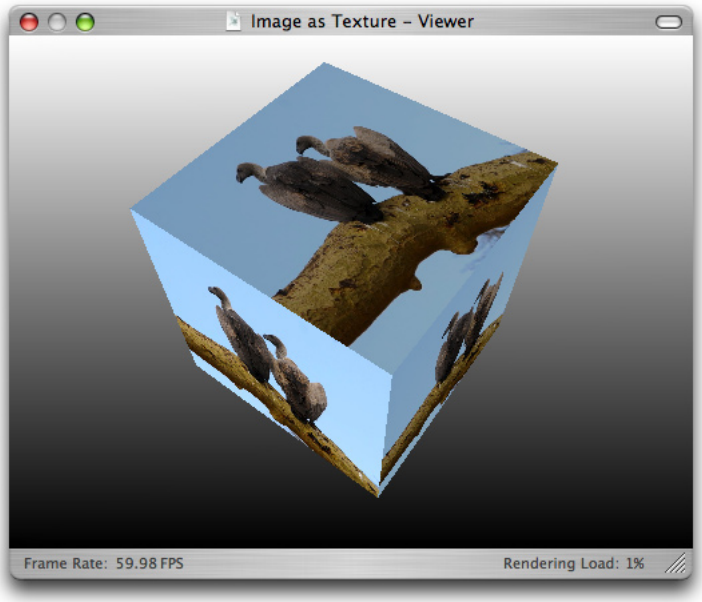
Figure 11-8 An image segmented into power-of-two tiles



Creating Textures from Image Data

OpenGL on the Macintosh provides several options for creating high-quality textures from image data. 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 11-9 shows an image used to texture a cube.

Figure 11-9 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. 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 11-3 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 11-3 Building an OpenGL texture from an `NSView` object

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

    [theView cacheDisplayInRect:[theView visibleRect] toBitmapImageRep:bitmap];
// 2
    samplesPerPixel = [bitmap samplesPerPixel];                                  // 3
    glPixelStorei(GL_UNPACK_ROW_LENGTH, [bitmap bytesPerRow]/samplesPerPixel);   // 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

   if(![bitmap isPlanar] &&
       (samplesPerPixel == 3 || samplesPerPixel == 4)) {                         // 9
        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
    }
}
```

Here's what the code does:

1. Allocates an `NSBitmapImageRep` object.
2. Initializes the `NSBitmapImageRep` object with bitmap data from the current view.
3. Gets the number of samples per pixel.
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. 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.

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 11-4 shows a code fragment that performs these steps. Note that you must have a valid, current OpenGL context.

Listing 11-4 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);
CGContextSetBlendMode(myBitmapContext, kCGBlendModeCopy);
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 11-5. 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 11-5, but alpha is premultiplied for the pixel data you get when using the code described in "Creating a Texture from a Cocoa View" (page 131) and "Creating a Texture from a Quartz Image Source" (page 133).

Listing 11-5 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 11-6 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 11-6 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, 0, 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,
```

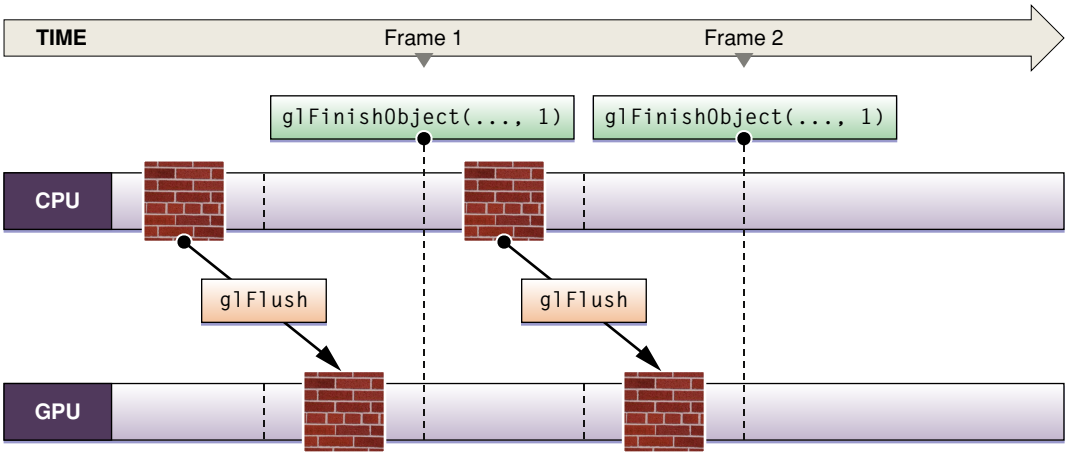
```
                GL_UNSIGNED_INT_8_8_8_8_REV, pixels);
```

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 11-10. Your application works on the data, flushes it to the GPU and waits until the GPU is finished before working on the data again.

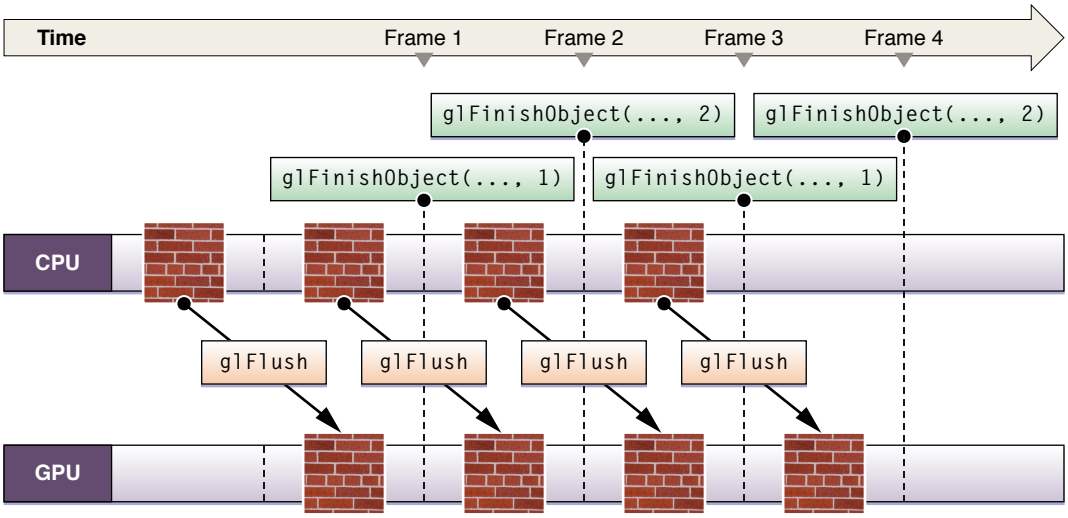
One technique for ensuring that the GPU is finished executing commands before your application sends more data is to insert a token into the command stream and use that to determine when the CPU can touch the data again, as described in "Use Fences for Finer-Grained Synchronization" (page 98). Figure 11-10 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 11-11 (page 138).

Figure 11-10 Single-buffered data



To double buffer data, you must supply two sets of data to work on. Note in Figure 11-11 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.

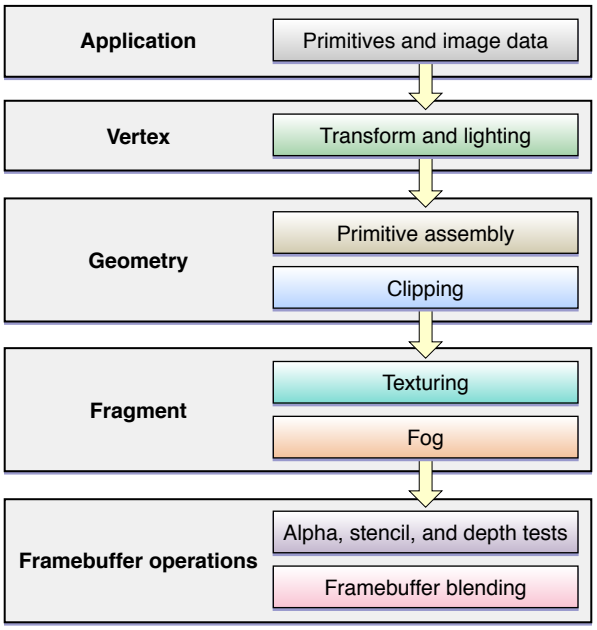
Figure 11-11 Double-buffered data



Customizing the OpenGL Pipeline with Shaders

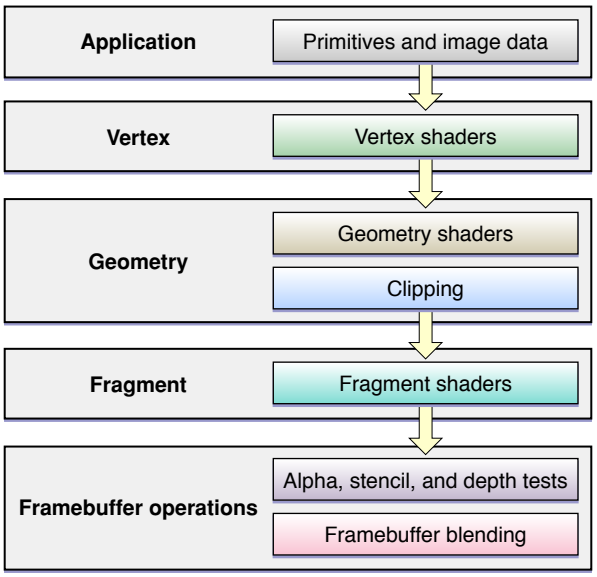
OpenGL 1.x used fixed functions to deliver a useful graphics pipeline to application developers. To configure the various stages of the pipeline shown in Figure 12-1, applications called OpenGL functions to tweak the calculations that were performed for each vertex and fragment. Complex algorithms required multiple rendering passes and dozens of function calls to configure the calculations that the programmer desired. Extensions offered new configuration options, but did not change the complex nature of OpenGL programming.

Figure 12-1 OpenGL fixed-function pipeline



Starting with OpenGL 2.0, some stages of the OpenGL pipeline can be replaced with **shaders**. A shader is a program written in a special shading language. This program is compiled by OpenGL and uploaded directly into the graphics hardware. Figure 12-2 shows where your applications can hook into the pipeline with shaders.

Figure 12-2 OpenGL shader pipeline



Shaders offer a considerable number of advantages to your application:

- Shaders give you precise control over the operations that are performed to render your images.
- Shaders allow for algorithms to be written in a terse, expressive format. Rather than writing complex blocks of configuration calls to implement a mathematical operation, you write code that expresses the algorithm directly.
- Older graphics processors implemented the fixed-function pipeline in hardware or microcode, but now graphics processors are general-purpose computing devices. The fixed function pipeline is itself implemented as a shader.
- Shaders allow for longer and more complex algorithms to be implemented using a single rendering pass. Because you have extensive control over the pipeline, it is also easier to implement multipass algorithms without requiring the data to be read back from the GPU.
- Your application can switch between different shaders with a single function call. In contrast, configuring the fixed-function pipeline incurs significant function-call overhead.

If your application uses the fixed-function pipeline, a critical task is to replace those tasks with shaders.

If you are new to shaders, *OpenGL Shading Language*, by Randi J. Rost, is an excellent guide for those looking to learn more about writing shaders and integrating them into your application. The rest of this chapter provides some boilerplate code, briefly describe the extensions that implement shaders, and discusses tools that Apple provides to assist you in writing shaders.

Shader Basics

OpenGL 2.0 offers vertex and fragment shaders, to take over the processing of those two stages of the graphics pipeline. These same capabilities are also offered by the ARB_shader_objects, ARB_vertex_shader and ARB_fragment_shaderextensions. Vertex shading is available on all hardware running OS X v10.5 or later. Fragment shading is available on all hardware running OS X v10.6 and the majority of hardware running OS X v10.5.

Creating a shader program is an expensive operation compared to other OpenGL state changes. Listing 12-1 presents a typical strategy to load, compile, and verify a shader program.

Listing 12-1 Loading a Shader

```
/** Initialization-time for shader **/
            GLuint shader, prog;
            GLchar *shaderText = "... shader text ...";

            // Create ID for shader
           shader = glCreateShader(GL_VERTEX_SHADER);

           // Define shader text
           glShaderSource(shaderText);

           // Compile shader
           glCompileShader(shader);

           // Associate shader with program
           glAttachShader(prog, shader);

          // Link program
           glLinkProgram(prog);

           // Validate program
           glValidateProgram(prog);

           // Check the status of the compile/link
           glGetProgramiv(prog, GL_INFO_LOG_LENGTH, &logLen);
           if(logLen > 0)
           {
               // Show any errors as appropriate
               glGetProgramInfoLog(prog, logLen, &logLen, log);
```

```
            fprintf(stderr, "Prog Info Log: %s\n", log);
    }

 // Retrieve all uniform locations that are determined during link phase
        for(i = 0; i < uniformCt; i++)
        {
            uniformLoc[i] = glGetUniformLocation(prog, uniformName);
        }

        // Retrieve all attrib locations that are determined during link phase

        for(i = 0; i < attribCt; i++)
        {
            attribLoc[i] = glGetAttribLocation(prog, attribName);
        }

/** Render stage for shaders **/
glUseProgram(prog);
```

This code loads the text source for a vertex shader, compiles it, and adds it to the program. A more complex example might also attach fragment and geometry shaders. The program is linked and validated for correctness. Finally, the program retrieves information about the inputs to the shader and stores then in its own arrays. When the application is ready to use the shader, it calls `glUseProgram` to make it the current shader.

For best performance, your application should create shaders when your application is initialized, and not inside the rendering loop. Inside your rendering loop, you can quickly switch in the appropriate shaders by calling `glUseProgram`. For best performance, use the vertex array object extension to also switch in the vertex pointers. See "Vertex Array Object" (page 116) for more information.

Advanced Shading Extensions

In addition to the standard shader, some Macs offer additional shading extensions to reveal advanced hardware capabilities. Not all of these extensions are available on all hardware, so you need to assess whether the features of each extension are worth implementing in your application.

Transform Feedback

The EXT_transform_feedback extension is available on all hardware running OS X v10.5 or later. With the feedback extension, you can capture the results of the vertex shader into a buffer object, which can be used as an input to future commands. This is similar to the pixel buffer object technique described in "Using Pixel Buffer Objects to Keep Data on the GPU" (page 124), but more directly captures the results you desire.

GPU Shader 4

The EXT_gpu_shader4 extension extends the OpenGL shading language to offer new operations, including:

- Full integer support.
- Built-in shader variable to reference the current vertex.
- Built-in shader variable to reference the current primitive. This makes it easier to use a shader to use the same static vertex data to render multiple primitives, using a shader and uniform variables to customize each instance of that primitive.
- Unfiltered texture fetches using integer coordinates.
- Querying the size of a texture within a shader.
- Offset texture lookups.
- Explicit gradient and LOD texture lookups.
- Depth Cubemaps.

Geometry Shaders

The EXT_geometry_shader4 extension allows your create geometry shaders. A geometry shader accepts transformed vertices and can add or remove vertices before passing them down to the rasterizer. This allows the application to add or remove geometry based on the calculated values in the vertex. For example, given a triangle and its neighboring vertices, your application could emit additional vertices to better create a more accurate appearance of a curved surface.

Uniform Buffers

The EXT_bindable_uniform extension allows your application to allocate buffer objects and use them as the source for uniform data in your shaders. Instead of relying on a single block of uniform memory supplied by OpenGL, your application allocates buffer objects using the same API that it uses to implement vertex buffer objects ("Vertex Buffers" (page 107)). Instead of making a function call for each uniform variable you want to change, you can swap all of the uniform data by binding to a different uniform buffer.

Techniques for Scene Antialiasing

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 with faster hardware and higher-resolution displays, there are several antialiasing techniques that can smooth edges to achieve a more realistic scene.

OpenGL supports antialiasing 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 antialiasing (FSAA). If your application needs point or line antialiasing instead of full scene antialiasing, use the built in OpenGL point and line antialiasing functions. These are described in Section 3.4.2 in the OpenGL Specification.

The three antialiasing 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 antialiasing. It describes multisampling and alpha channel sampling. The specification does not specifically mention supersampling but its wording doesn't preclude supersampling. The antialiasing methods that are available depend on the hardware and the actual implementation depends on the vendor. Some graphics cards support antialiasing 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 antialiasing technique you prefer. Hints are available as renderer attributes that you supply when you create a pixel format object.

A smaller subset of renderers support the EXT_framebuffer_blit and EXT_framebuffer_multisample extensions. These extensions allow your application to create multisampled offscreen frame buffer objects, render detailed scenes to them, with precise control over when the multisampled renderbuffer is resolved to a single displayable color per pixel.

Guidelines

Keep the following in mind when you set up full scene antialiasing:

- 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 antialiasing algorithm that operates on the full scene requires additional computing resources. There is a tradeoff between performance and quality. For that reason, you may want to provide a user interface that allows the user to enable and disable FSAA, or to choose the level of quality for antialiasing.
- 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 antialiasing. Other hardware mixes multisampling with supersampling techniques. The driver dictates the type of antialiasing 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`. You can also determine how many samples the renderer supports by calling `CGLDescribeRenderer` with the renderer property `kCGLRPMaxSamples`.

General Approach

The general approach to setting up full scene antialiasing is as follows:

1. Check to see what's supported. Not all renderers support the ARB multisample extension, so you need to check for this functionality (see “Detecting Functionality” (page 83)).

   To find out what type of antialiasing 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 antialiasing. Keep in mind that if you exclude unsupported hardware, the unsupported displays will not render anything. If you 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` or `kCGLPFASampleBuffers`) along with the number of multisample buffers. At this time the specification allows only one multisample buffer.
    - The appropriate samples constant (`NSOpenGLPFASamples` 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` or `kCGLPFANoRecovery`). Although enabling this attribute is not mandatory, it's recommended to prevent OpenGL from using software fallback as a renderer. Multisampled antialiasing performance is slow in the software renderer.
3. Optionally provide a hint for the type of antialiasing you want—multisampling, supersampling, or alpha sampling. See "Hinting for a Specific Antialiasing Technique" (page 147).
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.

Hinting for a Specific Antialiasing Technique

When you set up your renderer and buffer attributes for full scene antialiasing, you can specify a hint to prefer one antialiasing 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 13-1 lists the hinting constants available for the `NSOpenGLPixelFormat` class and CGL.

Table 13-1 Antialiasing hints

Multisampling	Supersampling	Alpha blending
`NSOpenGLPFAMultisample`	`NSOpenGLPFASupersample`	`NSOpenGLPFASampleAlpha`
`kCGLPFAMultisample`	`kCGLPFASupersample`	`kCGLPFASampleAlpha`

Concurrency and OpenGL

Concurrency is the notion of multiple things happening at the same time. In the context of computers, concurrency usually refers to executing tasks on more than one processor at the same time. By performing work in parallel, tasks complete sooner, and applications become more responsive to the user. The good news is that well-designed OpenGL applications already exhibit a specific form of concurrency—concurrency between application processing on the CPU and OpenGL processing on the GPU. Many of the techniques introduced in "OpenGL Application Design Strategies" (page 89) are aimed specifically at creating OpenGL applications that exhibit great CPU-GPU parallelism. However, modern computers not only contain a powerful GPU, but also contain multiple CPUs. Sometimes those CPUs have multiple cores, each capable of performing calculations independently of the others. It is critical that applications be designed to take advantage of concurrency where possible. Designing a concurrent application means decomposing the work your application performs into subtasks and identifying which tasks can safely operate in parallel and which tasks must be executed sequentially—that is, which tasks are dependent on either resources used by other tasks or results returned from those tasks.

Each process in OS X is made up of one or more threads. A **thread** is a stream of execution that runs code for the process. Multicore systems offer true concurrency by allowing multiple threads to execute simultaneously. Apple offers both traditional threads and a feature called **Grand Central Dispatch (GCD)**. Grand Central Dispatch allows you to decompose your application into smaller tasks without requiring the application to manage threads. GCD allocates threads based on the number of cores available on the system and automatically schedules tasks to those threads.

At a higher level, Cocoa offers `NSOperation` and `NSOperationQueue` to provide an Objective-C abstraction for creating and scheduling units of work. On OS X v10.6, operation queues use GCD to dispatch work; on OS X v10.5, operation queues create threads to execute your application's tasks.

This chapter does not attempt describe these technologies in detail. Before you consider how to add concurrency to your OpenGL application, you should first read*Concurrency Programming Guide*. If you plan on managing threads manually, you should also read *Threading Programming Guide*. Regardless of which technique you use, there are additional restrictions when calling OpenGL on multithreaded systems. This chapter helps you understand when multithreading improves your OpenGL application's performance, the restrictions OpenGL places on multithreaded applications, and common design strategies you might use to implement concurrency in an OpenGL application. Some of these design techniques can get you an improvement in just a few lines of code.

Identifying Whether an OpenGL Application Can Benefit from Concurrency

Creating a multithreaded application requires significant effort in the design, implementation, and testing of your application. Threads also add complexity and overhead to an application. For example, your application may need to copy data so that it can be handed to a worker thread, or multiple threads may need to synchronize access to the same resources. Before you attempt to implement concurrency in an OpenGL application, you should optimize your OpenGL code in a single-threaded environment using the techniques described in "OpenGL Application Design Strategies" (page 89). Focus on achieving great CPU-GPU parallelism first and then assess whether concurrent programming can provide an additional performance benefit.

A good candidate has either or both of the following characteristics:

- The application performs many tasks on the CPU that are independent of OpenGL rendering. Games, for example, simulate the game world, calculate artificial intelligence from computer-controlled opponents, and play sound. You can exploit parallelism in this scenario because many of these tasks are not dependent on your OpenGL drawing code.
- Profiling your application has shown that your OpenGL rendering code spends a lot of time in the CPU. In this scenario, the GPU is idle because your application is incapable of feeding it commands fast enough. If your CPU-bound code has already been optimized, you may be able to improve its performance further by splitting the work into tasks that execute concurrently.

If your application is blocked waiting for the GPU, and has no work it can perform in parallel with its OpenGL drawing commands, then it is not a good candidate for concurrency. If the CPU and GPU are both idle, then your OpenGL needs are probably simple enough that no further tuning is useful.

For more information on how to determine where your application spends its time, see "Tuning Your OpenGL Application" (page 155).

OpenGL Restricts Each Context to a Single Thread

Each thread in an OS X process has a single current OpenGL rendering context. Every time your application calls an OpenGL function, OpenGL implicitly looks up the context associated with the current thread and modifies the state or objects associated with that context.

OpenGL is not reentrant. If you modify the same context from multiple threads simultaneously, the results are unpredictable. Your application might crash or it might render improperly. 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*`. OpenGL commands that block—such as `fence` commands—do not synchronize threads.

GCD and `NSOperationQueue` objects can both execute your tasks on a thread of their choosing. They may create a thread specifically for that task, or they may reuse an existing thread. But in either case, you cannot guarantee which thread executes the task. For an OpenGL application, that means:

- Each task must set the context before executing any OpenGL commands.
- Your application must ensure that two tasks that access the same context are not allowed to execute concurrently.

Strategies for Implementing Concurrency in OpenGL Applications

A concurrent OpenGL application wants to focus on CPU parallelism so that OpenGL can provide more work to the GPU. Here are a few recommended strategies for implementing concurrency in an OpenGL application:

- Decompose your application into OpenGL and non-OpenGL tasks that can execute concurrently. Your OpenGL rendering code executes as a single task, so it still executes in a single thread. This strategy works best when your application has other tasks that require significant CPU processing.
- If performance profiling reveals that your application spends a lot of CPU time inside OpenGL, you can move some of that processing to another thread by enabling the multithreading in the OpenGL engine. The advantage of this method is its simplicity; enabling the multithreaded OpenGL engine takes just a few lines of code. See "Multithreaded OpenGL" (page 150).
- If your application spends a lot of CPU time preparing data to send to openGL, you can divide the work between tasks that prepare rendering data and tasks that submit rendering commands to OpenGL. See "Perform OpenGL Computations in a Worker Task" (page 151)
- If your application has multiple scenes it can render simultaneously or work it can perform in multiple contexts, it can create multiple tasks, with an OpenGL context per task. If the contexts can share the same resources, you can use context sharing when the contexts are created to share surfaces or OpenGL objects: display lists, textures, vertex and fragment programs, vertex array objects, and so on. See "Use Multiple OpenGL Contexts" (page 153)

Multithreaded OpenGL

Whenever your application calls OpenGL, the renderer processes the parameters to put them in a format that the hardware understands. The time required to process these commands varies depending on whether the inputs are already in a hardware-friendly format, but there is always some overhead in preparing commands for the hardware.

If your application spends a lot of time performing calculations inside OpenGL, and you've already taken steps to pick ideal data formats, your application might gain an additional benefit by enabling multithreading inside the OpenGL engine. The multithreaded OpenGL engine automatically creates a worker thread and transfers some of its calculations to that thread. On a multicore system, this allows internal OpenGL calculations performed on the CPU to act in parallel with your application, improving performance. Synchronizing functions continue to block the calling thread.

Listing 14-1 shows the code required to enable the multithreaded OpenGL engine.

Listing 14-1 Enabling the multithreaded OpenGL engine

```
CGLError err = 0;
CGLContextObj ctx = CGLGetCurrentContext();

// Enable the multithreading
err =  CGLEnable( ctx, kCGLCEMPEngine);

if (err != kCGLNoError )
{
     // Multithreaded execution may not be available
     // Insert your code to take appropriate action
}
```

> **Note:** Enabling or disabling multithreaded execution causes OpenGL to flush previous commands as well as incurring the overhead of setting up the additional thread. You should enable or disable multithreaded execution in an initialization function rather than in the rendering loop.

Enabling multithreading comes at a cost—OpenGL must copy parameters to transmit them to the worker thread. Because of this overhead, you should always test your application with and without multithreading enabled to determine whether it provides a substantial performance improvement.

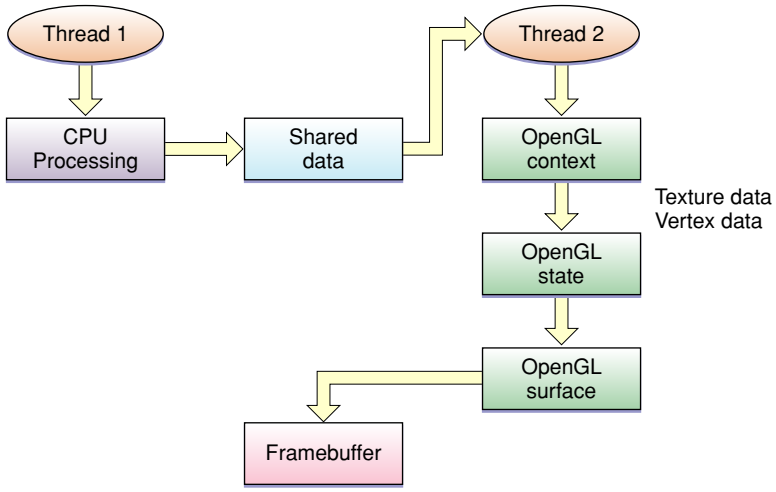
Perform OpenGL Computations in a Worker Task

Some applications perform lots of calculations on their data before passing that data down to the OpenGL renderer. For example, the application might create new geometry or animate existing geometry. Where possible, such calculations should be performed inside OpenGL. For example, vertex shaders and the transform

feedback extension might allow you to perform these calculations entirely within OpenGL. This takes advantage of the greater parallelism available inside the GPU, and reduces the overhead of copying results between your application and OpenGL.

The approach described in Figure 9-3 (page 92) alternates between updating OpenGL objects and executing rendering commands that use those objects. OpenGL renders on the GPU in parallel with your application's updates running on the CPU. If the calculations performed on the CPU take more processing time than those on the GPU, then the GPU spends more time idle. In this situation, you may be able to take advantage of parallelism on systems with multiple CPUs. Split your OpenGL rendering code into separate calculation and processing tasks, and run them in parallel. Figure 14-1 shows a clear division of labor. One task produces data that is consumed by the second and submitted to OpenGL.

Figure 14-1 CPU processing and OpenGL on separate threads



For best performance, your application should avoid copying data between the tasks. For example, rather than calculating the data in one task and copying it into a vertex buffer object in the other, map the vertex buffer object in the setup code and hand the pointer directly to the worker task.

If your application can further decompose the modifications task into subtasks, you may see better benefits. For example, assume two or more vertex buffers, each of which needs to be updated before submitting drawing commands. Each can be recalculated independently of the others. In this scenario, the modifications to each buffer becomes an operation, using an `NSOperationQueue` object to manage the work:

1. Set the current context.
2. Map the first buffer.
3. Create an `NSOperation` object whose task is to fill that buffer.
4. Queue that operation on the operation queue.

5. Perform steps 2 through 4 for the other buffers.
6. Call `waitUntilAllOperationsAreFinished` on the operation queue.
7. Unmap the buffers.
8. Execute rendering commands.

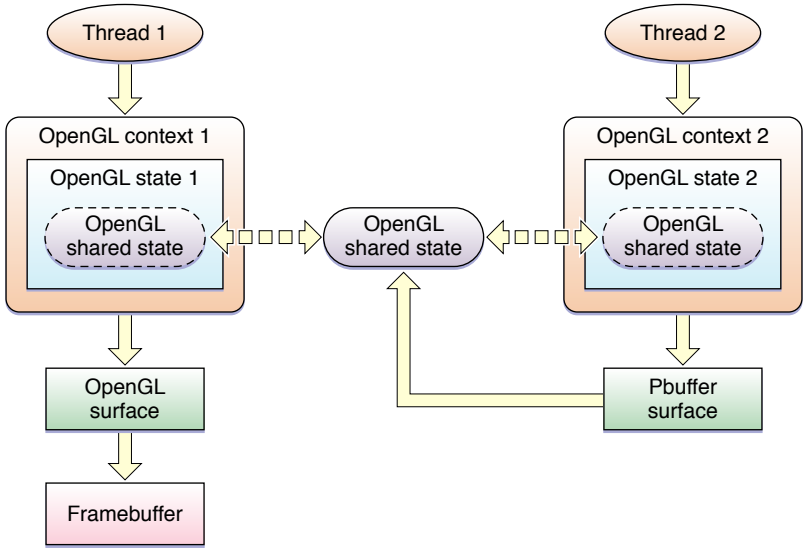
On a multicore system, multiple threads of execution may allow the buffers to be filled simultaneously. Steps 7 and 8 could even be performed by a separate operation queued onto the same operation queue, provided that operation set the proper dependencies.

Use Multiple OpenGL Contexts

If your application has multiple scenes that can be rendered in parallel, you can use a context for each scene you need to render. Create one context for each scene and assign each context to an operation or task. Because each task has its own context, all can submit rendering commands in parallel.

The Apple-specific OpenGL APIs also provide the option for sharing data between contexts, as shown in Figure 14-2. 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 14-2 Two contexts on separate threads



This is the most complex model for designing an application. Changes to objects in one context must be flushed so that other contexts see the changes. Similarly, when your application finishes operating on an object, it must flush those commands before exiting, to ensure that all rendering commands have been submitted to the hardware.

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 thread safe. You should never have more than one thread accessing a single context simultaneously.
- 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. 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. Mutex locking is necessary because 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`. 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.

  `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 Khronos OpenGL Working Group) 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.

Tuning Your OpenGL Application

After you design and implement your application, it is important that you spend some time analyzing its performance. The key to performance tuning your OpenGL application is to successively refine the design and implementation of your application. You do this by alternating between measuring your application, identifying where the bottleneck is, and removing the bottleneck.

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 OS X.

Next, take a close look at Instruments. Instruments consolidates many measurement tools into a single comprehensive performance-tuning application.

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
- Optimizing with Shark: Big Payoff, Small Effort
- *Instruments User Guide*
- *Shark User Guide*
- *Real world profiling with the OpenGL Profiler*
- *OpenGL Driver Monitor User Guide*
- *OpenGL Shader Builder User Guide*

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. In particular, Graphics Pipeline Performance is a critical article for understanding how to find the bottlenecks in your OpenGL application.
- *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Matt Pharr and Randima Fernando.

This chapter focuses on two main topics:

- "Gathering and Analyzing Baseline Performance Data" (page 156) shows how to use `top` and OpenGL Profiler to obtain and interpret baseline performance data.
- "Identifying Bottlenecks with Shark" (page 161) 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.

Gathering and Analyzing Baseline Performance Data

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

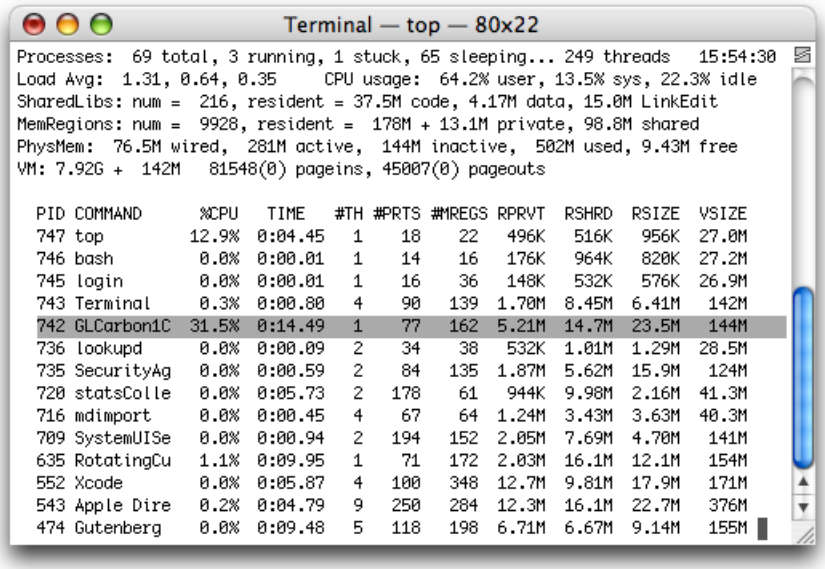
- `top` is a command-line utility that you run in the Terminal window. You can use `top` to assess how much CPU time your application consumes.
- OpenGL Profiler is an application that determines how much time an application spends in OpenGL. It also provides function traces that you can use to look 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.

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 15-1.

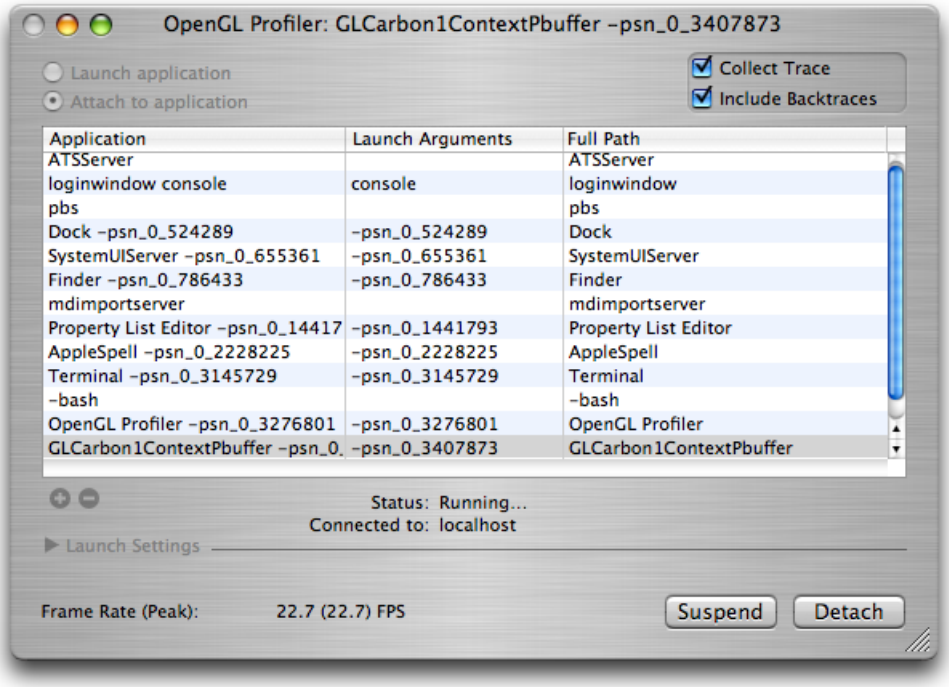
The `top` program indicates the amount of CPU time that an application uses. The CPU time serves as a good baseline value for gauging how much tuning your code needs. Figure 15-1 shows the percentage of CPU time for the OpenGL application GLCarbon1C (highlighted). Note this application utilizes 31.5% of CPU resources.

Figure 15-1 Output produced by the `top` application



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 15-2.

Figure 15-2 The OpenGL Profiler window



5. Select the option "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.

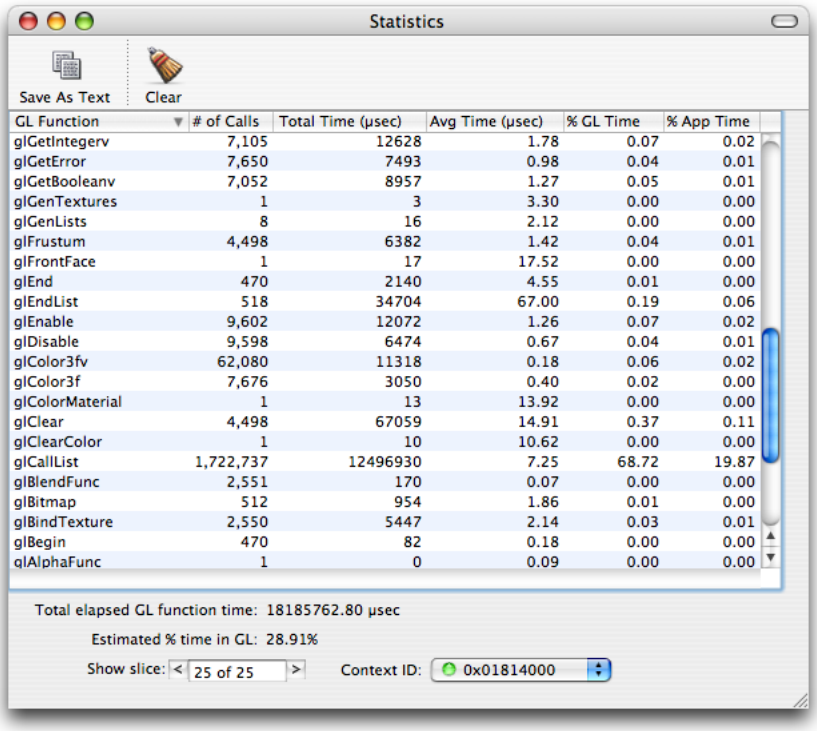
7. Open the Statistics and Trace windows by choosing them from the Views menu.

   Figure 15-3 provides an example of what the Statistics window looks like. Figure 15-4 (page 160) shows a Trace window.

   The estimated percentage of time spent in OpenGL is shown at the bottom of Figure 15-3. 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 15-3 A statistics window



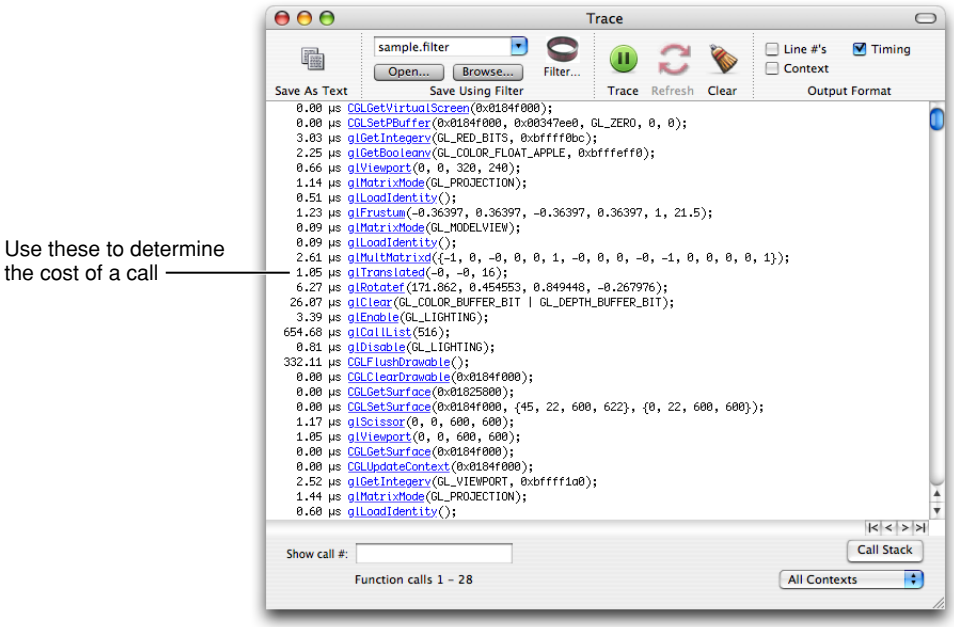
8. In the Trace window, look for duplicate function calls and redundant or unnecessary state changes.

   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 called only 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 15-4 A Trace window



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:

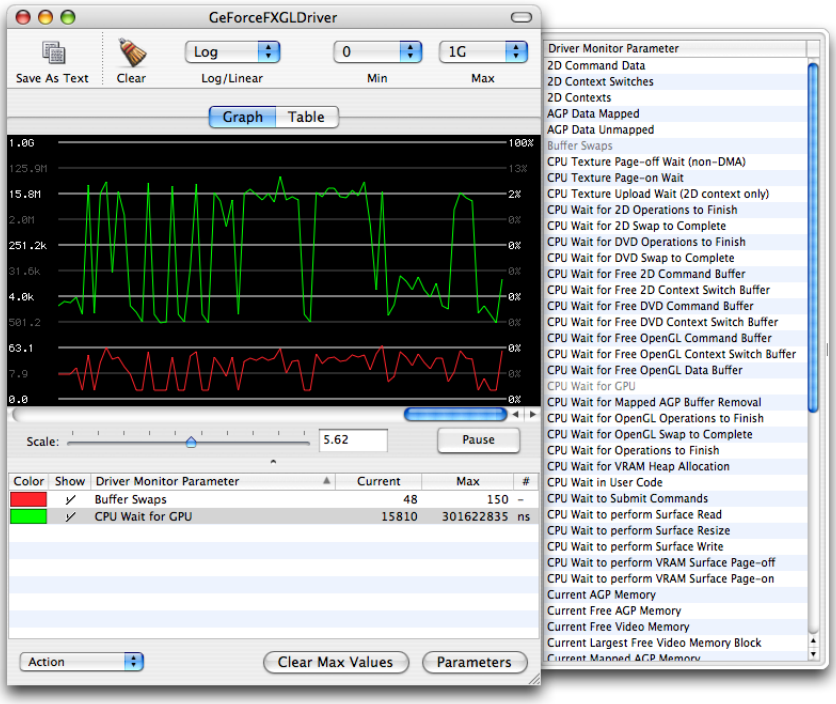
   ```
   New FPS = previous FPS * (1 +(% performance increase)) = 60 fps *(1.0911) =
   65.47 fps
   ```

   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.

Using OpenGL Driver Monitor to Measure Stalls

You can use OpenGL Driver Monitor to measure how long the CPU waits for the GPU, as shown in Figure 15-5. 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 15-5 The graph view in OpenGL Driver Monitor



Identifying Bottlenecks with Shark

Shark is an extremely useful tool for identifying places in your code that are slow and could benefit from optimization. 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 102).

> **Note:** If your data needs only to be swizzled, `glgProcessPixels` performs the swizzling reasonably fast although not as fast as if the data didn't need swizzling. But non-native data formats are converted one byte at a time and incurs 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 see this during a texture upload, for example.
- 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.

Legacy OpenGL Functionality by Version

OpenGL functionality changes with each 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 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 Renderer" (page 83).

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`

Functionality	Extension
Texture objects	GL_EXT_texture_object
Texture proxies	GL_EXT_texture
Texture replace environment	GL_EXT_texture
Vertex array	GL_EXT_vertex_array

There were 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`
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`

Functionality	Extension
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

Functionality	Extension
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

Version 2.1

Table A-7 Functionality added in OpenGL 2.1

Functionality	Extension
Pixel buffer objects	GL_ARB_pixel_buffer_object
sRGB textures	GL_EXT_texture_sRGB

Updating an Application to Support the OpenGL 3.2 Core Specification

The OpenGL 3.0 specification deprecated many areas of functionality defined in earlier versions of the OpenGL specification. The OpenGL 3.2 Core profile explicitly removes these deprecated features and adjusts other parts of the specification to provide a streamlined, clean programming interface to OpenGL. Use this chapter to assist you in migrating your application away from this deprecated functionality.

Removed Functionality

The features that were removed from OpenGL are described in in Appendix E of the OpenGL 3.2 Core specification, and you should use that as the definitive guide for the changes you need to make in your application. Here is a summary of most significant areas that changed:

- If your application uses the fixed-function pipeline, it must be rewritten to use shaders instead.
- If your application uses shaders, you must rewrite your shaders to use OpenGL Shading Language 1.5; many built-in shader variables provided in earlier versions of the OpenGL Shading Language were explicitly removed from the OpenGL Shading Language 1.5 specification. Similarly, your application may no longer provide vertex data using the fixed-function routines; all vertex attributes are now specified as generic vertex attributes.
- Your application must explicitly generate object names using the OpenGL API.
- Vertex data must be provided to OpenGL using buffer objects.
- The built-in matrix stack functionality from earlier versions of OpenGL has been removed; you must recreate this functionality using shader inputs.
- Support for auxiliary and accumulation buffers has been removed; use framebuffer objects instead.
- Your application no longer fetches the list of extensions as a single string. Instead, you first fetch the number of extensions and then separately fetch each extension string.

Extension Changes on OS X

OpenGL 3.2 provides functionality that earlier versions of OpenGL provided through extensions. Other extensions that were previously supported on OS X are no longer supported when your application uses the OpenGL 3.2 Core profile. Table B-1 lists extensions described elsewhere in this guide; use this table to determine whether the extension is supported, and if not, what equivalent functionality is supported.

Table B-1 Extensions described in this guide

Extension	Status
APPLE_fence	Obsolete. Use the `ARB_Sync` functionality provided by OpenGL 3.2 (Core).
ARB_vertex_buffer_object	Functionality provided by OpenGL 3.2 (Core).
APPLE_vertex_array_object	Obsolete. Use the `ARB_vertex_array_object` functionality provided by OpenGL 3.2 (Core).
APPLE_vertex_array_range	Obsolete. Use the `ARB_map_buffer_range` functionality provided by OpenGL 3.2 (Core).
APPLE_flush_buffer_range	Obsolete. Use the `ARB_map_buffer_range` functionality provided by OpenGL 3.2 (Core).
APPLE_client_storage	Supported.
APPLE_texture_range	Supported.
ARB_texture_rectangle	Functionality provided by OpenGL 3.2 (Core).
ARB_shader_objects	Functionality provided by OpenGL 3.2 (Core).
ARB_vertex_shader	Functionality provided by OpenGL 3.2 (Core).
ARB_fragment_shader	Functionality provided by OpenGL 3.2 (Core).
EXT_transform_feedback	Functionality provided by OpenGL 3.2 (Core).
EXT_gpu_shader4	Obsolete. Functionality included in GLSL 1.5
EXT_geometry_shader4	Functionality provided by OpenGL 3.2 (Core).
EXT_bindable_uniform	Obsolete. Use the `ARB_uniform_buffer_object` functionality provided by OpenGL 3.2 (Core).
ARB_pixel_buffer_object	Functionality provided by OpenGL 3.2 (Core).

Extension	Status
EXT_framebuffer_object	Obsolete. Use the `ARB_framebuffer_object` functionality provided by OpenGL 3.2 (Core).
APPLE_pixel_buffer	Obsolete. Use framebuffer objects instead.
NV_multisample_filter_hint	Obsolete. Use multisampled renderbuffers to precisely control multisampling.

Setting Up Function Pointers to OpenGL Routines

Function pointers to OpenGL routines allow you to deploy your application across multiple versions of 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 OS X and Windows.

> **Note:** If you are deploying your application only in 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 171) shows how to write a generic routine that you can reuse for any OpenGL application on the Macintosh platform.
- "Initializing Entry Points" (page 172) 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 Cocoa. You can use the Dynamic Loader function `NSLookupAndBindSymbol` to get the address of an OpenGL entry point.

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 83).

Listing C-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 C-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.

Initializing Entry Points

Listing C-2 shows how to use the `MyNSGLGetProcAddress` function from Listing C-1 (page 172) to obtain a few OpenGL entry points. A detailed explanation for each numbered line of code appears following the listing.

Listing C-2 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 C-1 (page 172).

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.

Document Revision History

This table describes the changes to *OpenGL Programming Guide for Mac*.

Date	Notes
2012-07-23	Updated with information on supporting high-resolution displays.
2011-06-06	Added new context options.
2010-11-15	Fixed a few small errors in the texture chapter. Updated the recommendations on when to use each texture uploading and downloading technique. Updated the code for creating a texture from a view's contents to use newer, better supported techniques.
2010-06-14	Corrected texture creation code snippets.
2010-03-24	Minor updates and clarifications.
2010-02-24	Substantial revisions to describe behaviors for OpenGL on OS X v10.5 and OS X v10.6. Removed information on obsolete and deprecated behaviors.
2009-08-28	Corrected errors in code listings. Pixel format attribute lists should be terminated with 0, not NULL. One call to glTexImage2D had an incorrect number of parameters.
2008-06-09	Updated the Cocoa OpenGL tutorial and made numerous other minor changes. Fixed compilation errors in Listing 8-1 (page 84). Added "Getting Decompressed Raw Pixel Data from a Source Image" (page 135). Updated links to OpenGL extensions.

Date	Notes
	Made several minor edits.
2007-12-04	Corrected minor typographical and technical errors. Added "Ensuring That Back Buffer Contents Remain the Same" (page 66). Revised "Deprecated Attributes" (page 70).
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 6-2 (page 68). Fixed drawRect method implementation in "Drawing to a Window or View" (page 35).
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 "Determining Whether Vertex and Fragment Processing Happens on the GPU" (page 78).
2006-10-03	Added a section on checking for GPU processing. Added "Determining Whether Vertex and Fragment Processing Happens on the GPU" (page 78). Fixed a number of minor typos in the code and in the text.
2006-09-05	Fixed minor technical problems.
2006-07-24	Made minor technical and typograhical changes throughout. Added information to "Surface Drawing Order Specifies the Position of the OpenGL Surface Relative to the Window" (page 77).

Date	Notes
	Changed `glCopyTexSubImage` to `glCopyTexSubImage2D` in "Downloading Texture Data" (page 136). Made minor improvements to Listing 11-6 (page 136). 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 166) 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"

Date	Notes
	Technical Q&A QA1209 “Updating OpenGL Contexts”
	Technical Q&A QA1248 “Context Sharing Tips”
	Technical Q&A QA1268 “Sharpening Full Scene Anti-Aliasing Details”
	Technical Q&A QA1269 “OS X OpenGL Interfaces”
	Technical Q&A QA1325 “Creating an OpenGL texture from an NSView”

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 Khronos OpenGL Working Group

aliased Said of graphics whose edges appear jagged; can be remedied by performing antialiasing operations.

antialiasing 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 Khronos OpenGL Working Group, which is the group that oversees the OpenGL specification and extensions to it.

attach To establish a connection between two existing objects. Compare bind.

bind To create a new object and then establish a connection between that object and a rendering context. Compare attach.

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, and color index values.

CGL (Core OpenGL) framework The Apple framework for using OpenGL graphics in OS X applications 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 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 Khronos OpenGL Working Group 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 The region of space that is seen by the observer and that is warped by perspective division.

FSAA (full scene antialiasing) 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 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 (nonuniform rational basis 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 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 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.

stencil buffer Memory used specifically for stencil testing. A stencil test is typically used to identify masking regions, to 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 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-, or 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 color 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.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010