OpenGL ES Programming Guide for iOS

Developer

Contents

About OpenGL ES 10
At a Glance 10
OpenGL ES Is a Platform-Neutral API Implemented in iOS 11
GLKit Provides a Drawing Surface and Animation Support 11
iOS Supports Alternative Rendering Targets 11
Apps Require Additional Performance Tuning 12
OpenGL ES May Not Be Used in Background Apps 12
OpenGL ES Places Additional Restrictions on Multithreaded Apps 12
How to Use This Document 13
Prerequisites 13
See Also 13

Checklist for Building OpenGL ES Apps for iOS 15
Choosing Which OpenGL ES Versions to Support 15
Verifying OpenGL ES Capabilities 16
Choosing a Rendering Destination 17
Integrating with iOS 17
Implementing a Rendering Engine 17
Debugging and Profiling 18

Configuring OpenGL ES Contexts 19
EAGL Is the iOS Implementation of an OpenGL ES Rendering Context 19
The Current Context Is the Target for OpenGL ES Function Calls 19
Every Context Targets a Specific Version of OpenGL ES 20
An EAGL Sharegroup Manages OpenGL ES Objects for the Context 21

Drawing with OpenGL ES and GLKit 23
A GLKit View Draws OpenGL ES Content on Demand 23
Creating and Configuring a GLKit View 24
Drawing With a GLKit View 25
Rendering Using a Delegate Object 26
A GLKit View Controller Animates OpenGL ES Content 28
Understanding the Animation Loop 28
Using a GLKit View Controller 29

Using GLKit to Develop Your Renderer 31
Handling Vector and Matrix Math 31
Migrating from the OpenGL ES 1.1 Fixed-Function Pipeline 31
Loading Texture Data 31

Drawing to Other Rendering Destinations 32
Creating a Framebuffer Object 32
Creating Offscreen Framebuffer Objects 33
Using Framebuffer Objects to Render to a Texture 34
Rendering to a Core Animation Layer 35
Drawing to a Framebuffer Object 37
Rendering on Demand or with an Animation Loop 37
Rendering a Frame 38
Using Multisampling to Improve Image Quality 40

Multitasking, High Resolution, and Other iOS Features 43
Implementing a Multitasking-Aware OpenGL ES App 43
Background Apps May Not Execute Commands on the Graphics Hardware 43
Delete Easily Re-Created Resources Before Moving to the Background 44
Supporting High-Resolution Displays 45
Supporting Multiple Interface Orientations 46
Presenting OpenGL ES Content on External Displays 46

OpenGL ES Design Guidelines 48
How to Visualize OpenGL ES 48
OpenGL ES as a Client-Server Architecture 48
OpenGL ES as a Graphics Pipeline 49
OpenGL ES Versions and Renderer Architecture 50
OpenGL ES 3.0 50
OpenGL ES 2.0 55
OpenGL ES 1.1 55
Designing a High-Performance OpenGL ES App 55
Avoid Synchronizing and Flushing Operations 57
Using glFlush Effectively 58
Avoid Querying OpenGL ES State 58
Use OpenGL ES to Manage Your Resources 59
Use Double Buffering to Avoid Resource Conflicts 59
Be Mindful of OpenGL ES State 61
Encapsulate State with OpenGL ES Objects 61
Organize Draw Calls to Minimize State Changes 61

Tuning Your OpenGL ES App 62
Debug and Profile Your App with Xcode and Instruments 62
Watch for OpenGL ES Errors in Xcode and Instruments 63
Annotate Your OpenGL ES Code for Informative Debugging and Profiling 64
General Performance Recommendations 65
Redraw Scenes Only When the Scene Data Changes 65
Disable Unused OpenGL ES Features 66
Simplify Your Lighting Models 66
Use Tile-Based Deferred Rendering Efficiently 66
Avoid Logical Buffer Loads and Stores 67
Use Hidden Surface Removal Effectively 67
Group OpenGL ES Commands for Efficient Resource Management 69
Minimize the Number of Drawing Commands 69
Use Instanced Drawing to Minimize Draw Calls 70
Minimize OpenGL ES Memory Usage 72
Be Aware of Core Animation Compositing Performance 72

Best Practices for Working with Vertex Data 74
Simplify Your Models 75
Avoid Storing Constants in Attribute Arrays 76
Use the Smallest Acceptable Types for Attributes 76
Use Interleaved Vertex Data 77
Avoid Misaligned Vertex Data 77
Use Triangle Strips to Batch Vertex Data 78
Use Vertex Buffer Objects to Manage Copying Vertex Data 80
Buffer Usage Hints 82
Consolidate Vertex Array State Changes Using Vertex Array Objects 84
Map Buffers into Client Memory for Fast Updates 86

Best Practices for Working with Texture Data 89
Load Textures During Initialization 89
Use the GLKit Framework to Load Texture Data 89
Reduce Texture Memory Usage 91
Compress Textures 91
Use Lower-Precision Color Formats 91
Use Properly Sized Textures 91
Combine Textures into Texture Atlases 92
Use Mipmapping to Reduce Memory Bandwidth Usage 93
Use Multitexturing Instead of Multiple Passes 93

Best Practices for Shaders 94
Compile and Link Shaders During Initialization 94
Check for Shader Program Errors When Debugging 94
Use Separate Shader Objects to Speed Compilation and Linking 95
Respect the Hardware Limits on Shaders 96
Use Precision Hints 97
Perform Vector Calculations Lazily 98
Use Uniforms or Constants Instead of Computing Values in a Shader 99
Use Branching Instructions with Caution 99
Eliminate Loops 99
Avoid Computing Array Indices in Shaders 100
Be Aware of Dynamic Texture Lookups 100
Fetch Framebuffer Data for Programmable Blending 101
Using Framebuffer Fetch in GLSL ES 1.0 102
Using Framebuffer Fetch in GLSL ES 3.0 103
Use Textures for Larger Memory Buffers in Vertex Shaders 104

Concurrency and OpenGL ES 106
Deciding Whether You Can Benefit from Concurrency 106
OpenGL ES Restricts Each Context to a Single Thread 107
Strategies for Implementing Concurrency in OpenGL ES Apps 108
Multithreaded OpenGL ES 108
Perform OpenGL ES Computations in a Worker Task 109
Use Multiple OpenGL ES Contexts 110
Guidelines for Threading OpenGL ES Apps 110

Adopting OpenGL ES 3.0 111
Checklist for Adopting OpenGL ES 3.0 111
Updating Extension Code 112
Remove Extension Suffixes 112
Modify Use of Extension APIs 112
Continue Using Most Other Extensions in OpenGL ES 3.0 114
Adopting OpenGL ES Shading Language version 3.0 114

Xcode OpenGL ES Tools Overview 116
Using the FPS Debug Gauge and GPU Report 116
Capturing and Analyzing an OpenGL ES Frame 118
Touring the OpenGL ES Frame Debugger 120
Navigator Area 121
Editor Area 122

Debug Area 128

Using texturetool to Compress Textures 131
texturetool Parameters 131

Document Revision History 136

Glossary 138

Figures and Listings

Configuring OpenGL ES Contexts 19

Figure 2-1 Two contexts sharing OpenGL ES objects 21
Listing 2-1 Supporting multiple versions of OpenGL ES in the same app 20
Listing 2-2 Creating two contexts with a common sharegroup 22

Drawing with OpenGL ES and GLKit 23

Figure 3-1 Rendering OpenGL ES content with a GLKit view 24
Figure 3-2 The animation loop 28
Listing 3-1 Configuring a GLKit view 25
Listing 3-2 Example drawing method for a GLKit view 25
Listing 3-3 Choosing a renderer class based on hardware features 27
Listing 3-4 Using a GLKit view and view controller to draw and animate OpenGL ES content 29

Drawing to Other Rendering Destinations 32

Figure 4-1 Framebuffer with color and depth renderbuffers 32
Figure 4-2 Core Animation shares the renderbuffer with OpenGL ES 35
Figure 4-3 iOS OpenGL Rendering Steps 38
Figure 4-4 How multisampling works 41
Listing 4-1 Creating and starting a display link 37
Listing 4-2 Clear framebuffer attachments 38
Listing 4-3 Discarding the depth framebuffer 39
Listing 4-4 Presenting the finished frame 40
Listing 4-5 Creating the multisample buffer 41

OpenGL ES Design Guidelines 48

Figure 6-1 OpenGL ES client-server architecture 48
Figure 6-2 OpenGL ES graphics pipeline 49
Figure 6-3 Example of fragment shader output to multiple render targets 51
Figure 6-4 Overview of a particle system animation 53
Figure 6-5 Example graphics pipeline configuration using transform feedback 54
Figure 6-6 App model for managing resources 56
Figure 6-7 Single-buffered texture data 59
Figure 6-8 Double-buffered texture data 60
Listing 6-1 Setting up multiple render targets 51

Listing 6-2 Fragment shader with output to multiple render targets 52

Tuning Your OpenGL ES App 62
Figure 7-1 Xcode Frame Debugger before and after adding debug marker groups 64
Figure 7-2 Trim transparent objects to reduce fragment processing 68
Listing 7-1 Using a debug marker to annotate drawing commands 64
Listing 7-2 Using a debug label to annotate an OpenGL ES object 65
Listing 7-3 Drawing many similar objects without instancing 70
Listing 7-4 OpenGL ES 3.0 vertex shader using `gl_InstanceID` to compute per-instance information 70
Listing 7-5 Using a vertex attribute for per-instance information 71
Listing 7-6 OpenGL ES 3.0 vertex shader using instanced arrays 71

Best Practices for Working with Vertex Data 74
Figure 8-1 Conversion of attribute data to shader variables 74
Figure 8-2 Interleaved memory structures place all data for a vertex together in memory 77
Figure 8-3 Use multiple vertex structures when some data is used differently 77
Figure 8-4 Align Vertex Data to avoid additional processing 78
Figure 8-5 Triangle strip 78
Figure 8-6 Use degenerate triangles to merge triangle strips 79
Figure 8-7 Vertex array object configuration 85
Listing 8-1 Using primitive restart in OpenGL ES 3.0 79
Listing 8-2 Submitting vertex data to a shader program 80
Listing 8-3 Creating a vertex buffer object 81
Listing 8-4 Drawing with a vertex buffer object 81
Listing 8-5 Drawing a model with multiple vertex buffer objects 83
Listing 8-6 Configuring a vertex array object 85
Listing 8-7 Dynamically updating a vertex buffer with manual synchronization 87

Best Practices for Working with Texture Data 89
Listing 9-1 Loading a two-dimensional texture from a file 90

Best Practices for Shaders 94
Figure 10-1 Traditional fixed-function blending 101
Figure 10-2 Programmable blending with framebuffer fetch 101
Listing 10-1 Read shader compile/link logs only in development builds 94
Listing 10-2 Compiling and using separate shader objects 95
Listing 10-3 Low precision is acceptable for fragment color 97
Listing 10-4 Poor use of vector operators 98
Listing 10-5 Proper use of vector operations 98

Listing 10-6 Specifying a write mask 98
Listing 10-7 Dependent Texture Read 100
Listing 10-8 Fragment shader for programmable blending in GLSL ES 1.0 102
Listing 10-9 Fragment shader for color post-processing in GLSL ES 3.0 103
Listing 10-10 Vertex shader for rendering from a height map 104

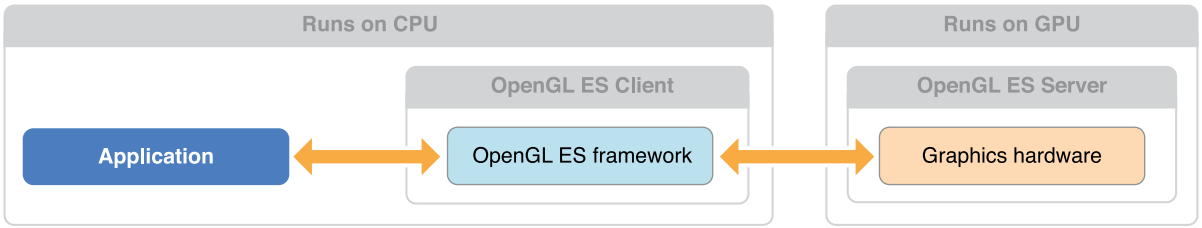
Xcode OpenGL ES Tools Overview 116
Figure B-1 FPS Debug Gauge and GPU Report 116
Figure B-2 Debug Bar with Capture OpenGL ES Frame button 118
Figure B-3 Frame debugger examining draw calls and resources 120
Figure B-4 Frame debugger examining shader program performance and analysis results 121
Figure B-5 View Frame By popup menu in navigator 121
Figure B-6 Framebuffer info popover 123
Figure B-7 Framebuffer settings popover 124
Figure B-8 GLSL shader source editor with update button 125
Figure B-9 Assistant editor previewing array buffer contents 126
Figure B-10 Assistant editor previewing vertex array object 126
Figure B-11 Assistant editor previewing cube map texture 127
Figure B-12 OpenGL ES debug bar 128
Figure B-13 Debug area with GL Context and Bound GL Objects views 129
Figure B-14 Debug area with Auto and Context Info views 130

Using texturetool to Compress Textures 131
Listing C-1 Encoding options 132
Listing C-2 Encoding images into the PVRTC compression format 134
Listing C-3 Encoding images into the PVRTC compression format while creating a preview 135

About OpenGL ES

The **Open Graphics Library (OpenGL)** is used for visualizing 2D and 3D data. It is a multipurpose open-standard graphics library that supports applications for 2D and 3D digital content creation, mechanical and architectural design, virtual prototyping, flight simulation, video games, and more. You use OpenGL to configure a 3D graphics pipeline and submit data to it. Vertices are transformed and lit, assembled into primitives, and rasterized to create a 2D image. OpenGL is designed to translate function calls into graphics commands that can be sent to underlying graphics hardware. Because this underlying hardware is dedicated to processing graphics commands, OpenGL drawing is typically very fast.

OpenGL for Embedded Systems (OpenGL ES) is a simplified version of OpenGL that eliminates redundant functionality to provide a library that is both easier to learn and easier to implement in mobile graphics hardware.



At a Glance

OpenGL ES allows an app to harness the power of the underlying graphics processor. The GPU on iOS devices can perform sophisticated 2D and 3D drawing, as well as complex shading calculations on every pixel in the final image. You should use OpenGL ES if the design requirements of your app call for the most direct and comprehensive access possible to GPU hardware. Typical clients for OpenGL ES include video games and simulations that present 3D graphics.

OpenGL ES is a low-level, hardware-focused API. Though it provides the most powerful and flexible graphics processing tools, it also has a steep learning curve and a significant effect on the overall design of your app. For apps that require high-performance graphics for more specialized uses, iOS provides several higher-level frameworks:

- The Sprite Kit framework provides a hardware-accelerated animation system optimized for creating 2D games. (See *Sprite Kit Programming Guide*.)

- The Core Image framework provides real-time filtering and analysis for still and video images. (See *Core Image Programming Guide*.)
- Core Animation provides the hardware-accelerated graphics rendering and animation infrastructure for all iOS apps, as well as a simple declarative programming model that makes it simple to implement sophisticated user interface animations. (See *Core Animation Programming Guide*.)
- You can add animation, physics-based dynamics, and other special effects to Cocoa Touch user interfaces using features in the UIKit framework.

OpenGL ES Is a Platform-Neutral API Implemented in iOS

Because OpenGL ES is a C-based API, it is extremely portable and widely supported. As a C API, it integrates seamlessly with Objective-C Cocoa Touch apps. The OpenGL ES specification does not define a windowing layer; instead, the hosting operating system must provide functions to create an OpenGL ES **rendering context**, which accepts commands, and a **framebuffer**, where the results of any drawing commands are written to. Working with OpenGL ES on iOS requires using iOS classes to set up and present a drawing surface and using platform-neutral API to render its contents.

Relevant Chapters: "Checklist for Building OpenGL ES Apps for iOS" (page 15), "Configuring OpenGL ES Contexts" (page 19)

GLKit Provides a Drawing Surface and Animation Support

Views and view controllers, defined by the UIKit framework, control the presentation of visual content on iOS. The GLKit framework provides OpenGL ES–aware versions of these classes. When you develop an OpenGL ES app, you use a `GLKView` object to render your OpenGL ES content. You can also use a `GLKViewController` object to manage your view and support animating its contents.

Relevant Chapters: "Drawing with OpenGL ES and GLKit" (page 23)

iOS Supports Alternative Rendering Targets

Besides drawing content to fill an entire screen or part of a view hierarchy, you can also use OpenGL ES framebuffer objects for other rendering strategies. iOS implements standard OpenGL ES framebuffer objects, which you can use for rendering to an offscreen buffer or to a texture for use elsewhere in an OpenGL ES scene. In addition, OpenGL ES on iOS supports rendering to a Core Animation layer (the `CAEAGLLayer` class), which you can then combine with other layers to build your app's user interface or other visual displays.

Relevant Chapters: "Drawing to Other Rendering Destinations" (page 32)

Apps Require Additional Performance Tuning

Graphics processors are parallelized devices optimized for graphics operations. To get great performance in your app, you must carefully design your app to feed data and commands to OpenGL ES so that the graphics hardware runs in parallel with your app. A poorly tuned app forces either the CPU or the GPU to wait for the other to finish processing commands.

You should design your app to efficiently use the OpenGL ES API. Once you have finished building your app, use Instruments to fine tune your app's performance. If your app is bottlenecked inside OpenGL ES, use the information provided in this guide to optimize your app's performance.

Xcode provides tools to help you improve the performance of your OpenGL ES apps.

Relevant Chapters: "OpenGL ES Design Guidelines" (page 48), "Best Practices for Working with Vertex Data" (page 74), "Best Practices for Working with Texture Data" (page 89), "Best Practices for Shaders" (page 94), "Tuning Your OpenGL ES App" (page 62)

OpenGL ES May Not Be Used in Background Apps

Apps that are running in the background may not call OpenGL ES functions. If your app accesses the graphics processor while it is in the background, it is automatically terminated by iOS. To avoid this, your app should flush any pending commands previously submitted to OpenGL ES prior to being moved into the background and avoid calling OpenGL ES until it is moved back to the foreground.

Relevant Chapters: "Multitasking, High Resolution, and Other iOS Features" (page 43)

OpenGL ES Places Additional Restrictions on Multithreaded Apps

Designing apps to take advantage of concurrency can be useful to help improve your app's performance. If you intend to add concurrency to an OpenGL ES app, you must ensure that it does not access the same context from two different threads at the same time.

Relevant Chapters: "Concurrency and OpenGL ES" (page 106)

How to Use This Document

Begin by reading the first three chapters: "Checklist for Building OpenGL ES Apps for iOS" (page 15), "Configuring OpenGL ES Contexts" (page 19), "Drawing with OpenGL ES and GLKit" (page 23). These chapters provide an overview of how OpenGL ES integrates into iOS and all the details necessary to get your first OpenGL ES apps up and running on an iOS device.

If you're familiar with the basics of using OpenGL ES in iOS, read "Drawing to Other Rendering Destinations" (page 32) and "Multitasking, High Resolution, and Other iOS Features" (page 43) for important platform-specific guidelines. Developers familiar with using OpenGL ES in iOS versions before 5.0 should study "Drawing with OpenGL ES and GLKit" (page 23) for details on new features for streamlining OpenGL ES development.

Finally, read "OpenGL ES Design Guidelines" (page 48), "Tuning Your OpenGL ES App" (page 62), and the following chapters to dig deeper into how to design efficient OpenGL ES apps.

Unless otherwise noted, OpenGL ES code examples in this book target OpenGL ES 3.0. You may need to make changes to use these code examples with other OpenGL ES versions.

Prerequisites

Before attempting use OpenGL ES, you should already be familiar with general iOS app architecture. See *Start Developing iOS Apps Today*.

This document is not a complete tutorial or a reference for the cross-platform OpenGL ES API. To learn more about OpenGL ES, consult the references below.

See Also

OpenGL ES is an open standard defined by the Khronos Group. For more information about the OpenGL ES standard, please consult their web page at http://www.khronos.org/opengles/.

- *OpenGL® ES 2.0 Programming Guide*, published by Addison-Wesley, provides a comprehensive introduction to OpenGL ES concepts.

- *OpenGL® Shading Language, Third Edition*, also published by Addison-Wesley, provides many shading algorithms useable in your OpenGL ES app. You may need to modify some of these algorithms to run efficiently on mobile graphics processors.
- OpenGL ES API Registry is the official repository for the OpenGL ES specifications, the OpenGL ES shading language specifications, and documentation for OpenGL ES extensions.
- *OpenGL ES Framework Reference* describes the platform-specific functions and classes provided by Apple to integrate OpenGL ES into iOS.
- *iOS Device Compatibility Reference* provides more detailed information on the hardware and software features available to your app.
- *GLKit Framework Reference* describes a framework provided by Apple to make it easier to develop OpenGL ES 2.0 and 3.0 apps.

Checklist for Building OpenGL ES Apps for iOS

The OpenGL ES specification defines a platform-neutral API for using GPU hardware to render graphics. Platforms implementing OpenGL ES provide a rendering context for executing OpenGL ES commands, framebuffers to hold rendering results, and one or more rendering destinations that present the contents of a framebuffer for display. In iOS, the `EAGLContext` class implements a rendering context. iOS provides only one type of framebuffer, the OpenGL ES framebuffer object, and the `GLKView` and `CAEAGLLayer` classes implement rendering destinations.

Building an OpenGL ES app in iOS requires several considerations, some of which are generic to OpenGL ES programming and some of which are specific to iOS. Follow this checklist and the detailed sections below to get started:

1. Determine which version(s) of OpenGL ES have the right feature set for your app, and create an OpenGL ES context.
2. Verify at runtime that the device supports the OpenGL ES capabilities you want to use.
3. Choose where to render your OpenGL ES content.
4. Make sure your app runs correctly in iOS.
5. Implement your rendering engine.
6. Use Xcode and Instruments to debug your OpenGL ES app and tune it for optimal performance .

Choosing Which OpenGL ES Versions to Support

Decide whether your app should support OpenGL ES 3.0, OpenGL ES 2.0, OpenGL ES 1.1, or multiple versions.

- OpenGL ES 3.0 is new in iOS 7. It adds a number of new features that enable higher performance, general-purpose GPU computing techniques, and more complex visual effects previously only possible on desktop-class hardware and game consoles.
- OpenGL ES 2.0 is the baseline profile for iOS devices, featuring a configurable graphics pipeline based on programmable shaders.
- OpenGL ES 1.1 provides only a basic fixed-function graphics pipeline and is available in iOS primarily for backward compatibility.

You should target the version or versions of OpenGL ES that support the features and devices most relevant to your app. To learn more about the OpenGL ES capabilities of iOS devices, read *iOS Device Compatibility Reference*.

To create contexts for the versions of OpenGL ES you plan to support, read "Configuring OpenGL ES Contexts" (page 19). To learn how your choice of OpenGL ES version relates to the rendering algorithms you might use in your app, read "OpenGL ES Versions and Renderer Architecture" (page 50).

Verifying OpenGL ES Capabilities

The *iOS Device Compatibility Reference* summarizes the capabilities and extensions available on shipping iOS devices. However, to allow your app to run on as many devices and iOS versions as possible, your app should always query the OpenGL ES implementation for its capabilities at runtime.

To determine implementation specific limits such as the maximum texture size or maximum number of vertex attributes, look up the value for the corresponding token (such as `MAX_TEXTURE_SIZE` or `MAX_VERTEX_ATTRIBS`, as found in the `gl.h` header) using the appropriate `glGet` function for its data type.

To check for OpenGL ES 3.0 extensions, use the `glGetIntegerv` and `glGetStringi` functions as in the following code example:

```
BOOL CheckForExtension(NSString *searchName)
{
    // Create a set containing all extension names.
    // (For better performance, create the set only once and cache it for future
use.)
    int max = 0;
    glGetIntegerv(GL_NUM_EXTENSIONS, &max);
    NSMutableSet *extensions = [NSMutableSet set];
    for (int i = 0; i < max; i++) {
        [extensions addObject: @( (char *)glGetStringi(GL_EXTENSIONS, i) )];
    }
    return [extensions containsObject: searchName];
}
```

To check for OpenGL ES 1.1 and 2.0 extensions, call `glGetString(GL_EXTENSIONS)` to get a space-delimited list of all extension names.

Choosing a Rendering Destination

In iOS, a framebuffer object stores the results of drawing commands. (iOS does not implement window-system-provided framebuffers.) You can use the contents of a framebuffer object in multiple ways:

- The GLKit framework provides a view that draws OpenGL ES content and manages its own framebuffer object, and a view controller that supports animating OpenGL ES content. Use these classes to create full screen views or to fit your OpenGL ES content into a UIKit view hierarchy. To learn about these classes, read "Drawing with OpenGL ES and GLKit" (page 23).
- The `CAEAGLLayer` class provides a way to draw OpenGL ES content as part of a Core Animation layer composition. You must create your own framebuffer object when using this class.
- As with any OpenGL ES implementation, you can also use framebuffers for offscreen graphics processing or rendering to a texture for use elsewhere in the graphics pipeline. With OpenGL ES 3.0, offscreen buffers can be used in rendering algorithms that utilize multiple render targets.

To learn about rendering to an offscreen buffer, a texture, or a Core Animation layer, read "Drawing to Other Rendering Destinations" (page 32).

Integrating with iOS

iOS apps support multitasking by default, but handling this feature correctly in an OpenGL ES app requires additional consideration. Improper use of OpenGL ES can result in your app being killed by the system when in the background.

Many iOS devices include high-resolution displays, so your app should support multiple display sizes and resolutions.

To learn about supporting these and other iOS features, read "Multitasking, High Resolution, and Other iOS Features" (page 43).

Implementing a Rendering Engine

There are many possible strategies for designing your OpenGL ES drawing code, the full details of which are beyond the scope of this document. Many aspects of rendering engine design are generic to all implementations of OpenGL and OpenGL ES.

To learn about design considerations important for iOS devices, read "OpenGL ES Design Guidelines" (page 48) and "Concurrency and OpenGL ES" (page 106).

Debugging and Profiling

Xcode and Instruments provide a number of tools for tracking down rendering problems and analyzing OpenGL ES performance in your app.

To learn more about solving problems and improving performance in your OpenGL ES app, read "Tuning Your OpenGL ES App" (page 62).

Configuring OpenGL ES Contexts

Every implementation of OpenGL ES provides a way to create rendering contexts to manage the state required by the OpenGL ES specification. By placing this state in a context, multiple apps can easily share the graphics hardware without interfering with the other's state.

This chapter details how to create and configure contexts on iOS.

EAGL Is the iOS Implementation of an OpenGL ES Rendering Context

Before your app can call any OpenGL ES functions, it must initialize an `EAGLContext` object. The `EAGLContext` class also provides methods used to integrate OpenGL ES content with Core Animation.

The Current Context Is the Target for OpenGL ES Function Calls

Every thread in an iOS app has a **current context**; when you call an OpenGL ES function, this is the context whose state is changed by the call. The thread maintains a strong reference to the `EAGLContext` object.

To set a thread's current context, call the `EAGLContext` class method `setCurrentContext:` when executing on that thread.

```
[EAGLContext setCurrentContext: myContext];
```

Call the `EAGLContext` class method `currentContext` to retrieve a thread's current context.

Note: If your app actively switches between two or more contexts on the same thread, call the `glFlush` function before setting a new context as the current context. This ensures that previously submitted commands are delivered to the graphics hardware in a timely fashion.

Every Context Targets a Specific Version of OpenGL ES

An `EAGLContext` object supports only one version of OpenGL ES. For example, code written for OpenGL ES 1.1 is not compatible with an OpenGL ES 2.0 or 3.0 context. Code using core OpenGL ES 2.0 features is compatible with a OpenGL ES 3.0 context, and code designed for OpenGL ES 2.0 extensions can often be used in an OpenGL ES 3.0 context with minor changes. Many new OpenGL ES 3.0 features and increased hardware capabilities require an OpenGL ES 3.0 context.

Your app decides which version of OpenGL ES to support when it creates and initializes the `EAGLContext` object. If the device does not support the requested version of OpenGL ES, the `initWithAPI:` method returns `nil`. Your app must test to ensure that a context was initialized successfully before using it.

To support multiple versions of OpenGL ES as rendering options in your app, you should first attempt to initialize a rendering context of the newest version you want to target. If the returned object is `nil`, initialize a context of an older version instead. Listing 2-1 demonstrates how to do this.

Listing 2-1 Supporting multiple versions of OpenGL ES in the same app

```
EAGLContext* CreateBestEAGLContext()
{
   EAGLContext *context = [[EAGLContext alloc]
initWithAPI:kEAGLRenderingAPIOpenGLES3];
   if (context == nil) {
      context = [[EAGLContext alloc] initWithAPI:kEAGLRenderingAPIOpenGLES2];
   }
   return context;
}
```

A context's `API` property states which version of OpenGL ES the context supports. Your app would test the context's `API` property and use it to choose the correct rendering path. A common pattern for implementing this is to create a class for each rendering path; your app tests the context and creates a renderer once, on initialization.

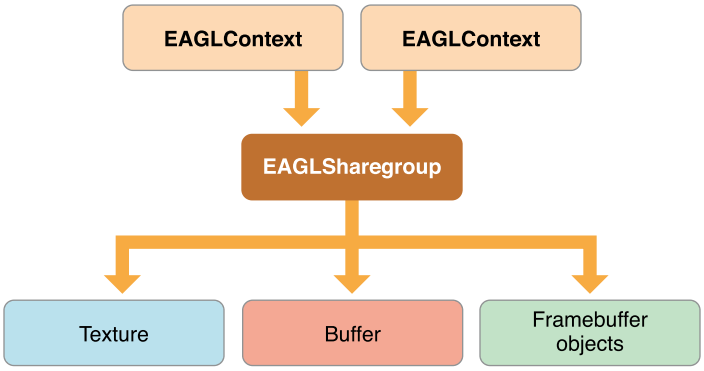
An EAGL Sharegroup Manages OpenGL ES Objects for the Context

Although the context holds the OpenGL ES state, it does not directly manage OpenGL ES objects. Instead, OpenGL ES objects are created and maintained by an `EAGLSharegroup` object. Every context contains an `EAGLSharegroup` object that it delegates object creation to.

The advantage of a sharegroup becomes obvious when two or more contexts refer to the same sharegroup, as shown in Figure 2-1. When multiple contexts are connected to a common sharegroup, OpenGL ES objects created by any context are available on all contexts; if you bind to the same object identifier on another context than the one that created it, you reference the *same* OpenGL ES object. Resources are often scarce on mobile devices; creating multiple copies of the same content on multiple contexts is wasteful. Sharing common resources makes better use of the available graphics resources on the device.

A sharegroup is an opaque object; it has no methods or properties that your app can call. Contexts that use the sharegroup object keep a strong reference to it.

Figure 2-1 Two contexts sharing OpenGL ES objects



Sharegroups are most useful under two specific scenarios:

- When most of the resources shared between the contexts are unchanging.
- When you want your app to be able to create new OpenGL ES objects on a thread other than the main thread for the renderer. In this case, a second context runs on a separate thread and is devoted to fetching data and creating resources. After the resource is loaded, the first context can bind to the object and use it immediately. The `GLKTextureLoader` class uses this pattern to provide asynchronous texture loading.

To create multiple contexts that reference the same sharegroup, the first context is initialized by calling `initWithAPI:`; a sharegroup is automatically created for the context. The second and later contexts are initialized to use the first context's sharegroup by calling the `initWithAPI:sharegroup:` method instead. Listing 2-2 shows how this would work. The first context is created using the convenience function defined in Listing 2-1 (page 20). The second context is created by extracting the API version and sharegroup from the first context.

Important: All contexts associated with the same sharegroup must use the same version of the OpenGL ES API as the initial context.

Listing 2-2 Creating two contexts with a common sharegroup

```
EAGLContext* firstContext = CreateBestEAGLContext();

EAGLContext* secondContext = [[EAGLContext alloc] initWithAPI:[firstContext API]
sharegroup: [firstContext sharegroup]];
```

It is your app's responsibility to manage state changes to OpenGL ES objects when the sharegroup is shared by multiple contexts. Here are the rules:

- Your app may access the object across multiple contexts simultaneously provided the object is not being modified.
- While the object is being modified by commands sent to a context, the object must not be read or modified on any other context.
- After an object has been modified, all contexts must rebind the object to see the changes. The contents of the object are undefined if a context references it before binding it.

Here are the steps your app should follow to update an OpenGL ES object:

1. Call `glFlush` on every context that may be using the object.
2. On the context that wants to modify the object, call one or more OpenGL ES functions to change the object.
3. Call `glFlush` on the context that received the state-modifying commands.
4. On every other context, rebind the object identifier.

Note: Another way to share objects is to use a single rendering context, but multiple destination framebuffers. At rendering time, your app binds the appropriate framebuffer and renders its frames as needed. Because all of the OpenGL ES objects are referenced from a single context, they see the same OpenGL ES data. This pattern uses less resources, but is only useful for single-threaded apps where you can carefully control the state of the context.

Drawing with OpenGL ES and GLKit

The GLKit framework provides view and view controller classes that eliminate the setup and maintenance code that would otherwise be required for drawing and animating OpenGL ES content. The `GLKView` class manages OpenGL ES infrastructure to provide a place for your drawing code, and the `GLKViewController` class provides a rendering loop for smooth animation of OpenGL ES content in a GLKit view. These classes extend the standard UIKit design patterns for drawing view content and managing view presentation. As a result, you can focus your efforts primarily on your OpenGL ES rendering code and get your app up and running quickly. The GLKit framework also provides other features to ease OpenGL ES 2.0 and 3.0 development.

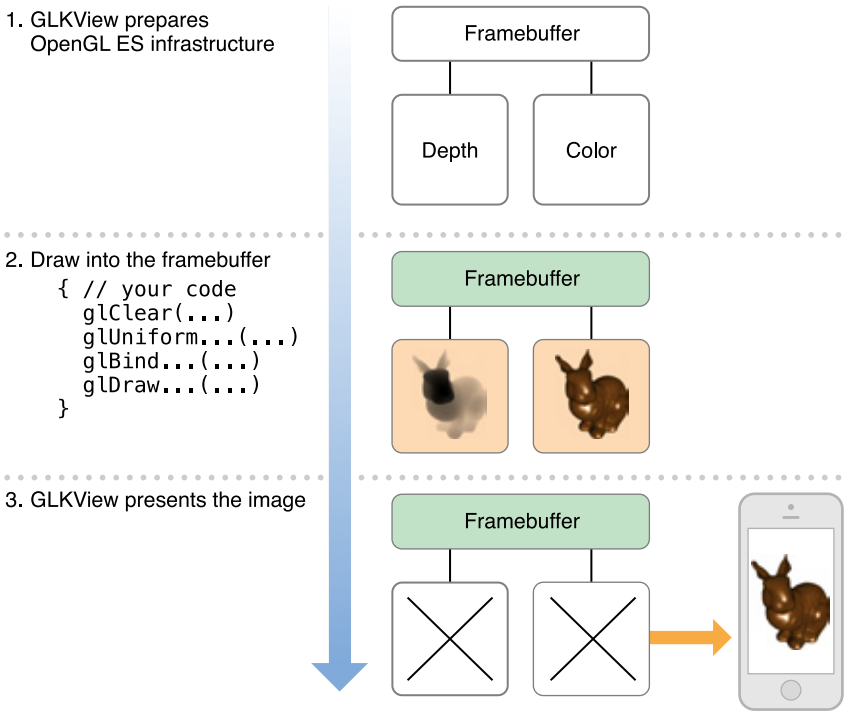
A GLKit View Draws OpenGL ES Content on Demand

The `GLKView` class provides an OpenGL ES–based equivalent of the standard `UIView` drawing cycle. A `UIView` instance automatically configures its graphics context so that your `drawRect:` implementation need only perform Quartz 2D drawing commands, and a `GLKView` instance automatically configures itself so that your drawing method need only perform OpenGL ES drawing commands. The `GLKView` class provides this functionality by maintaining a framebuffer object that holds the results of your OpenGL ES drawing commands, and then automatically presents them to Core Animation once your drawing method returns.

Like a standard UIKit view, a GLKit view renders its content on demand. When your view is first displayed, it calls your drawing method—Core Animation caches the rendered output and displays it whenever your view is shown. When you want to change the contents of your view, call its `setNeedsDisplay` method and the view again calls your drawing method, caches the resulting image, and presents it on screen. This approach

is useful when the data used to render an image changes infrequently or only in response to user action. By rendering new view contents only when you need to, you conserve battery power on the device and leave more time for the device to perform other actions.

Figure 3-1 Rendering OpenGL ES content with a GLKit view



Creating and Configuring a GLKit View

You can create and configure a `GLKView` object either programmatically or using Interface Builder. Before you can use it for drawing, you must associate it with an `EAGLContext` object (see "Configuring OpenGL ES Contexts" (page 19)).

- When creating a view programmatically, first create a context and then pass it to the view's `initWithFrame:context:` method.
- After loading a view from a storyboard, create a context and set it as the value of the view's `context` property.

A GLKit view automatically creates and configures its own OpenGL ES framebuffer object and renderbuffers. You control the attributes of these objects using the view's drawable properties, as illustrated in Listing 3-1. If you change the size, scale factor, or drawable properties of a GLKit view, it automatically deletes and re-creates the appropriate framebuffer objects and renderbuffers the next time its contents are drawn.

Listing 3-1 Configuring a GLKit view

```
- (void)viewDidLoad
{
    [super viewDidLoad];

   // Create an OpenGL ES context and assign it to the view loaded from storyboard
    GLKView *view = (GLKView *)self.view;
    view.context = [[EAGLContext alloc] initWithAPI:kEAGLRenderingAPIOpenGLES2];

    // Configure renderbuffers created by the view
    view.drawableColorFormat = GLKViewDrawableColorFormatRGBA8888;
    view.drawableDepthFormat = GLKViewDrawableDepthFormat24;
    view.drawableStencilFormat = GLKViewDrawableStencilFormat8;

    // Enable multisampling
    view.drawableMultisample = GLKViewDrawableMultisample4X;
}
```

You can enable multisampling for a `GLKView` instance using its `drawableMultisample` property. Multisampling is a form of **antialiasing** that smooths jagged edges, improving image quality in most 3D apps at the cost of using more memory and fragment processing time—if you enable multisampling, always test your app's performance to ensure that it remains acceptable.

Drawing With a GLKit View

Figure 3-1 (page 24) outlines the three steps for drawing OpenGL ES content: preparing OpenGL ES infrastructure, issuing drawing commands, and presenting the rendered content to Core Animation for display. The `GLKView` class implements the first and third steps. For the second step, you implement a drawing method like the example in Listing 3-2.

Listing 3-2 Example drawing method for a GLKit view

```
- (void)drawRect:(CGRect)rect
{
    // Clear the framebuffer
    glClearColor(0.0f, 0.0f, 0.1f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
    // Draw using previously configured texture, shader, uniforms, and vertex array
    glBindTexture(GL_TEXTURE_2D, _planetTexture);
    glUseProgram(_diffuseShading);
    glUniformMatrix4fv(_uniformModelViewProjectionMatrix, 1, 0,
_modelViewProjectionMatrix.m);
    glBindVertexArrayOES(_planetMesh);
    glDrawElements(GL_TRIANGLE_STRIP, 256, GL_UNSIGNED_SHORT);
}
```

> **Note:** The `glClear` function hints to OpenGL ES that any existing framebuffer contents can be discarded, avoiding costly memory operations to load the previous contents into memory. To ensure optimal performance, you should always call this function before drawing.

The `GLKView` class is able to provide a simple interface for OpenGL ES drawing because it manages the standard parts of the OpenGL ES rendering process:

- Before invoking your drawing method, the view:
  - Makes its `EAGLContext` object the current context
  - Creates a framebuffer object and renderbuffers based on its current size, scale factor, and drawable properties (if needed)
  - Binds the framebuffer object as the current destination for drawing commands
  - Sets the OpenGL ES viewport to match the framebuffer size
- After your drawing method returns, the view:
  - Resolves multisampling buffers (if multisampling is enabled)
  - Discards renderbuffers whose contents are no longer needed
  - Presents renderbuffer contents to Core Animation for caching and display

Rendering Using a Delegate Object

Many OpenGL ES apps implement rendering code in a custom class. An advantage of this approach is that it allows you to easily support multiple rendering algorithms by defining a different renderer class for each. Rendering algorithms that share common functionality can inherit it from a superclass. For example, you might

use different renderer classes to support both OpenGL ES 2.0 and 3.0 (see "Configuring OpenGL ES Contexts" (page 19)). Or you might use them to customize rendering for better image quality on devices with more powerful hardware.

GLKit is well suited to this approach—you can make your renderer object the delegate of a standard `GLKView` instance. Instead of subclassing `GLKView` and implementing the `drawRect:` method, your renderer class adopts the `GLKViewDelegate` protocol and implements the `glkView:drawInRect:` method. Listing 3-3 demonstrates choosing a renderer class based on hardware features at app launch time.

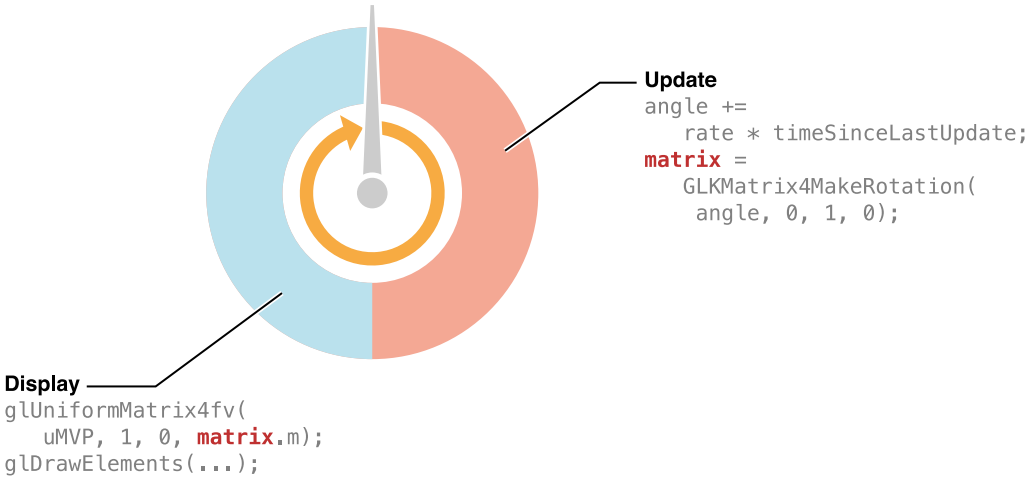
Listing 3-3 Choosing a renderer class based on hardware features

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Create a context so we can test for features
    EAGLContext *context = [[EAGLContext alloc]
initWithAPI:kEAGLRenderingAPIOpenGLES2];
    [EAGLContext setCurrentContext:context];

    // Choose a rendering class based on device features
    GLint maxTextureSize;
    glGetIntegerv(GL_MAX_TEXTURE_SIZE, &maxTextureSize);
    if (maxTextureSize > 2048)
        self.renderer = [[MyBigTextureRenderer alloc] initWithContext:context];
    else
        self.renderer = [[MyRenderer alloc] initWithContext:context];

    // Make the renderer the delegate for the view loaded from the main storyboard
    GLKView *view = (GLKView *)self.window.rootViewController.view;
    view.delegate = self.renderer;

    // Give the OpenGL ES context to the view so it can draw
    view.context = context;

    return YES;
}
```

A GLKit View Controller Animates OpenGL ES Content

By default, a `GLKView` object renders its contents on demand. That said, a key advantage to drawing with OpenGL ES is its ability to use graphics processing hardware for continuous animation of complex scenes—apps such as games and simulations rarely present static images. For these cases, the GLKit framework provides a view controller class that maintains an animation loop for the `GLKView` object it manages. This loop follows a design pattern common in games and simulations, with two phases: **update** and **display**. Figure 3-2 shows a simplified example of an animation loop.

Figure 3-2 The animation loop



Understanding the Animation Loop

For the update phase, the view controller calls its own `update` method (or its delegate's `glkViewControllerUpdate:` method). In this method, you should prepare for drawing the next frame. For example, a game might use this method to determine the positions of player and enemy characters based on input events received since the last frame, and a scientific visualization might use this method to run a step of its simulation. If you need timing information to determine your app's state for the next frame, use one of the view controller's timing properties such as the `timeSinceLastUpdate` property. In Figure 3-2, the update phase increments an `angle` variable and uses it to calculate a transformation matrix.

For the display phase, the view controller calls its view's `display` method, which in turn calls your drawing method. In your drawing method, you submit OpenGL ES drawing commands to the GPU to render your content. For optimal performance, your app should modify OpenGL ES objects at the start of rendering a new frame, and submit drawing commands afterward. In Figure 3-2, the display phase sets a uniform variable in a shader program to the matrix calculated in the update phase, and then submits a drawing command to render new content.

The animation loop alternates between these two phases at the rate indicated by the view controller's `framesPerSecond` property. You can use the `preferredFramesPerSecond` property to set a desired frame rate—to optimize performance for the current display hardware, the view controller automatically chooses an optimal frame rate close to your preferred value.

> **Important:** For best results, choose a frame rate your app can consistently achieve. A smooth, consistent frame rate produces a more pleasant user experience than a frame rate that varies erratically.

Using a GLKit View Controller

Listing 3-4 demonstrates a typical strategy for rendering animated OpenGL ES content using a `GLKViewController` subclass and `GLKView` instance.

Listing 3-4 Using a GLKit view and view controller to draw and animate OpenGL ES content

```
@implementation PlanetViewController // subclass of GLKViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    // Create an OpenGL ES context and assign it to the view loaded from storyboard
    GLKView *view = (GLKView *)self.view;
    view.context = [[EAGLContext alloc] initWithAPI:kEAGLRenderingAPIOpenGLES2];

    // Set animation frame rate
    self.preferredFramesPerSecond = 60;

    // Not shown: load shaders, textures and vertex arrays, set up projection
matrix
    [self setupGL];
}

- (void)update
{
    _rotation += self.timeSinceLastUpdate * M_PI_2; // one quarter rotation per
second
```

```
    // Set up transform matrices for the rotating planet
    GLKMatrix4 modelViewMatrix = GLKMatrix4MakeRotation(_rotation, 0.0f, 1.0f,
0.0f);
    _normalMatrix =
GLKMatrix3InvertAndTranspose(GLKMatrix4GetMatrix3(modelViewMatrix), NULL);
    _modelViewProjectionMatrix = GLKMatrix4Multiply(_projectionMatrix,
modelViewMatrix);
}

- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect
{
    // Clear the framebuffer
    glClearColor(0.0f, 0.0f, 0.1f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Set shader uniforms to values calculated in -update
    glUseProgram(_diffuseShading);
    glUniformMatrix4fv(_uniformModelViewProjectionMatrix, 1, 0,
_modelViewProjectionMatrix.m);
    glUniformMatrix3fv(_uniformNormalMatrix, 1, 0, _normalMatrix.m);

    // Draw using previously configured texture and vertex array
    glBindTexture(GL_TEXTURE_2D, _planetTexture);
    glBindVertexArrayOES(_planetMesh);
    glDrawElements(GL_TRIANGLE_STRIP, 256, GL_UNSIGNED_SHORT, 0);
}

@end
```

In this example, an instance of the `PlanetViewController` class (a custom `GLKViewController` subclass) is loaded from a storyboard, along with a standard `GLKView` instance and its drawable properties. The `viewDidLoad` method creates an OpenGL ES context and provides it to the view, and also sets the frame rate for the animation loop.

The view controller is automatically the delegate of its view, so it implements both the update and display phases of the animation loop. In the `update` method, it calculates the transformation matrices needed to display a rotating planet. In the `glkView:drawInRect:` method, it provides those matrices to a shader program and submits drawing commands to render the planet geometry.

Using GLKit to Develop Your Renderer

In addition to view and view controller infrastructure, the GLKit framework provides several other features to ease OpenGL ES development on iOS.

Handling Vector and Matrix Math

OpenGL ES 2.0 and later doesn't provide built-in functions for creating or specifying transformation matrices. Instead, programmable shaders provide vertex transformation, and you specify shader inputs using generic uniform variables. The GLKit framework includes a comprehensive library of vector and matrix types and functions, optimized for high performance on iOS hardware. (See *GLKit Framework Reference*.)

Migrating from the OpenGL ES 1.1 Fixed-Function Pipeline

OpenGL ES 2.0 and later removes all functionality associated with the OpenGL ES 1.1 fixed-function graphics pipeline. The `GLKBaseEffect` class provides an Objective-C analog to the transformation, lighting and shading stages of the OpenGL ES 1.1 pipeline, and the `GLKSkyboxEffect` and `GLKReflectionMapEffect` classes add support for common visual effects. See the reference documentation for these classes for details.

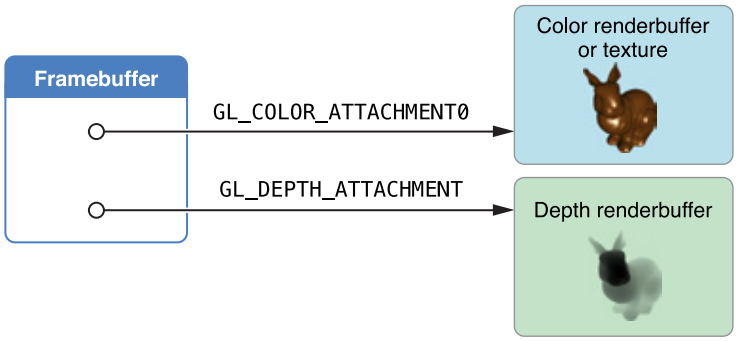
Loading Texture Data

The `GLKTextureLoader` class provides a simple way to load texture data from any image format supported by iOS into an OpenGL ES context, synchronously or asynchronously. (See "Use the GLKit Framework to Load Texture Data" (page 89).)

Drawing to Other Rendering Destinations

Framebuffer objects are the destination for rendering commands. When you create a framebuffer object, you have precise control over its storage for color, depth, and stencil data. You provide this storage by attaching images to the framebuffer, as shown in Figure 4-1. The most common image attachment is a renderbuffer object. You can also attach an OpenGL ES texture to the color attachment point of a framebuffer, which means that any drawing commands are rendered into the texture. Later, the texture can act as an input to future rendering commands. You can also create multiple framebuffer objects in an single rendering context. You might do this so that you can share the same rendering pipeline and OpenGL ES resources between multiple framebuffers.

Figure 4-1 Framebuffer with color and depth renderbuffers



All of these approaches require manually creating framebuffer and renderbuffer objects to store the rendering results from your OpenGL ES context, as well as writing additional code to present their contents to the screen and (if needed) run an animation loop.

Creating a Framebuffer Object

Depending on what task your app intends to perform, your app configures different objects to attach to the framebuffer object. In most cases, the difference in configuring the framebuffer is in what object is attached to the framebuffer object's color attachment point:

- To use the framebuffer for offscreen image processing, attach a renderbuffer. See "Creating Offscreen Framebuffer Objects" (page 33).

- To use the framebuffer image as an input to a later rendering step, attach a texture. See "Using Framebuffer Objects to Render to a Texture" (page 34).
- To use the framebuffer in a Core Animation layer composition, use a special Core Animation–aware renderbuffer. See "Rendering to a Core Animation Layer" (page 35).

Creating Offscreen Framebuffer Objects

A framebuffer intended for offscreen rendering allocates all of its attachments as OpenGL ES renderbuffers. The following code allocates a framebuffer object with color and depth attachments.

1. Create the framebuffer and bind it.

```
GLuint framebuffer;
glGenFramebuffers(1, &framebuffer);
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
```

2. Create a color renderbuffer, allocate storage for it, and attach it to the framebuffer's color attachment point.

```
GLuint colorRenderbuffer;
glGenRenderbuffers(1, &colorRenderbuffer);
glBindRenderbuffer(GL_RENDERBUFFER, colorRenderbuffer);
glRenderbufferStorage(GL_RENDERBUFFER, GL_RGBA8, width, height);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
GL_RENDERBUFFER, colorRenderbuffer);
```

3. Create a depth or depth/stencil renderbuffer, allocate storage for it, and attach it to the framebuffer's depth attachment point.

```
GLuint depthRenderbuffer;
glGenRenderbuffers(1, &depthRenderbuffer);
glBindRenderbuffer(GL_RENDERBUFFER, depthRenderbuffer);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT16, width,
height);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
GL_RENDERBUFFER, depthRenderbuffer);
```

4. Test the framebuffer for completeness. This test only needs to be performed when the framebuffer's configuration changes.

```
GLenum status = glCheckFramebufferStatus(GL_FRAMEBUFFER) ;
if(status != GL_FRAMEBUFFER_COMPLETE) {
    NSLog(@"failed to make complete framebuffer object %x", status);
}
```

After drawing to an offscreen renderbuffer, you can return its contents to the CPU for further processing using the `glReadPixels` function.

Using Framebuffer Objects to Render to a Texture

The code to create this framebuffer is almost identical to the offscreen example, but now a texture is allocated and attached to the color attachment point.

1. Create the framebuffer object (using the same procedure as in "Creating Offscreen Framebuffer Objects" (page 33)).
2. Create the destination texture, and attach it to the framebuffer's color attachment point.

```
// create the texture
GLuint texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8,  width, height, 0, GL_RGBA,
GL_UNSIGNED_BYTE, NULL);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
 texture, 0);
```

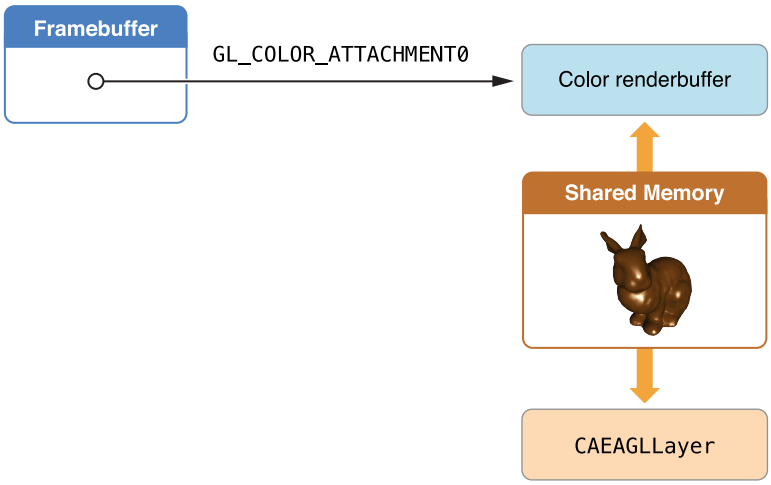
3. Allocate and attach a depth buffer (as before).
4. Test the framebuffer for completeness (as before).

Although this example assumes you are rendering to a color texture, other options are possible. For example, using the `OES_depth_texture` extension, you can attach a texture to the depth attachment point to store depth information from the scene into a texture. You might use this depth information to calculate shadows in the final rendered scene.

Rendering to a Core Animation Layer

Core Animation is the central infrastructure for graphics rendering and animation on iOS. You can compose your app's user interface or other visual displays using layers that host content rendered using different iOS subsystems, such as UIKit, Quartz 2D, and OpenGL ES. OpenGL ES connects to Core Animation through the `CAEAGLLayer` class, a special type of Core Animation layer whose contents come from an OpenGL ES renderbuffer. Core Animation composites the renderbuffer's contents with other layers and displays the resulting image on screen.

Figure 4-2 Core Animation shares the renderbuffer with OpenGL ES



The `CAEAGLLayer` provides this support to OpenGL ES by providing two key pieces of functionality. First, it allocates shared storage for a renderbuffer. Second, it presents the renderbuffer to Core Animation, replacing the layer's previous contents with data from the renderbuffer. An advantage of this model is that the contents of the Core Animation layer do not need to be drawn in every frame, only when the rendered image changes.

> **Note:** The `GLKView` class automates the steps below, so you should use it when you want to draw with OpenGL ES in the content layer of a view.

To use a Core Animation layer for OpenGL ES rendering:

1. Create a `CAEAGLLayer` object and configure its properties.

   For optimal performance, set the value of the layer's `opaque` property to `YES`. See "Be Aware of Core Animation Compositing Performance" (page 72).

   Optionally, configure the surface properties of the rendering surface by assigning a new dictionary of values to the `drawableProperties` property of the `CAEAGLLayer` object. You can specify the pixel format for the renderbuffer and specify whether the renderbuffer's contents are discarded after they are sent to Core Animation. For a list of the permitted keys, see *EAGLDrawable Protocol Reference*.

2. Allocate an OpenGL ES context and make it the current context. See "Configuring OpenGL ES Contexts" (page 19).
3. Create the framebuffer object (as in "Creating Offscreen Framebuffer Objects" (page 33) above).
4. Create a color renderbuffer, allocating its storage by calling the context's `renderbufferStorage:fromDrawable:` method and passing the layer object as the parameter. The width, height and pixel format are taken from the layer and used to allocate storage for the renderbuffer.

```
GLuint colorRenderbuffer;
glGenRenderbuffers(1, &colorRenderbuffer);
glBindRenderbuffer(GL_RENDERBUFFER, colorRenderbuffer);
[myContext renderbufferStorage:GL_RENDERBUFFER fromDrawable:myEAGLLayer];
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
GL_RENDERBUFFER, colorRenderbuffer);
```

> **Note:** When the Core Animation layer's bounds or properties change, your app should reallocate the renderbuffer's storage. If you do not reallocate the renderbuffers, the renderbuffer size won't match the size of the layer; in this case, Core Animation may scale the image's contents to fit in the layer.

5. Retrieve the height and width of the color renderbuffer.

```
GLint width;
GLint height;
glGetRenderbufferParameteriv(GL_RENDERBUFFER, GL_RENDERBUFFER_WIDTH,
&width);
glGetRenderbufferParameteriv(GL_RENDERBUFFER, GL_RENDERBUFFER_HEIGHT,
&height);
```

   In earlier examples, the width and height of the renderbuffers were explicitly provided to allocate storage for the buffer. Here, the code retrieves the width and height from the color renderbuffer after its storage is allocated. Your app does this because the actual dimensions of the color renderbuffer are calculated based on the layer's bounds and scale factor. Other renderbuffers attached to the framebuffer must have the same dimensions. In addition to using the height and width to allocate the depth buffer, use them to assign the OpenGL ES viewport and to help determine the level of detail required in your app's textures and models. See "Supporting High-Resolution Displays" (page 45).
6. Allocate and attach a depth buffer (as before).

7. Test the framebuffer for completeness (as before).
8. Add the `CAEAGLLayer` object to your Core Animation layer hierarchy by passing it to the `addSublayer:` method of a visible layer.

Drawing to a Framebuffer Object

Now that you have a framebuffer object, you need to fill it. This section describes the steps required to render new frames and present them to the user. Rendering to a texture or offscreen framebuffer acts similarly, differing only in how your app uses the final frame.

Rendering on Demand or with an Animation Loop

You must choose when to draw your OpenGL ES content when rendering to a Core Animation layer, just as when drawing with GLKit views and view controllers. If rendering to an offscreen framebuffer or texture, draw whenever is appropriate to the situations where you use those types of framebuffers.

For on-demand drawing, implement your own method to draw into and present your renderbuffer, and call it whenever you want to display new content.

To draw with an animation loop, use a `CADisplayLink` object. A display link is a kind of timer provided by Core Animation that lets you synchronize drawing to the refresh rate of a screen. Listing 4-1 (page 37) shows how you can retrieve the screen showing a view, use that screen to create a new display link object and add the display link object to the run loop.

> **Note:** The `GLKViewController` class automates the usage of `CADisplayLink` objects for animating `GLKView` content. Use the `CADisplayLink` class directly only if you need behavior beyond what the GLKit framework provides.

Listing 4-1 Creating and starting a display link

```
displayLink = [myView.window.screen displayLinkWithTarget:self
selector:@selector(drawFrame)];
[displayLink addToRunLoop:[NSRunLoop currentRunLoop] forMode:NSDefaultRunLoopMode];
```

Inside your implementation of the `drawFrame` method, read the display link's `timestamp` property to get the timestamp for the next frame to be rendered. It can use that value to calculate the positions of objects in the next frame.

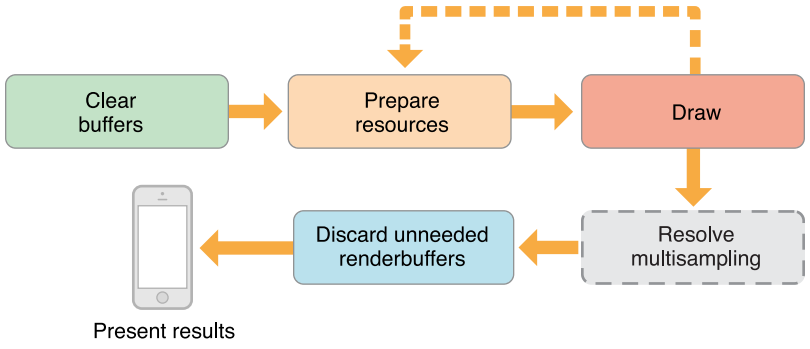
Normally, the display link object is fired every time the screen refreshes; that value is usually 60 Hz, but may vary on different devices. Most apps do not need to update the screen 60 times per second. You can set the display link's `frameInterval` property to the number of actual frames that go by before your method is called. For example, if the frame interval was set to `3`, your app is called every third frame, or roughly 20 frames per second.

> **Important:** For best results, choose a frame rate your app can consistently achieve. A smooth, consistent frame rate produces a more pleasant user experience than a frame rate that varies erratically.

Rendering a Frame

Figure 4-3 (page 38) shows the steps an OpenGL ES app should take on iOS to render and present a frame. These steps include many hints to improve performance in your app.

Figure 4-3 iOS OpenGL Rendering Steps



Clear Buffers

At the start of every frame, erase the contents of all framebuffer attachments whose contents from a previous frames are not needed to draw the next frame. Call the `glClear` function, passing in a bit mask with all of the buffers to clear, as shown in Listing 4-2.

Listing 4-2 Clear framebuffer attachments

```
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
```

Using `glClear` hints to OpenGL ES that the existing contents of a renderbuffer or texture can be discarded, avoiding costly operations to load the previous contents into memory.

Prepare OpenGL ES Objects

This step and the next step is the heart of your app, where you decide what you want to display to the user. In this step, you prepare all of the OpenGL ES objects — vertex buffer objects, textures and so on — that are needed to render the frame.

Execute Drawing Commands

This step takes the objects you prepared in the previous step and submits drawing commands to use them. Designing this portion of your rendering code to run efficiently is covered in detail in "OpenGL ES Design Guidelines" (page 48). For now, the most important performance optimization to note is that your app runs faster if it only modifies OpenGL ES objects at the start of rendering a new frame. Although your app can alternate between modifying objects and submitting drawing commands (as shown by the dotted line), it runs faster if it only performs each step once.

Resolve Multisampling

If your app uses multisampling to improve image quality, your app must resolve the pixels before they are presented to the user. Multisampling is covered in detail in "Using Multisampling to Improve Image Quality" (page 40).

Discard Unneeded Renderbuffers

A *discard* operation is a performance hint that tells OpenGL ES that the contents of one or more renderbuffers are no longer needed. By hinting to OpenGL ES that you do not need the contents of a renderbuffer, the data in the buffers can be discarded and expensive tasks to keep the contents of those buffers updated can be avoided.

At this stage in the rendering loop, your app has submitted all of its drawing commands for the frame. While your app needs the color renderbuffer to display to the screen, it probably does not need the depth buffer's contents. Listing 4-3 discards the contents of the depth buffer.

Listing 4-3 Discarding the depth framebuffer

```
const GLenum discards[]  = {GL_DEPTH_ATTACHMENT};
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
glDiscardFramebufferEXT(GL_FRAMEBUFFER,1,discards);
```

> **Note:** The `glDiscardFramebufferEXT` function is provided by the `EXT_discard_framebuffer` extension for OpenGL ES 1.1 and 2.0. In a OpenGL ES 3.0 context, use the `glInvalidateFramebuffer` function instead.

Present the Results to Core Animation

At this step, the color renderbuffer holds the completed frame, so all you need to do is present it to the user. Listing 4-4 (page 40) binds the renderbuffer to the context and presents it. This causes the completed frame to be handed to Core Animation.

Listing 4-4 Presenting the finished frame

```
glBindRenderbuffer(GL_RENDERBUFFER, colorRenderbuffer);
[context presentRenderbuffer:GL_RENDERBUFFER];
```

By default, you must assume that the contents of the renderbuffer are *discarded* after your app presents the renderbuffer. This means that every time your app presents a frame, it must completely re-create the frame's contents when it renders a new frame. The code above always erases the color buffer for this reason.

If your app wants to preserve the contents of the color renderbuffer between frames, add the `kEAGLDrawablePropertyRetainedBacking` key to the dictionary stored in the `drawableProperties` property of the `CAEAGLLayer` object, and remove the `GL_COLOR_BUFFER_BIT` constant from the earlier `glClear` function call. Retained backing may require iOS to allocate additional memory to preserve the buffer's contents, which may reduce your app's performance.

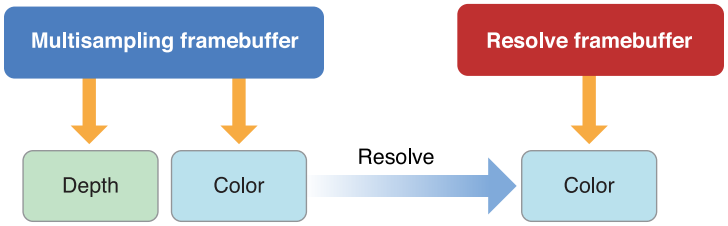
Using Multisampling to Improve Image Quality

Multisampling is a form of **antialiasing** that smooths jagged edges and improves image quality in most 3D apps. OpenGL ES 3.0 includes multisampling as part of the core specification, and iOS provides it in OpenGL ES 1.1 and 2.0 through the `APPLE_framebuffer_multisample` extension. Multisampling uses more memory and fragment processing time to render the image, but it may improve image quality at a lower performance cost than using other approaches.

Figure 4-4 shows how multisampling works. Instead of creating one framebuffer object, your app creates two. The **multisampling buffer** contains all attachments necessary to render your content (typically color and depth buffers). The **resolve buffer** contains only the attachments necessary to display a rendered image to the user (typically a color renderbuffer, but possibly a texture), created using the appropriate procedure from "Creating a Framebuffer Object" (page 32). The multisample renderbuffers are allocated using the same dimensions as

the resolve framebuffer, but each includes an additional parameter that specifies the number of samples to store for each pixel. Your app performs all of its rendering to the multisampling buffer and then generates the final antialiased image by *resolving* those samples into the resolve buffer.

Figure 4-4 How multisampling works



Listing 4-5 shows the code to create the multisampling buffer. This code uses the width and height of the previously created buffer. It calls the `glRenderbufferStorageMultisampleAPPLE` function to create multisampled storage for the renderbuffer.

Listing 4-5 Creating the multisample buffer

```
glGenFramebuffers(1, &sampleFramebuffer);
glBindFramebuffer(GL_FRAMEBUFFER, sampleFramebuffer);


glGenRenderbuffers(1, &sampleColorRenderbuffer);
glBindRenderbuffer(GL_RENDERBUFFER, sampleColorRenderbuffer);
glRenderbufferStorageMultisampleAPPLE(GL_RENDERBUFFER, 4, GL_RGBA8_OES, width,
height);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_RENDERBUFFER,
sampleColorRenderbuffer);


glGenRenderbuffers(1, &sampleDepthRenderbuffer);
glBindRenderbuffer(GL_RENDERBUFFER, sampleDepthRenderbuffer);
glRenderbufferStorageMultisampleAPPLE(GL_RENDERBUFFER, 4, GL_DEPTH_COMPONENT16,
width, height);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER,
sampleDepthRenderbuffer);


if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
    NSLog(@"Failed to make complete framebuffer object %x",
glCheckFramebufferStatus(GL_FRAMEBUFFER));
```

Here are the steps to modify your rendering code to support multisampling:

1. During the Clear Buffers step, you clear the multisampling framebuffer's contents.

```
glBindFramebuffer(GL_FRAMEBUFFER, sampleFramebuffer);
glViewport(0, 0, framebufferWidth, framebufferHeight);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

2. After submitting your drawing commands, you resolve the contents from the multisampling buffer into the resolve buffer. The samples stored for each pixel are combined into a single sample in the resolve buffer.

```
glBindFramebuffer(GL_DRAW_FRAMEBUFFER_APPLE, resolveFrameBuffer);
glBindFramebuffer(GL_READ_FRAMEBUFFER_APPLE, sampleFramebuffer);
glResolveMultisampleFramebufferAPPLE();
```

3. In the Discard step, you can discard both renderbuffers attached to the multisample framebuffer. This is because the contents you plan to present are stored in the resolve framebuffer.

```
const GLenum discards[]  = {GL_COLOR_ATTACHMENT0,GL_DEPTH_ATTACHMENT};
glDiscardFramebufferEXT(GL_READ_FRAMEBUFFER_APPLE,2,discards);
```

4. In the Present Results step, you present the color renderbuffer attached to the resolve framebuffer.

```
glBindRenderbuffer(GL_RENDERBUFFER, colorRenderbuffer);
[context presentRenderbuffer:GL_RENDERBUFFER];
```

Multisampling is not free; additional memory is required to store the additional samples, and resolving the samples into the resolve framebuffer takes time. If you add multisampling to your app, always test your app's performance to ensure that it remains acceptable.

Note: The above code assumes an OpenGL ES 1.1 or 2.0 context. Multisampling is part of the core OpenGL ES 3.0 API, but the functions are different. See the specification for details.

Multitasking, High Resolution, and Other iOS Features

Many aspects of working with OpenGL ES are platform neutral, but some details of working with OpenGL ES on iOS bear special consideration. In particular, an iOS app using OpenGL ES must handle multitasking correctly or risk being terminated when it moves to the background. You should also consider display resolution and other device features when developing OpenGL ES content for iOS devices.

Implementing a Multitasking-Aware OpenGL ES App

Your app can continue to run when a user switches to another app. For an overall discussion of multitasking on iOS, see "App States and Multitasking".

An OpenGL ES app must perform additional work when it is moved into the background. If an app handles these tasks improperly, it may be terminated by iOS instead. Also, an app may want to free OpenGL ES resources so that those resources are made available to the foreground app.

Background Apps May Not Execute Commands on the Graphics Hardware

An OpenGL ES app is terminated if it attempts to execute OpenGL ES commands on the graphics hardware. iOS prevents background apps from accessing the graphics processor so that the frontmost app is always able to present a great experience to the user. Your app can be terminated not only if it makes OpenGL ES calls while in the background but also if previously submitted commands are flushed to the GPU while in the background. Your app must ensure that all previously submitted commands have finished executing before moving into the background.

If you use a GLKit view and view controller, and only submit OpenGL ES commands during your drawing method, your app automatically behaves correctly when it moves to the background. The `GLKViewController` class, by default, pauses its animation timer when your app becomes inactive, ensuring that your drawing method is not called.

If you do not use GLKit views or view controllers or if you submit OpenGL ES commands outside a `GLKView` drawing method, you must take the following steps to ensure that your app is not terminated in the background:

1. In your app delegate's `applicationWillResignActive:` method, your app should stop its animation timer (if any), place itself into a known good state, and then call the `glFinish` function.

2. In your app delegate's `applicationDidEnterBackground:` method, your app may want to delete some of its OpenGL ES objects to make memory and resources available to the foreground app. Call the `glFinish` function to ensure that the resources are removed immediately.
3. After your app exits its `applicationDidEnterBackground:` method, it must not make any new OpenGL ES calls. If it makes an OpenGL ES call, it is terminated by iOS.
4. In your app's `applicationWillEnterForeground:` method, re-create any objects and restart your animation timer.

To summarize, your app needs to call the `glFinish` function to ensure that all previously submitted commands are drained from the command buffer and are executed by OpenGL ES. After it moves into the background, you must avoid all use of OpenGL ES until it moves back into the foreground.

Delete Easily Re-Created Resources Before Moving to the Background

Your app is never required to free up OpenGL ES objects when it moves into the background. Usually, your app should avoid disposing of its content. Consider two scenarios:

- A user is playing your game and exits it briefly to check their calendar. When the player returns to your game, the game's resources are still in memory, and the game can resume immediately.
- Your OpenGL ES app is in the background when the user launches another OpenGL ES app. If that app needs more memory than is available on the device, the system silently and automatically terminates your app without requiring it to perform any additional work.

Your goal should be to design your app to be a good citizen: This means keeping the time it takes to move to the foreground as short as possible while also reducing its memory footprint while it is in the background.

Here's how you should handle the two scenarios:

- Your app should keep textures, models and other assets in memory; resources that take a long time to re-create should never be disposed of when your app moves into the background.
- Your app should dispose of objects that can be quickly and easily re-created. Look for objects that consume large amounts of memory.

Easy targets are the framebuffers your app allocates to hold rendering results. When your app is in the background, it is not visible to the user and may not render any new content using OpenGL ES. That means the memory consumed by your app's framebuffers is allocated, but is not useful. Also, the contents of the framebuffers are *transitory*; most app re-create the contents of the framebuffer every time they render a new frame. This makes renderbuffers a memory-intensive resource that can be easily re-created, becoming a good candidate for an object that can be disposed of when moving into the background.

If you use a GLKit view and view controller, the `GLKViewController` class automatically disposes of its associated view's framebuffers when your app moves into the background. If you manually create framebuffers for other uses, you should dispose of them when your app moves to the background. In either case, you should also consider what other transitory resources your app can dispose of at that time.

Supporting High-Resolution Displays

By default, the value of a GLKit view's `contentScaleFactor` property matches the scale of the screen that contains it, so its associated framebuffer is configured for rendering at the full resolution of the display. For more information on how high-resolution displays are supported in UIKit, see "Supporting High-Resolution Screens In Views".

If you present OpenGL ES content using a Core Animation layer, its scale factor is set to `1.0` by default. To draw at the full resolution of a Retina display, you should change the scale factor of the `CAEAGLLayer` object to match the screen's scale factor.

When supporting devices with high resolution displays, you should adjust the model and texture assets of your app accordingly. When running on a high-resolution device, you might want to choose more detailed models and textures to render a better image. Conversely, on a standard-resolution device, you can use smaller models and textures.

Important: Many OpenGL ES API calls express dimensions in screen pixels. If you use a scale factor greater than `1.0`, you should adjust dimensions accordingly when using the `glScissor`, `glBlitFramebuffer`, `glLineWidth`, or `glPointSize` functions or the `gl_PointSize` shader variable.

An important factor when determining how to support high-resolution displays is performance. The doubling of scale factor on a Retina display quadruples the number of pixels, causing the GPU to process four times as many fragments. If your app performs many per-fragment calculations, the increase in pixels may reduce the frame rate. If you find that your app runs significantly slower at a higher scale factor, consider one of the following options:

- Optimize your fragment shader's performance using the performance-tuning guidelines found in this document.
- Implement a simpler algorithm in your fragment shader. By doing so, you are reducing the quality of individual pixels to render the overall image at a higher resolution.
- Use a fractional scale factor between 1.0 and and the screen's scale factor. A scale factor of 1.5 provides better quality than a scale factor of 1.0 but needs to fill fewer pixels than an image scaled to 2.0.

- Use lower-precision formats for your `GLKView` object's `drawableColorFormat` and `drawableDepthFormat` properties. By doing this, you reduce the memory bandwidth required to operate on the underlying renderbuffers.
- Use a lower scale factor and enable multisampling. An added advantage is that multisampling also provides higher quality on devices that do not support high-resolution displays.

  To enable multisampling for a `GLKView` object, change the value of its `drawableMultisample` property. If you are not rendering to a GLKit view, you must manually set up multisampling buffers and resolve them before presenting a final image (see "Using Multisampling to Improve Image Quality" (page 40)).

  Multisampling is not free; additional memory is required to store the additional samples, and resolving the samples into the resolve framebuffer takes time. If you add multisampling to your app, always test your app's performance to ensure that it remains acceptable.

Supporting Multiple Interface Orientations

Like any app, an OpenGL ES app should support the user interface orientations appropriate to its content. You declare the supported interface orientations for your app in its information property list, or for the view controller hosting your OpenGL ES content using its `supportedInterfaceOrientations` method. (See *View Controller Programming Guide for iOS* for details.)

By default, the `GLKViewController` and `GLKView` classes handle orientation changes automatically: When the user rotates the device to a supported orientation, the system animates the orientation change and changes the size of the view controller's view. When its size changes, a `GLKView` object adjusts the size of its framebuffer and viewport accordingly. If you need to respond to this change, implement the `viewWillLayoutSubviews` or `viewDidLayoutSubviews` method in your `GLKViewController` subclass, or implement the `layoutSubviews` method if you're using a custom `GLKView` subclass.

If you draw OpenGL ES content using a Core Animation layer, your app should still include a view controller to manage user interface orientation.

Presenting OpenGL ES Content on External Displays

An iOS device can be attached to an external display. The resolution of an external display and its content scale factor may differ from the resolution and scale factor of the main screen; your code that renders a frame should adjust to match.

The procedure for drawing on an external display is almost identical to that running on the main screen.

1. Create a window on the external display by following the steps in *Multiple Display Programming Guide for iOS*.
2. Add to the window the appropriate view or view controller objects for your rendering strategy.
    - If rendering with GLKit, set up instances of `GLKViewController` and `GLKView` (or your custom subclasses) and add them to the window using its `rootViewController` property.
    - If rendering to a Core Animation layer, add the view containing your layer as a subview of the window. To use an animation loop for rendering, create a display link object optimized for the external display by retrieving the `screen` property of the window and calling its `displayLinkWithTarget:selector:` method.

OpenGL ES Design Guidelines

Now that you've mastered the basics of using OpenGL ES in an iOS app, use the information in this chapter to help you design your app's rendering engine for better performance. This chapter introduces key concepts of renderer design; later chapters expand on this information with specific best practices and performance techniques.

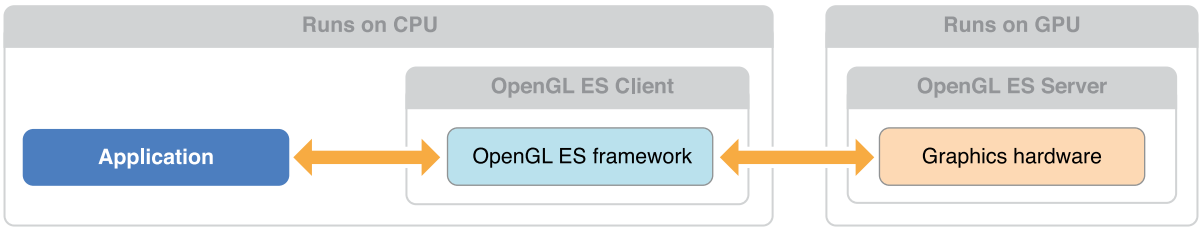
How to Visualize OpenGL ES

This section describes two perspectives for visualizing the design of OpenGL ES: as a client-server architecture and as a pipeline. Both perspectives can be useful in planning and evaluating the architecture of your app.

OpenGL ES as a Client-Server Architecture

Figure 6-1 visualizes OpenGL ES as a client-server architecture. Your app communicates state changes, texture and vertex data, and rendering commands to the OpenGL ES client. The client translates this data into a format that the graphics hardware understands, and forwards them to the GPU. These processes add overhead to your app's graphics performance.

Figure 6-1 OpenGL ES client-server architecture

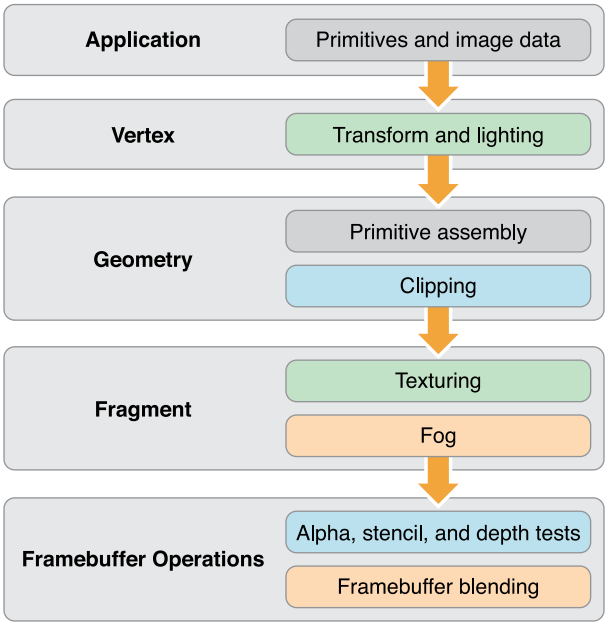


Achieving great performance requires carefully managing this overhead. A well-designed app reduces the frequency of calls it makes to OpenGL ES, uses hardware-appropriate data formats to minimize translation costs, and carefully manages the flow of data between itself and OpenGL ES.

OpenGL ES as a Graphics Pipeline

Figure 6-2 visualizes OpenGL ES as a graphics pipeline. Your app configures the graphics pipeline, and then executes drawing commands to send vertex data down the pipeline. Successive stages of the pipeline run a vertex shader to process the vertex data, assemble vertices into primitives, rasterize primitives into fragments, run a fragment shader to compute color and depth values for each fragment, and blend fragments into a framebuffer for display.

Figure 6-2 OpenGL ES graphics pipeline



Use the pipeline as a mental model to identify what work your app performs to generate a new frame. Your renderer design consists of writing shader programs to handle the vertex and fragment stages of the pipeline, organizing the vertex and texture data that you feed into these programs, and configuring the OpenGL ES state machine that drives fixed-function stages of the pipeline.

Individual stages in the graphics pipeline can calculate their results simultaneously—for example, your app might prepare new primitives while separate portions of the graphics hardware perform vertex and fragment calculations on previously submitted geometry. However, later stages depend on the output of earlier stages. If any pipeline stage performs too much work or performs too slowly, other pipeline stages sit idle until the slowest stage completes its work. A well-designed app balances the work performed by each pipeline stage according to graphics hardware capabilities.

Important: When you tune your app's performance, the first step is usually to determine which stage it is bottlenecked in, and why.

OpenGL ES Versions and Renderer Architecture

iOS supports three versions of OpenGL ES. Newer versions provide more flexibility, allowing you to implement rendering algorithms that include high-quality visual effects without compromising performance..

OpenGL ES 3.0

OpenGL ES 3.0 is new in iOS 7. Your app can use features introduced in OpenGL ES 3.0 to implement advanced graphics programming techniques—previously available only on desktop-class hardware and game consoles—for faster graphics performance and compelling visual effects.

Some key features of OpenGL ES 3.0 are highlighted below. For a complete overview, see the *OpenGL ES 3.0 Specification* in the OpenGL ES API Registry.

OpenGL ES Shading Language Version 3.0

GLSL ES 3.0 adds new features such as uniform blocks, 32-bit integers, and additional integer operations, for performing more general-purpose computing tasks within vertex and fragment shader programs. To use the new language in a shader program, your shader source code must begin with the `#version 330 es` directive. OpenGL ES 3.0 contexts remain compatible with shaders written for OpenGL ES 2.0.

For more details, see "Adopting OpenGL ES Shading Language version 3.0" (page 114) and the *OpenGL ES Shading Language 3.0 Specification* in the OpenGL ES API Registry.

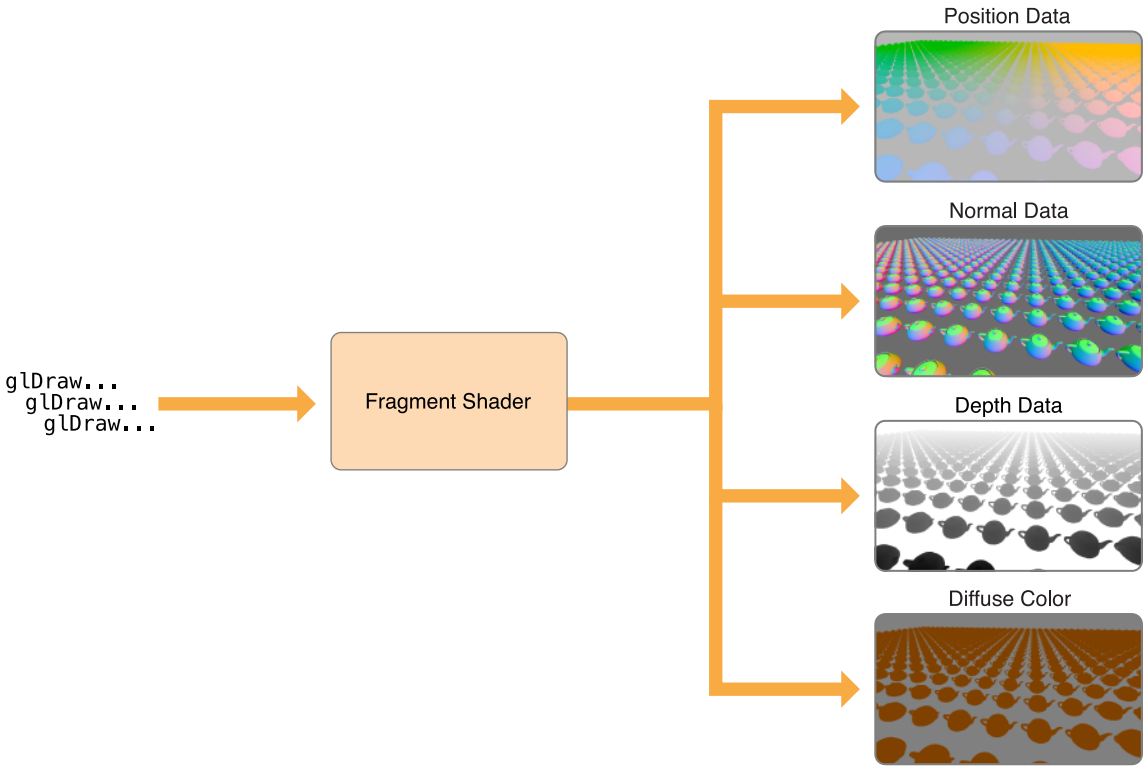
Multiple Render Targets

By enabling multiple render targets, you can create fragment shaders that write to multiple framebuffer attachments simultaneously.

This feature enables the use of advanced rendering algorithms such as **deferred shading**, in which your app first renders to a set of textures to store geometry data, then performs one or more shading passes that read from those textures and perform lighting calculations to output a final image. Because this approach precomputes the inputs to lighting calculations, the incremental performance cost for adding larger numbers

of lights to a scene is much smaller. Deferred shading algorithms require multiple render target support, as shown in Figure 6-3, to achieve reasonable performance. Otherwise, rendering to multiple textures requires a separate drawing pass for each texture.

Figure 6-3 Example of fragment shader output to multiple render targets



You set up multiple render targets with an addition to the process described in "Creating a Framebuffer Object" (page 32). Instead of creating a single color attachment for a framebuffer, you create several. Then, call the `glDrawBuffers` function to specify which framebuffer attachments to use in rendering, as shown in Listing 6-1.

Listing 6-1 Setting up multiple render targets

```
// Attach (previously created) textures to the framebuffer.
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
_colorTexture, 0);
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D,
_positionTexture, 0);
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT2, GL_TEXTURE_2D,
_normalTexture, 0);
```

```
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT,
GL_TEXTURE_2D, _depthTexture, 0);


// Specify the framebuffer attachments for rendering.
GLenum targets[] = {GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1,
GL_COLOR_ATTACHMENT2};
glDrawBuffers(3, targets);
```

When your app issues drawing commands, your fragment shader determines what color (or non-color data) is output for each pixel in each render target. Listing 6-2 shows a basic fragment shader that renders to multiple targets by assigning to fragment output variables whose locations match those set in Listing 6-1.

Listing 6-2 Fragment shader with output to multiple render targets

```
#version 300 es


uniform lowp sampler2D myTexture;
in mediump vec2 texCoord;
in mediump vec4 position;
in mediump vec3 normal;

layout(location = 0) out lowp vec4 colorData;
layout(location = 1) out mediump vec4 positionData;
layout(location = 2) out mediump vec4 normalData;

void main()
{
    colorData = texture(myTexture, texCoord);
    positionData = position;
    normalData = vec4(normalize(normal), 1.0);
}
```

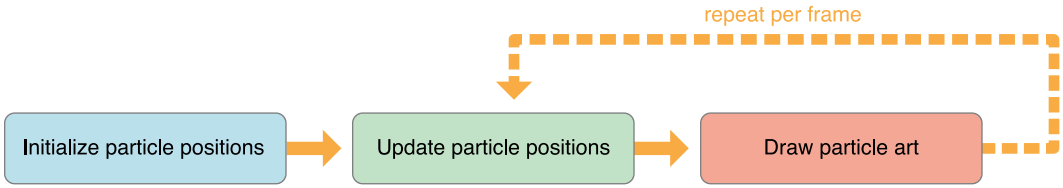
Multiple render targets can also be useful for other advanced graphics techniques, such as real-time reflections, screen-space ambient occlusion, and volumetric lighting.

Transform Feedback

Graphics hardware uses a highly parallelized architecture optimized for vector processing. You can make better use of this hardware with the new transform feedback feature, which lets you capture output from a vertex shader into a buffer object in GPU memory. You can capture data from one rendering pass to use in another, or disable parts of the graphics pipeline and use transform feedback for general-purpose computation.

One technique that benefits from transform feedback is animated particle effects. A general architecture for rendering a particle system is illustrated in Figure 6-4. First, the app sets up the initial state of the particle simulation. Then, for each frame rendered, the app runs a step of its simulation, updating the position, orientation, and velocity of each simulated particle, and then draws visual assets representing the current state of the particles.

Figure 6-4 Overview of a particle system animation

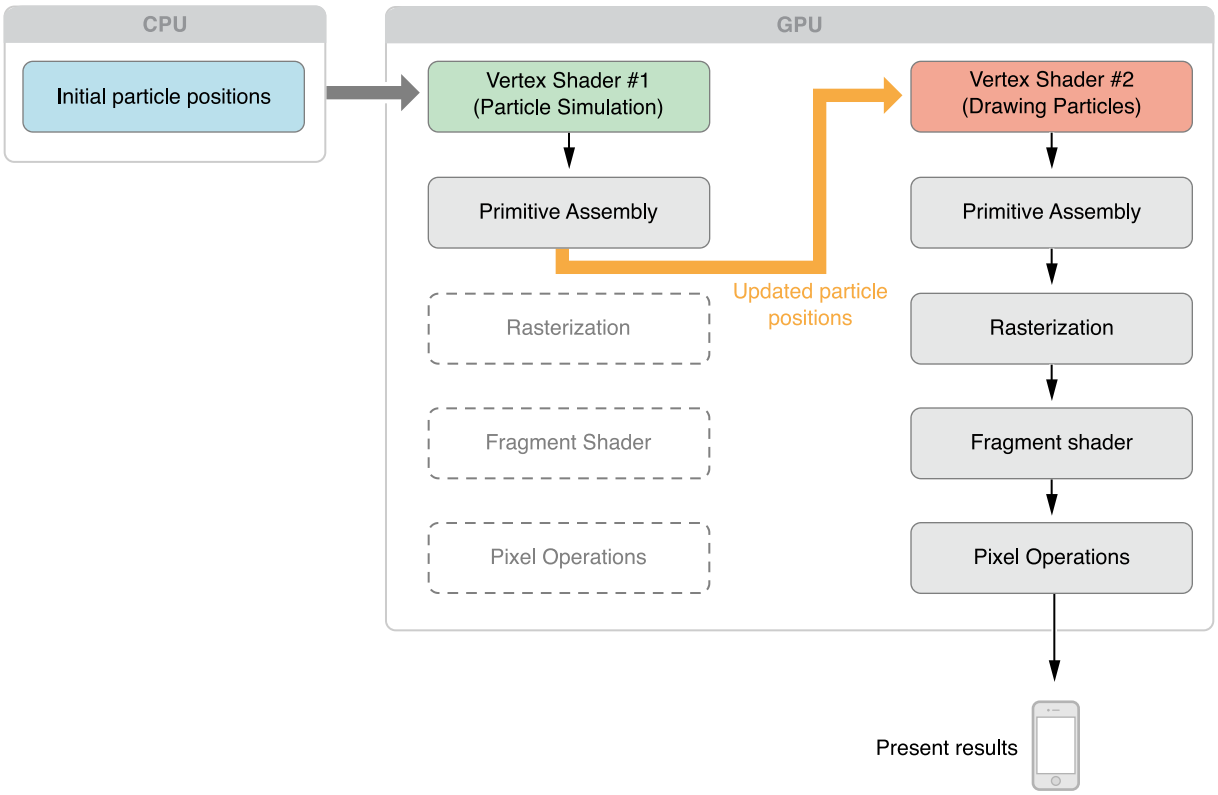


Traditionally, apps implementing particle systems run their simulations on the CPU, storing the results of the simulation in a vertex buffer to be used in rendering particle art. However, transferring the contents of the vertex buffer to GPU memory is time-consuming. Transform feedback, by optimizing the power of parallel architecture available in modern GPU hardware, solves the problem more efficiently.

With transform feedback, you can design your rendering engine to solve this problem more efficiently. Figure 6-5 shows an overview of how your app might configure the OpenGL ES graphics pipeline to implement a particle system animation. Because OpenGL ES represents each particle and its state as a vertex, the GPU's

vertex shader stage can run the simulation for several particles at once. Because the vertex buffer containing particle state data is reused between frames, the expensive process of transferring that data to GPU memory only happens once, at initialization time.

Figure 6-5 Example graphics pipeline configuration using transform feedback



1. At initialization time, create a vertex buffer and fill it with data containing the initial state of all particles in the simulation.
2. Implement your particle simulation in a GLSL vertex shader program, and run it each frame by drawing the contents of the vertex buffer containing particle position data.
    - To render with transform feedback enabled, call the `glBeginTransformFeedback` function. (Call `glEndTransformFeedback()` before resuming normal drawing.)
    - Use the `glTransformFeedbackVaryings` function to specify which shader outputs should be captured by transform feedback, and use the `glBindBufferBase` or `glBindBufferRange` function and `GL_TRANSFORM_FEEDBACK_BUFFER` buffer type to specify the buffer they will be captured into.
    - Disable rasterization (and subsequent stages of the pipeline) by calling `glEnable(GL_RASTERIZER_DISCARD)`.

3. To render the simulation results for display, use the vertex buffer containing particle positions as an input to second drawing pass, with rasterization (and the rest of the pipeline) once again enabled and using vertex and fragment shaders appropriate for rendering your app's visual content.
4. On the next frame, use the vertex buffer output by the last frame's simulation step as input to the next simulation step.

Other graphics programming techniques that can benefit from transform feedback include skeletal animation (also known as skinning) and ray marching.

OpenGL ES 2.0

OpenGL ES 2.0 provides a flexible graphics pipeline with programmable shaders, and is available on all current iOS devices. Many features formally introduced in the OpenGL ES 3.0 specification are available to iOS devices through OpenGL ES 2.0 extensions, so you can implement many advanced graphics programming techniques while remaining compatible with most devices.

OpenGL ES 1.1

OpenGL ES 1.1 provides only a basic fixed-function graphics pipeline. iOS supports OpenGL ES 1.1 primarily for backward compatibility. If you are maintaining an OpenGL ES 1.1 app, consider updating your code for newer OpenGL ES versions.

The GLKit framework can assist you in transitioning from the OpenGL ES 1.1 fixed-function pipeline to later versions. For details, read "Using GLKit to Develop Your Renderer" (page 31).

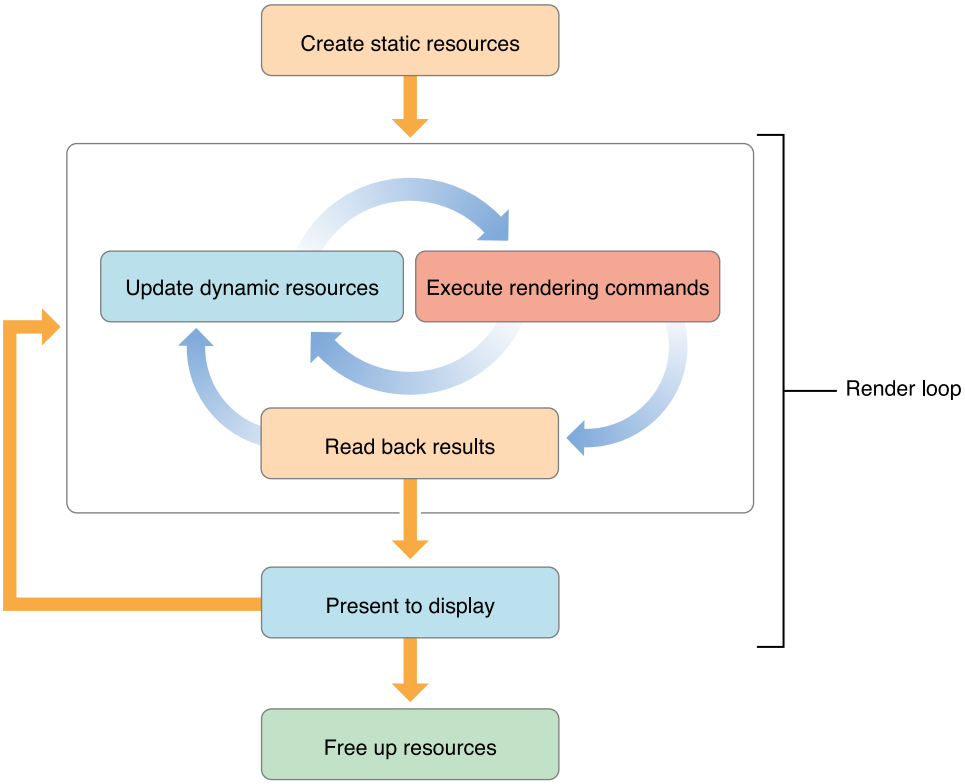
Designing a High-Performance OpenGL ES App

To summarize, a well-designed OpenGL ES app needs to:

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

Figure 6-6 suggests a process flow for an app that uses OpenGL ES to perform animation to the display.

Figure 6-6 App model for managing resources



When the app launches, the first thing it does is initialize resources that it does not intend to change over the lifetime of the app. Ideally, the app encapsulates those resources into OpenGL ES objects. The goal is to create any object that can remain unchanged for the runtime of the app (or even a portion of the app's lifetime, such as the duration of a level in a game), trading increased initialization time for better rendering performance. Complex commands or state changes should be replaced with OpenGL ES objects that can be used with a single function call. For example, configuring the fixed-function pipeline can take dozens of function calls. Instead, compile a graphics shader at initialization time, and switch to it at runtime with a single function call. OpenGL ES objects that are expensive to create or modify should almost always be created as static objects.

The rendering loop processes all of the items you intend to render to the OpenGL ES context, then presents the results to the display. In an animated scene, some data is updated for every frame. In the inner rendering loop shown in Figure 6-6, the app alternates between updating rendering resources (creating or modifying OpenGL ES objects in the process) and submitting drawing 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, preventing the app and OpenGL ES from accessing the same resources simultaneously. On iOS, modifying an OpenGL ES object can be expensive when the modification is not performed at the start or the end of a frame.

An important goal for this inner loop is to avoid copying data back from OpenGL ES to the app. Copying results from the GPU to the CPU can be very slow. If the copied data is also used later as part of the process of rendering the current frame, as shown in the middle rendering loop, your app blocks until all previously submitted drawing commands are completed.

After the app submits all drawing commands needed in the frame, it presents the results to the screen. A non-interactive app would copy the final image to app memory for further processing.

Finally, when your app is ready to quit, or when it finishes with a major task, it frees OpenGL ES objects to make additional resources available, either for itself or for other apps.

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. Try to avoid modifying dynamic resources except at the beginning or the end of a frame.
- Avoid reading intermediate rendering results back to your app.

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

- "Avoid Synchronizing and Flushing Operations" (page 57)
- "Avoid Querying OpenGL ES State" (page 58)
- "Use OpenGL ES to Manage Your Resources" (page 59)
- "Use Double Buffering to Avoid Resource Conflicts" (page 59)
- "Be Mindful of OpenGL ES State" (page 61)
- "Encapsulate State with OpenGL ES Objects" (page 61)

Avoid Synchronizing and Flushing Operations

The OpenGL ES specification doesn't require implementations to execute commands immediately. Often, commands are queued to a command buffer and executed by the hardware at a later time. Usually, OpenGL ES waits until the app has queued many commands before sending the commands to the hardware—batch processing is usually more efficient. However, some OpenGL ES functions must flush the command buffer immediately. Other functions not only flush the command buffer but also block until previously submitted commands have completed before returning control over the app. Use flushing and synchronizing commands only when that behavior is necessary. Excessive use of flushing or synchronizing commands may cause your app to stall while it waits for the hardware to finish rendering.

These situations require OpenGL ES to submit the command buffer to the hardware for execution.

- The function `glFlush` sends the command buffer to the graphics hardware. It blocks until commands are submitted to the hardware but does not wait for the commands to finish executing.
- The function `glFinish` flushes the command buffer and then waits for all previously submitted commands to finish executing on the graphics hardware.
- Functions that retrieve framebuffer content (such as `glReadPixels`) also wait for submitted commands to complete.
- The command buffer is full.

Using glFlush Effectively

On some desktop OpenGL implementations, it can be useful to periodically call the `glFlush` function to efficiently balance CPU and GPU work, but this is not the case in iOS. The Tile-Based Deferred Rendering algorithm implemented by iOS graphics hardware depends on buffering all vertex data in a scene at once, so it can be optimally processed for hidden surface removal. Typically, there are only two situations where an OpenGL ES app should call the `glFlush` or `glFinish` functions.

- You should flush the command buffer when your app moves to the background, because executing OpenGL ES commands on the GPU while your app is in the background causes iOS to terminate your app. (See "Implementing a Multitasking-Aware OpenGL ES App" (page 43).)
- If your app shares OpenGL ES objects (such as vertex buffers or textures) between multiple contexts, you should call the `glFlush` function to synchronize access to these resources. For example, you should call the `glFlush` function after loading vertex data in one context to ensure that its contents are ready to be retrieved by another context. This advice also applies when sharing OpenGL ES objects with other iOS APIs such as Core Image.

Avoid Querying OpenGL ES State

Calls to `glGet*()`, including `glGetError()`, may require OpenGL ES 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. To avoid this, maintain your own copy of any state you need to query, and access it directly, rather than calling OpenGL ES.

When errors occur, OpenGL ES sets an error flag. These and other errors appear in OpenGL ES Frame Debugger in Xcode or OpenGL ES Analyzer in Instruments. You should use those tools instead of the `glGetError` function, which degrades performance if called frequently. Other queries such as

`glCheckFramebufferStatus()`,`glGetProgramInfoLog()` and `glValidateProgram()` are also generally only useful while developing and debugging. You should omit calls to these functions in Release builds of your app.

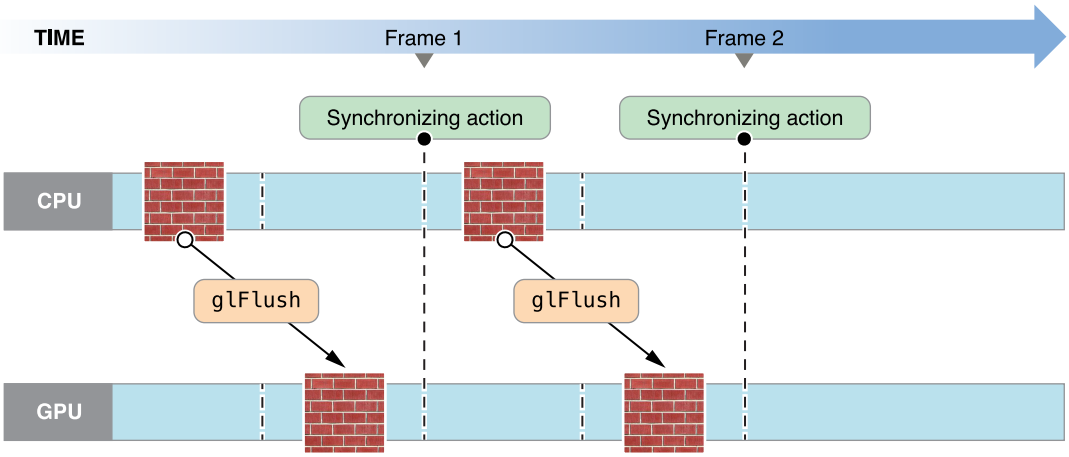
Use OpenGL ES to Manage Your Resources

Many pieces of OpenGL data can be stored directly inside the OpenGL ES rendering context and its associated sharegroup object. The OpenGL ES implementation is free to transform the data into a format that is optimal for the graphics hardware. This can significantly improve performance, especially for data that changes infrequently. Your app can also provide hints to OpenGL ES about how it intends to use the data. An OpenGL ES implementation can use these hints to process the data more efficiently. For example, static data might be placed in memory that the graphics processor can readily fetch, or even into dedicated graphics memory.

Use Double Buffering to Avoid Resource Conflicts

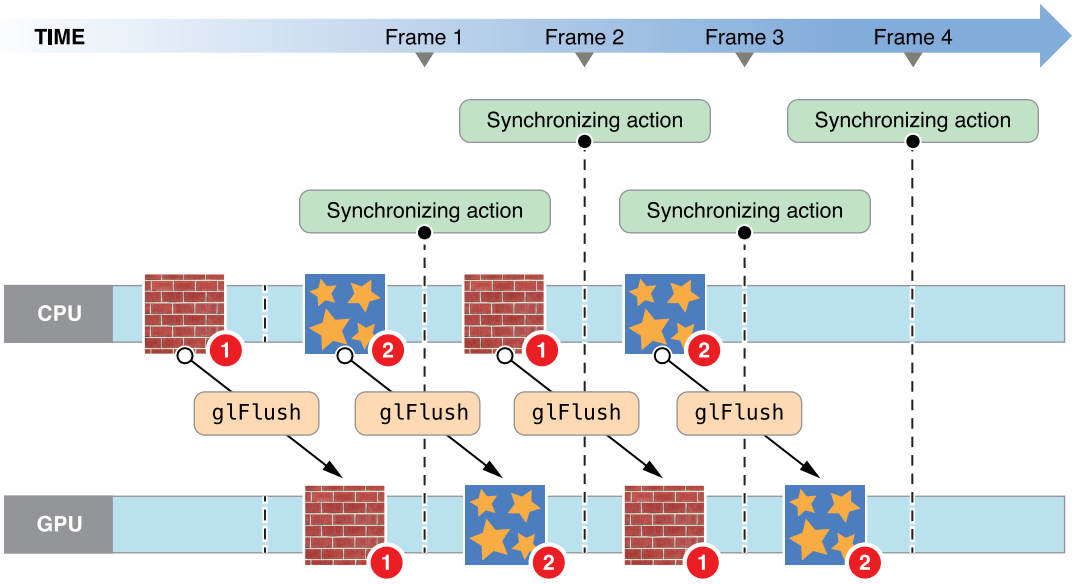
Resource conflicts occur when your app and OpenGL ES access an OpenGL ES object at the same time. When one participant attempts to modify an OpenGL ES object being used by the other, they may block until the object is no longer in use. Once they begin modifying the object, the other participant may not access the object until the modifications are complete. Alternatively, OpenGL ES may implicitly duplicate the object so that both participants can continue to execute commands. Either option is safe, but each can end up as a bottleneck in your app. Figure 6-7 shows this problem. In this example, there is a single texture object, which both OpenGL ES and your app want to use. When the app attempts to change the texture, it must wait until previously submitted drawing commands complete—the CPU synchronizes to the GPU.

Figure 6-7 Single-buffered texture data



To solve this problem, your app could perform additional work between changing the object and drawing with it. But, if your app does not have additional work it can perform, it should explicitly create two identically sized objects; while one participant reads an object, the other participant modifies the other. Figure 6-8 illustrates the double-buffered approach. While the GPU operates on one texture, the CPU modifies the other. After the initial startup, neither the CPU or GPU sits idle. Although shown for textures, this solution works for almost any type of OpenGL ES object.

Figure 6-8 Double-buffered texture data



Double buffering is sufficient for most apps, but it requires that both participants finish processing commands in roughly the same time. To avoid blocking, you can add more buffers; this implements a traditional producer-consumer model. If the producer finishes before the consumer finishes processing commands, it takes an idle buffer and continues to process commands. In this situation, the producer idles only if the consumer falls badly behind.

Double and triple buffering trade off consuming additional memory to prevent the pipeline from stalling. The additional use of memory may cause pressure on other parts of your app. On an iOS device, memory can be scarce; your design may need to balance using more memory with other app optimizations.

Be Mindful of OpenGL ES State

OpenGL ES implementations maintain a complex set of state data, including switches you set with the `glEnable` or `glDisable` functions, the current shader program and its uniform variables, currently bound texture units, and currently bound vertex buffers and their enabled vertex attributes. The hardware has one current state, which is compiled and cached lazily. Switching state is expensive, so it's best to design your app 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. For instance, if you call a `glUniform` function with the same arguments more than once, OpenGL ES may not check to see if the same uniform state is already set. It simply updates the state value even if that value is identical to the current value.

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.

Encapsulate State with OpenGL ES Objects

To reduce state changes, create objects that collect multiple OpenGL ES state changes into an object that can be bound with a single function call. For example, vertex array objects store the configuration of multiple vertex attributes into a single object. See "Consolidate Vertex Array State Changes Using Vertex Array Objects" (page 84).

Organize Draw Calls to Minimize State Changes

Changing OpenGL ES state has no immediate effect. Instead, when you issue a drawing command, OpenGL ES performs the work necessary to draw with a set of state values. You can reduce the CPU time spent reconfiguring the graphics pipeline by minimizing state changes. For example, keep a state vector in your app, and set the corresponding OpenGL ES state only if your state changes between draw calls. Another useful algorithm is state sorting—keep track of the drawing operations you need to do and the amount of state change necessary for each, then sort them to perform operations using the same state consecutively.

The iOS implementation of OpenGL ES can cache some of the configuration data it needs for efficient switching between states, but the initial configuration for each unique state set takes longer. For consistent performance, you can "prewarm" each state set you plan to use during a setup routine:

1. Enable a state configuration or shader you plan to use.
2. Draw a trivial number of vertices using that state configuration.
3. Flush the OpenGL ES context so that drawing during this prewarm phase is not displayed.

Tuning Your OpenGL ES App

The performance of OpenGL ES apps in iOS differs from that of OpenGL in OS X or other desktop operating systems. Although powerful computing devices, iOS–based devices do not have the memory or CPU power that desktop or laptop computers possess. Embedded GPUs are optimized for lower memory and power usage, using algorithms different from those a typical desktop or laptop GPU might use. Rendering your graphics data inefficiently can result in a poor frame rate or dramatically reduce the battery life of an iOS-based device.

Later chapters describe many techniques to improve your app's performance; this chapter covers overall strategies. Unless otherwise labeled, the advice in this chapter pertains to all versions of OpenGL ES.

Debug and Profile Your App with Xcode and Instruments

Don't optimize your app until you test its performance in a variety of scenarios on a variety of devices. Xcode and Instruments include tools to help you identify performance and correctness problems in your app.

- Monitor the **Xcode debug gauges** for a general overview of performance. These gauges are visible whenever you run your app from Xcode, making it easy to spot changes in performance while developing your app.
- Use the **OpenGL ES Analysis** and **OpenGL ES Driver** tools in Instruments for a deeper understanding of run-time performance. Get detailed information on your app's resource use and conformance to OpenGL ES best practices, and selectively disable portions of the graphics pipeline so you can determine which part is a significant bottleneck in your app. For more information, see *Instruments User Guide*.
- Use the **OpenGL ES Frame Debugger** and **Performance Analyzer** tools in Xcode for pinpoint troubleshooting of performance and rendering issues. Capture all OpenGL ES commands used to render and present a single frame, then walk through those commands to see the effect of each on the OpenGL ES state, bound resources, and the output framebuffer. You can also view shader source code, edit it, and see how your changes affect the rendered image. On OpenGL ES 3.0–capable devices, the Frame Debugger also indicates which draw calls and shader instructions contribute most to rendering time. For more information about these tools, see "Xcode OpenGL ES Tools Overview" (page 116).

Watch for OpenGL ES Errors in Xcode and Instruments

OpenGL ES errors occur when your app uses the OpenGL ES API incorrectly (for example, by requesting operations that the underlying hardware is not capable of performing). Even if your content renders correctly, these errors may indicate performance problems. The traditional way to check for OpenGL ES errors is to call the `glGetError` function; however, repeatedly calling this function can significantly degrade performance. Instead, use the tools described above to test for errors:

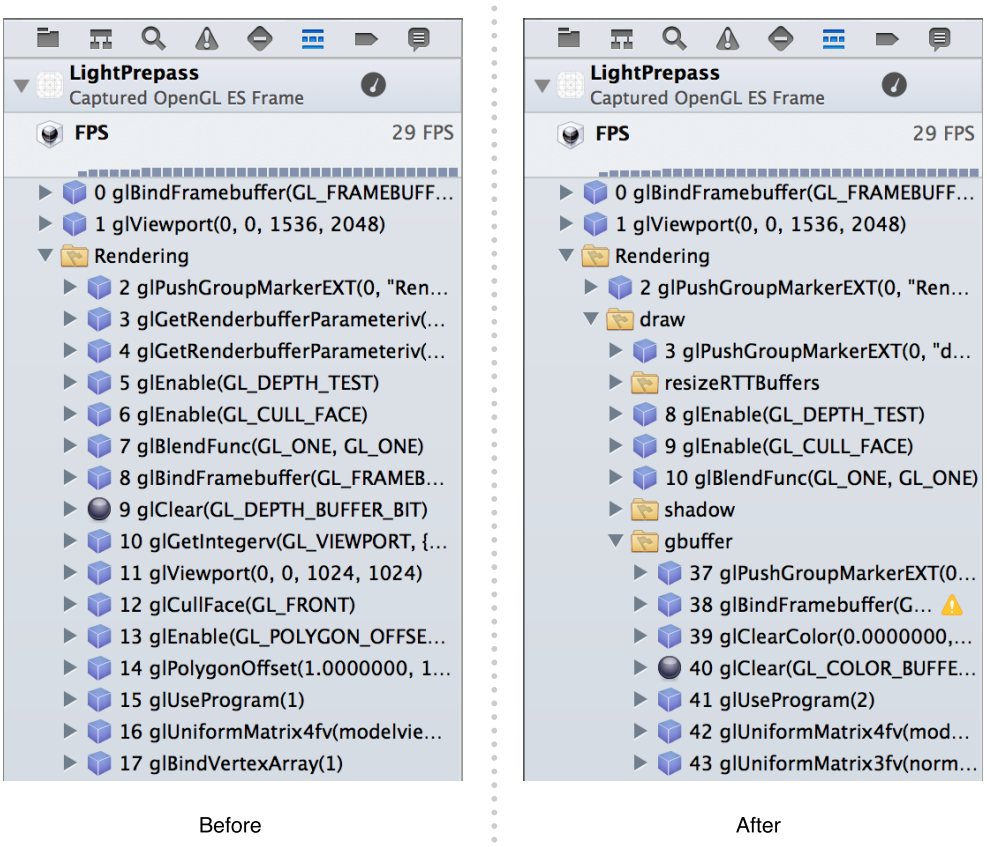
- When profiling your app in Instruments, see the detail pane for OpenGL ES Analyzer tool to view any OpenGL ES errors reported while recording.
- While debugging your app in Xcode, capture a frame to examine the drawing commands used to produce it, as well as any errors encountered while performing those commands.

You can also configure Xcode to stop program execution when an OpenGL ES error is encountered. (See Adding an OpenGL ES Error Breakpoint.)

Annotate Your OpenGL ES Code for Informative Debugging and Profiling

You can make debugging and profiling more efficient by organizing your OpenGL ES commands into logical groups and adding meaningful labels to OpenGL ES objects. These groups and labels appear in OpenGL ES Frame Debugger in Xcode as shown in Figure 7-1, and in OpenGL ES Analyzer in Instruments. To add groups and labels, use the `EXT_debug_marker` and `EXT_debug_label` extensions.

Figure 7-1 Xcode Frame Debugger before and after adding debug marker groups



When you have a sequence of drawing commands that represent a single meaningful operation—for example, drawing a game character—you can use a marker to group them for debugging. Listing 7-1 shows how to group the texture, program, vertex array, and draw calls for a single element of a scene. First, it calls the `glPushGroupMarkerEXT` function to provide a meaningful name, then it issues a group of OpenGL ES commands. Finally, it closes the group with a call to the `glPopGroupMarkerEXT` function.

Listing 7-1 Using a debug marker to annotate drawing commands

```
glPushGroupMarkerEXT(0, "Draw Spaceship");
glBindTexture(GL_TEXTURE_2D, _spaceshipTexture);
```

```
glUseProgram(_diffuseShading);
glBindVertexArrayOES(_spaceshipMesh);
glDrawElements(GL_TRIANGLE_STRIP, 256, GL_UNSIGNED_SHORT, 0);
glPopGroupMarkerEXT();
```

You can use multiple nested markers to create a hierarchy of meaningful groups in a complex scene. When you use the `GLKView` class to draw OpenGL ES content, it automatically creates a "Rendering" group containing all commands in your drawing method. Any markers you create are nested within this group.

Labels provide meaningful names for OpenGL ES objects, such as textures, shader programs, and vertex array objects. Call the `glLabelObjectEXT` function to give an object a name to be shown when debugging and profiling. Listing 7-2 illustrates using this function to label a vertex array object. If you use the `GLKTextureLoader` class to load texture data, it automatically labels the OpenGL ES texture objects it creates with their filenames.

Listing 7-2 Using a debug label to annotate an OpenGL ES object

```
glGenVertexArraysOES(1, &_spaceshipMesh);
glBindVertexArrayOES(_spaceshipMesh);
glLabelObjectEXT(GL_VERTEX_ARRAY_OBJECT_EXT, _spaceshipMesh, 0, "Spaceship");
```

General Performance Recommendations

Use common sense to guide your performance tuning efforts. For example, if your app draws only a few dozen triangles per frame, changing how it submits vertex data is unlikely to improve its performance. Look for optimizations that provide the most performance improvement for your effort.

Redraw Scenes Only When the Scene Data Changes

Your app should wait until something in the scene changes before rendering a new frame. Core Animation caches the last image presented to the user and continues to display it until a new frame is presented.

Even when your data changes, it is not necessary to render frames at the speed the hardware processes commands. A slower but fixed frame rate often appears smoother to the user than a fast but variable frame rate. A fixed frame rate of 30 frames per second is sufficient for most animation and helps reduce power consumption.

Disable Unused OpenGL ES Features

The best calculation is one that your app never performs. For example, if a result can be precalculated and stored in your model data, you can avoid performing that calculation at runtime.

If your app is written for OpenGL ES 2.0 or later, do not create a single shader with lots of switches and conditionals that performs every task your app needs to render the scene. Instead, compile multiple shader programs that each perform a specific, focused task.

If your app uses OpenGL ES 1.1, disable any fixed-function operations that are not necessary to render the scene. For example, if your app does not require lighting or blending, disable those functions. Similarly, if your app draws only 2D models, it should disable fog and depth testing.

Simplify Your Lighting Models

These guidelines apply both to fixed-function lighting in OpenGL ES 1.1 and shader-based lighting calculations you use in your custom shaders in OpenGL ES 2.0 or later.

- Use the fewest lights possible and the simplest lighting type for your app. Consider using directional lights instead of spot lighting, which require more calculations. Shaders should perform lighting calculations in model space; consider using simpler lighting equations in your shaders over more complex lighting algorithms.
- Pre-compute your lighting and store the color values in a texture that can be sampled by fragment processing.

Use Tile-Based Deferred Rendering Efficiently

All GPUs used in iOS devices use **tile-based deferred rendering (TBDR)**. When you call OpenGL ES functions to submit rendering commands to the hardware, the commands are buffered until a large list of commands is accumulated. The hardware does not begin processing vertices and shading pixels until you present the renderbuffer or flush the command buffer. It then renders these commands as a single operation by dividing the framebuffer into tiles and then drawing the commands once for each tile, with each tile rendering only the primitives that are visible within it. (The `GLKView` class presents the renderbuffer after your drawing method returns. If you create your own renderbuffer for display using the `CAEAGLLayer` class, you present it using the OpenGL ES context's `presentRenderbuffer:` method. The `glFlush` or `glFinish` function flushes the command buffer.)

Because tile memory is part of the GPU hardware, parts of the rendering process such as depth testing and blending are much more efficient—in both time and energy usage—than on a traditional stream-based GPU architecture. Because this architecture processes all vertices for an entire scene at once, the GPU can perform

hidden surface removal before fragments are processed. Pixels that are not visible are discarded without sampling textures or performing fragment processing, significantly reducing the calculations that the GPU must perform to render the tile.

Some rendering strategies that are useful on traditional stream-based renderer have a high performance costs on iOS graphics hardware. Following the guidelines below can help your app perform well on TBDR hardware.

Avoid Logical Buffer Loads and Stores

When a TBDR graphics processor begins rendering a tile, it must first transfer the contents of that portion of the framebuffer from shared memory to the tile memory on the GPU. This memory transfer, called a **logical buffer load**, takes time and energy. Most often, a framebuffer's previous contents are not necessary for drawing the next frame. Avoid the performance cost of loading previous buffer contents by calling `glClear` whenever you begin rendering a new frame.

Similarly, when the GPU finishes rendering a tile, it must write the tile's pixel data back to shared memory. This transfer, called a **logical buffer store**, also has a performance cost. At least one such transfer is necessary for each frame drawn—the color renderbuffer displayed on the screen must be transferred to shared memory so it can be presented by Core Animation. Other framebuffer attachments used in your rendering algorithm (for example, depth, stencil, and multisampling buffers) need not be preserved, because their contents will be recreated on the next frame drawn. OpenGL ES automatically stores these buffers to shared memory—incurring a performance cost—unless you explicitly invalidate them. To invalidate a buffer, use the `glInvalidateFramebuffer` command in OpenGL ES 3.0 or the `glDiscardFramebufferEXT` command in OpenGL ES 1.1 or 2.0. (For more details, see "Discard Unneeded Renderbuffers" (page 39).) When you use the basic drawing cycle provided by `GLKView` class, it automatically invalidates any drawable depth, stencil, or multisampling buffers it creates.

Logical buffer store and load operations also occur if you switch rendering destinations. If you render to a texture, then to a view's framebuffer, then to the same texture again, the texture's contents must be repeatedly transferred between shared memory and the GPU. Batch your drawing operations so that all drawing to a rendering destination is done together. When you do switch framebuffers (using the `glBindFramebuffer` or `glFramebufferTexture2D` function or `bindDrawable` method), invalidate unneeded framebuffer attachments to avoid causing a logical buffer store.

Use Hidden Surface Removal Effectively

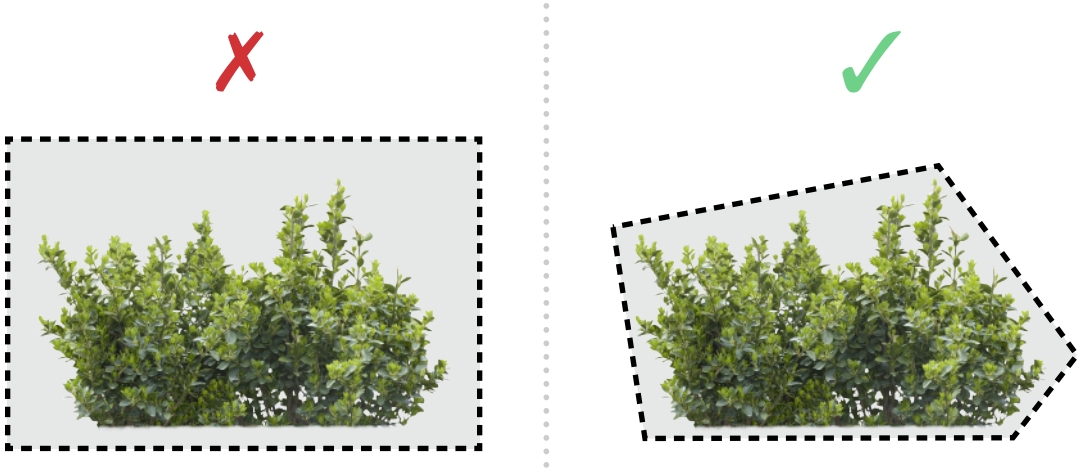
A TBDR graphics processor automatically uses the depth buffer to perform hidden surface removal for the entire scene, ensuring that only one fragment shader is run for each pixel. Traditional techniques for reducing fragment processing are not necessary. For example, sorting objects or primitives by depth from front to back effectively duplicates the work done by the GPU, wasting CPU time.

The GPU cannot perform hidden surface removal when blending or alpha testing is enabled, or if a fragment shader uses the `discard` instruction or writes to the `gl_FragDepth` output variable. In these cases, the GPU cannot decide the visibility of a fragment using the depth buffer, so it must run the fragment shaders for all primitives covering each pixel, greatly increasing the time and energy required to render a frame. To avoid this performance cost, minimize your use of blending, `discard` instructions, and depth writes.

If you cannot avoid blending, alpha testing, or discard instructions, consider the following strategies for reducing their performance impact:

- Sort objects by opacity. Draw opaque objects first. Next draw objects requiring a shader using the `discard` operation (or alpha testing in OpenGL ES 1.1). Finally, draw alpha-blended objects.
- Trim objects requiring blending or `discard` instructions to reduce the number of fragments processed. For example, instead of drawing a square to render a 2D sprite texture containing mostly empty space, draw a polygon that more closely approximates the shape of the image, as shown in Figure 7-2. The performance cost of additional vertex processing is much less than that of running fragment shaders whose results will be unused.

Figure 7-2 Trim transparent objects to reduce fragment processing



- Use the `discard` instruction as early as possible in your fragment shader to avoid performing calculations whose results are unused.
- Instead of using alpha testing or `discard` instructions to kill pixels, use alpha blending with alpha set to zero. The color framebuffer is not modified, but the graphics hardware can still use any Z-buffer optimizations it performs. This does change the value stored in the depth buffer and so may require back-to-front sorting of the transparent primitives.

- If your performance is limited by unavoidable discard operations, consider a "Z-Prepass" rendering strategy. Render your scene once with a simple fragment shader containing only your discard logic (avoiding expensive lighting calculations) to fill the depth buffer. Then, render your scene again using the `GL_EQUAL` depth test function and your lighting shaders. Though multipass rendering normally incurs a performance penalty, this approach can yield better performance than a single-pass render that involves a large number of discard operations.

Group OpenGL ES Commands for Efficient Resource Management

The memory bandwidth and computational savings described above perform best when processing large scenes. But when the hardware receives OpenGL ES commands that require it to render smaller scenes, the renderer loses much of its efficiency. For example, if your app renders batches of triangles using a texture and then modifies the texture, the OpenGL ES implementation must either flush out those commands immediately or duplicate the texture—neither option uses the hardware efficiently. Similarly, any attempt to read pixel data from a framebuffer requires that preceding commands be processed if they would alter that framebuffer.

To avoid these performance penalties, organize your sequence of OpenGL ES calls so that all drawing commands for each rendering target are performed together.

Minimize the Number of Drawing Commands

Every time your app submits primitives to be processed by OpenGL ES, the CPU prepares the commands for the graphics hardware. If your app uses many `glDrawArrays` or `glDrawElements` calls to render a scene, its performance may be limited by CPU resources without fully utilizing the GPU.

To reduce this overhead, look for ways to consolidate your rendering into fewer draw calls. Useful strategies include:

- Merging multiple primitives into a single triangle strip, as described in "Use Triangle Strips to Batch Vertex Data" (page 78). For best results, consolidate primitives that are drawn in close spatial proximity. Large, sprawling models are more difficult for your app to efficiently cull when they are not visible in the frame.
- Creating texture atlases to draw multiple primitives using different portions of the same texture image, as described in "Combine Textures into Texture Atlases" (page 92).
- Using instanced drawing to render many similar objects, as described below.

Use Instanced Drawing to Minimize Draw Calls

Instanced drawing commands allow you to draw the same vertex data multiple times using a single draw call. Instead of using CPU time to set up the variations between different instances of a mesh—such as a position offset, transformation matrix, color or texture coordinates—and issuing a draw command for each, instanced drawing moves the processing of instance variations into shader code run on the GPU.

Vertex data that is reused multiple times is a prime candidate for instanced drawing. For example, the code in Listing 7-3 draws an object at multiple positions within a scene. However, the many `glUniform` and `glDrawArrays` calls add CPU overhead, reducing performance.

Listing 7-3 Drawing many similar objects without instancing

```
for (x = 0; x < 10; x++) {
    for (y = 0; y < 10; y++) {
        glUniform4fv(uniformPositionOffset, 1, positionOffsets[x][y]);
        glDrawArrays(GL_TRIANGLES, 0, numVertices);
    }
}
```

Adopting instanced drawing requires two steps: first, replace loops like the above with a single call to `glDrawArraysInstanced` or `glDrawElementsInstanced`. These calls are otherwise identical to `glDrawArrays` or `glDrawElements`, but with an additional parameter indicating the number of instances to draw (`100` for the example in Listing 7-3). Second, choose and implement one of the two strategies OpenGL ES provides for using per-instance information in your vertex shader.

With the **shader instance ID** strategy, your vertex shader derives or looks up per-instance information. Each time the vertex shader runs, its `gl_InstanceID` built-in variable contains a number identifying the instance currently being drawn. Use this number to calculate a position offset, color, or other per-instance variation in shader code, or to look up per-instance information in a uniform array or other bulk storage. For example, Listing 7-4 uses this technique to draw 100 instances of a mesh positioned in a 10 x 10 grid.

Listing 7-4 OpenGL ES 3.0 vertex shader using `gl_InstanceID` to compute per-instance information

```
#version 300 es

in vec4 position;

uniform mat4 modelViewProjectionMatrix;
```

```
void main()
{
    float xOffset = float(gl_InstanceID % 10) * 0.5 - 2.5;
    float yOffset = float(gl_InstanceID / 10) * 0.5 - 2.5;
    vec4 offset = vec4(xOffset, yOffset, 0, 0);

    gl_Position = modelViewProjectionMatrix * (position + offset);
}
```

With the **instanced arrays** strategy, you store per-instance information in a vertex array attribute. Your vertex shader can then access that attribute to make use of per-instance information. Call the `glVertexAttribDivisor` function to specify how that attribute advances as OpenGL ES draws each instance. Listing 7-5 demonstrates setting up a vertex array for instanced drawing, and Listing 7-6 shows the corresponding shader.

Listing 7-5 Using a vertex attribute for per-instance information

```
#define kMyInstanceDataAttrib 5

glGenBuffers(1, &_instBuffer);
glBindBuffer(GL_ARRAY_BUFFER, _instBuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(instData), instData, GL_STATIC_DRAW);
glEnableVertexAttribArray(kMyInstanceDataAttrib);
glVertexAttribPointer(kMyInstanceDataAttrib, 2, GL_FLOAT, GL_FALSE, 0, 0);
glVertexAttribDivisor(kMyInstanceDataAttrib, 1);
```

Listing 7-6 OpenGL ES 3.0 vertex shader using instanced arrays

```
#version 300 es

layout(location = 0) in vec4 position;
layout(location = 5) in vec2 inOffset;

uniform mat4 modelViewProjectionMatrix;

void main()
```

```
{
    vec4 offset = vec4(inOffset, 0.0, 0.0)
    gl_Position = modelViewProjectionMatrix * (position + offset);
}
```

Instanced drawing is available in the core OpenGL ES 3.0 API and in OpenGL ES 2.0 through the `EXT_draw_instanced` and `EXT_instanced_arrays` extensions.

Minimize OpenGL ES Memory Usage

Your iOS app shares main memory with the system and other iOS apps. Memory allocated for OpenGL ES reduces the memory available for other uses in your app. With that in mind, allocate only the memory that you need and deallocate it as soon as your app no longer needs it. Here are a few ways you can save memory:

- After loading an image into an OpenGL ES texture, free the original image.
- Allocate a depth buffer only when your app requires it.
- If your app does not need all of its resources at once, load only a subset of the items. For example, a game might be divided into levels; each loads a subset of the total resources that fits within a more strict resource limit.

The virtual memory system in iOS does not use a swap file. When a low-memory condition is detected, instead of writing volatile pages to disk, the virtual memory frees up nonvolatile memory to give your running app the memory it needs. Your app should strive to use as little memory as possible and be prepared to dispose of objects that are not essential to your app. Responding to low-memory conditions is covered in detail in the *iOS App Programming Guide*.

Be Aware of Core Animation Compositing Performance

Core Animation composites the contents of renderbuffers with any other layers in your view hierarchy, regardless of whether those layers were drawn with OpenGL ES, Quartz or other graphics libraries. That's helpful, because it means that OpenGL ES is a first-class citizen to Core Animation. However, mixing OpenGL ES content with other content takes time; when used improperly, your app may perform too slowly to reach interactive frame rates.

For the absolute best performance, your app should rely solely on OpenGL ES to render your content. Size the view that holds your OpenGL ES content to match the screen, make sure its `opaque` property is set to `YES` (the default for `GLKView` objects) and that no other views or Core Animation layers are visible.

If you render into a Core Animation layer that is composited on top of other layers, making your `CAEAGLLayer` object opaque reduces—but doesn't eliminate—the performance cost. If your `CAEAGLLayer` object is blended on top of layers underneath it in the layer hierarchy, the renderbuffer's color data must be in a premultiplied alpha format to be composited correctly by Core Animation. Blending OpenGL ES content on top of other content has a severe performance penalty.

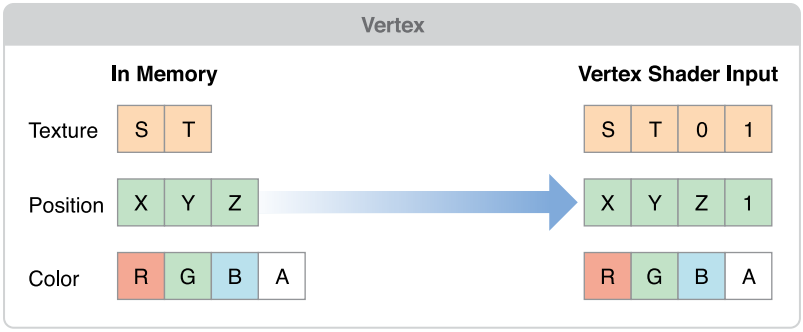
Best Practices for Working with Vertex Data

To render a frame using OpenGL ES your app configures the graphics pipeline and submits graphics primitives to be drawn. In some apps, all primitives are drawn using the same pipeline configuration; other apps may render different elements of the frame using different techniques. But no matter which primitives you use in your app or how the pipeline is configured, your app provides vertices to OpenGL ES. This chapter provides a refresher on vertex data and follows it with targeted advice for how to efficiently process vertex data.

A vertex consists of one or more **attributes**, such as the position, the color, the normal, or texture coordinates. An OpenGL ES 2.0 or 3.0 app is free to define its own attributes; each attribute in the vertex data corresponds to an attribute variable that acts as an input to the vertex shader. An OpenGL 1.1 app uses attributes defined by the fixed-function pipeline.

You define an attribute as a vector consisting of one to four **components**. All components in the attribute share a common data type. For example, a color might be defined as four `GLubyte` components (red, green, blue, alpha). When an attribute is loaded into a shader variable, any components that are not provided in the app data are filled in with default values by OpenGL ES. The last component is filled with `1`, and other unspecified components are filled with `0`, as illustrated in Figure 8-1.

Figure 8-1 Conversion of attribute data to shader variables



Your app may configure an attribute to be a *constant*, which means the same values are used for all vertices submitted as part of a draw command, or an *array*, which means that each vertex a value for that attribute. When your app calls a function in OpenGL ES to draw a set of vertices, the vertex data is copied from your app to the graphics hardware. The graphics hardware than acts on the vertex data, processing each vertex in the shader, assembling primitives and rasterizing them out into the framebuffer. One advantage of OpenGL ES is that it standardizes on a single set of functions to submit vertex data to OpenGL ES, removing older and less efficient mechanisms that were provided by OpenGL.

Apps that must submit a large number of primitives to render a frame need to carefully manage their vertex data and how they provide it to OpenGL ES. The practices described in this chapter can be summarized in a few basic principles:

- Reduce the size of your vertex data.
- Reduce the pre-processing that must occur before OpenGL ES can transfer the vertex data to the graphics hardware.
- Reduce the time spent copying vertex data to the graphics hardware.
- Reduce computations performed for each vertex.

Simplify Your Models

The graphics hardware of iOS-based devices is very powerful, but the images it displays are often very small. You don't need extremely complex models to present compelling graphics on iOS. Reducing the number of vertices used to draw a model directly reduces the size of the vertex data and the calculations performed on your vertex data.

You can reduce the complexity of a model by using some of the following techniques:

- Provide multiple versions of your model at different levels of detail, and choose an appropriate model at runtime based on the distance of the object from the camera and the dimensions of the display.
- Use textures to eliminate the need for some vertex information. For example, a bump map can be used to add detail to a model without adding more vertex data.
- Some models add vertices to improve lighting details or rendering quality. This is usually done when values are calculated for each vertex and interpolated across the triangle during the rasterization stage. For example, if you directed a spotlight at the center of a triangle, its effect might go unnoticed because the brightest part of the spotlight is not directed at a vertex. By adding vertices, you provide additional interpolant points, at the cost of increasing the size of your vertex data and the calculations performed on the model. Instead of adding additional vertices, consider moving calculations into the fragment stage of the pipeline instead:
    - If your app uses OpenGL ES 2.0 or later, then your app performs the calculation in the vertex shader and assigns it to a varying variable. The varying value is interpolated by the graphics hardware and passed to the fragment shader as an input. Instead, assign the calculation's inputs to varying variables and perform the calculation in the fragment shader. Doing this changes the cost of performing that calculation from a per-vertex cost to a per-fragment cost, reduces pressure on the vertex stage and more pressure on the fragment stage of the pipeline. Do this when your app is blocked on vertex processing, the calculation is inexpensive and the vertex count can be significantly reduced by the change.

- If your app uses OpenGL ES 1.1, you can perform per-fragment lighting using DOT3 lighting. You do this by adding a bump map texture to hold normal information and applying the bump map using a texture combine operation with the `GL_DOT3_RGB` mode.

Avoid Storing Constants in Attribute Arrays

If your models include attributes that uses data that remains constant across the entire model, do not duplicate that data for each vertex. OpenGL ES 2.0 and 3.0 apps can either set a constant vertex attribute or use a uniform shader value to hold the value instead. OpenGL ES 1.1 app should use a per-vertex attribute function such as `glColor4ub` or `glTexCoord2f` instead.

Use the Smallest Acceptable Types for Attributes

When specifying the size of each of your attribute's components, choose the smallest data type that provides acceptable results. Here are some guidelines:

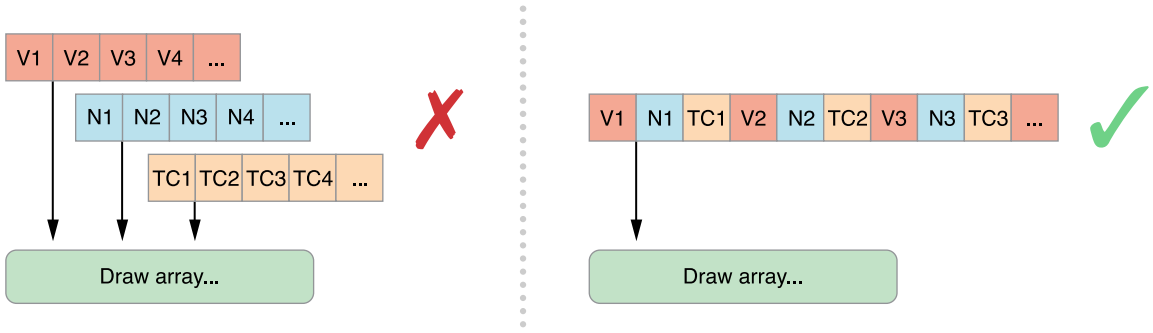
- Specify vertex colors using four unsigned byte components (`GL_UNSIGNED_BYTE`).
- Specify texture coordinates using 2 or 4 unsigned bytes (`GL_UNSIGNED_BYTE`) or unsigned short (`GL_UNSIGNED_SHORT`). Do not pack multiple sets of texture coordinates into a single attribute.
- Avoid using the OpenGL ES `GL_FIXED` data type. It requires the same amount of memory as `GL_FLOAT`, but provides a smaller range of values. All iOS devices support hardware floating-point units, so floating point values can be processed more quickly.
- OpenGL ES 3.0 contexts support a wider range of small data types, such as `GL_HALF_FLOAT` and `GL_INT_2_10_10_10_REV`. These often provide sufficient precision for attributes such as normals, with a smaller footprint than `GL_FLOAT`.

If you specify smaller components, be sure you reorder your vertex format to avoid misaligning your vertex data. See "Avoid Misaligned Vertex Data" (page 77).

Use Interleaved Vertex Data

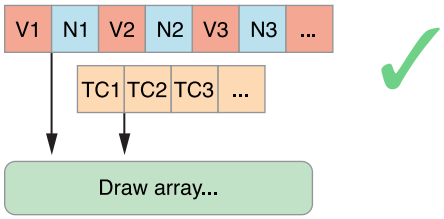
You can specify vertex data as a series of arrays (also known as a *struct of arrays*) or as an array where each element includes multiple attributes (an *array of structs*). The preferred format on iOS is an array of structs with a single interleaved vertex format. Interleaved data provides better memory locality for each vertex.

Figure 8-2 Interleaved memory structures place all data for a vertex together in memory



An exception to this rule is when your app needs to update some vertex data at a rate different from the rest of the vertex data, or if some data can be shared between two or more models. In either case, you may want to separate the attribute data into two or more structures.

Figure 8-3 Use multiple vertex structures when some data is used differently

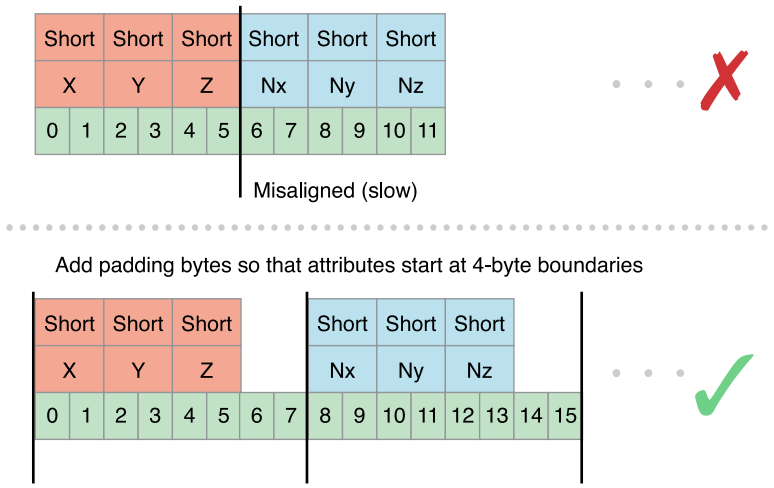


Avoid Misaligned Vertex Data

When you are designing your vertex structure, align the beginning of each attribute to an offset that is either a multiple of its component size or 4 bytes, whichever is larger. When an attribute is misaligned, iOS must perform additional processing before passing the data to the graphics hardware.

In Figure 8-4 (page 78), the position and normal data are each defined as three short integers, for a total of six bytes. The normal data begins at offset 6, which is a multiple of the native size (2 bytes), but is not a multiple of 4 bytes. If this vertex data were submitted to iOS, iOS would have to take additional time to copy and align the data before passing it to the hardware. To fix this, explicitly add two bytes of padding after each attribute.

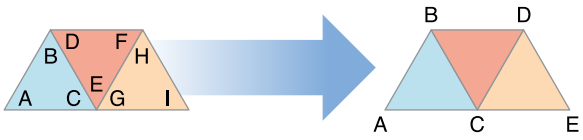
Figure 8-4 Align Vertex Data to avoid additional processing



Use Triangle Strips to Batch Vertex Data

Using triangle strips significantly reduces the number of vertex calculations that OpenGL ES must perform on your models. On the left side of Figure 8-5, three triangles are specified using a total of nine vertices. C, E and G actually specify the same vertex! By specifying the data as a triangle strip, you can reduce the number of vertices from nine to five.

Figure 8-5 Triangle strip

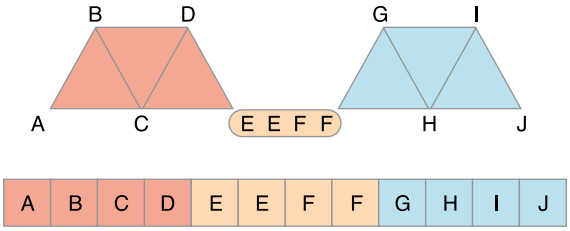


Sometimes, your app can combine more than one triangle strip into a single larger triangle strip. All of the strips must share the same rendering requirements. This means:

- You must use the same shader to draw all of the triangle strips.
- You must be able to render all of the triangle strips without changing any OpenGL state.
- The triangle strips must share the same vertex attributes.

To merge two triangle strips, duplicate the last vertex of the first strip and the first vertex of the second strip, as shown in Figure 8-6. When this strip is submitted to OpenGL ES, triangles DEE, EEF, EFF, and FFG are considered degenerate and not processed or rasterized.

Figure 8-6 Use degenerate triangles to merge triangle strips



For best performance, your models should be submitted as a single indexed triangle strip. To avoid specifying data for the same vertex multiple times in the vertex buffer, use a separate index buffer and draw the triangle strip using the `glDrawElements` function (or the `glDrawElementsInstanced` or `glDrawRangeElements` functions, if appropriate).

In OpenGL ES 3.0, you can use the primitive restart feature to merge triangle strips without using degenerate triangles. When this feature is enabled, OpenGL ES treats the largest possible value in an index buffer as a command to finish one triangle strip and start another. Listing 8-1 demonstrates this approach.

Listing 8-1 Using primitive restart in OpenGL ES 3.0

```
// Prepare index buffer data (not shown: vertex buffer data, loading vertex and
index buffers)
GLushort indexData[11] = {
    0, 1, 2, 3, 4,    // triangle strip ABCDE
    0xFFFF,          // primitive restart index (largest possible GLushort value)
    5, 6, 7, 8, 9,    // triangle strip FGHIJ
};

// Draw triangle strips
glEnable(GL_PRIMITIVE_RESTART_FIXED_INDEX);
glDrawElements(GL_TRIANGLE_STRIP, 11, GL_UNSIGNED_SHORT, 0);
```

Where possible, sort vertex and index data so triangles that share common vertices are drawn reasonably close to each other in the triangle strip. Graphics hardware often caches recent vertex calculations to avoid recalculating a vertex.

Use Vertex Buffer Objects to Manage Copying Vertex Data

Listing 8-2 provides a function that a simple app might use to provide position and color data to the vertex shader. It enables two attributes and configures each to point at the interleaved vertex structure. Finally, it calls the `glDrawElements` function to render the model as a single triangle strip.

Listing 8-2 Submitting vertex data to a shader program

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

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

    glVertexAttribPointer(GLKVertexAttribPosition, 2, GL_FLOAT, GL_FALSE,
        sizeof(vertexStruct), &vertices[0].position);
    glEnableVertexAttribArray(GLKVertexAttribPosition);
    glVertexAttribPointer(GLKVertexAttribColor, 4, GL_UNSIGNED_BYTE, GL_TRUE,
        sizeof(vertexStruct), &vertices[0].color);
    glEnableVertexAttribArray(GLKVertexAttribColor);

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

This code works, but is inefficient. Each time `DrawModel` is called, the index and vertex data are copied to OpenGL ES, and transferred to the graphics hardware. If the vertex data does not change between invocations, these unnecessary copies can impact performance. To avoid unnecessary copies, your app should store its vertex data in a **vertex buffer object** (VBO). Because OpenGL ES owns the vertex buffer object's memory, it can store the buffer in memory that is more accessible to the graphics hardware, or pre-process the data into the preferred format for the graphics hardware.

Note: When using vertex array objects in OpenGL ES 3.0, you must also use vertex buffer objects.

Listing 8-3 creates a pair of vertex buffer objects, one to hold the vertex data and the second for the strip's indices. In each case, the code generates a new object, binds it to be the current buffer, and fills the buffer. `CreateVertexBuffers` would be called when the app is initialized.

Listing 8-3 Creating a vertex buffer object

```
GLuint    vertexBuffer;
GLuint    indexBuffer;

void CreateVertexBuffers()
{

    glGenBuffers(1, &vertexBuffer);
    glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    glGenBuffers(1, &indexBuffer);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);
   glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);

}
```

Listing 8-4 modifies Listing 8-2 (page 80) to use the vertex buffer objects. The key difference in Listing 8-4 is that the parameters to the `glVertexAttribPointer` functions no longer point to the vertex arrays. Instead, each is an offset into the vertex buffer object.

Listing 8-4 Drawing with a vertex buffer object

```
void DrawModelUsingVertexBuffers()
{
    glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
    glVertexAttribPointer(GLKVertexAttribPosition, 2, GL_FLOAT, GL_FALSE,
        sizeof(vertexStruct), (void *)offsetof(vertexStruct, position));
    glEnableVertexAttribArray(GLKVertexAttribPosition);
    glVertexAttribPointer(GLKVertexAttribColor, 4, GL_UNSIGNED_BYTE, GL_TRUE,
```

```
        sizeof(vertexStruct), (void *)offsetof(vertexStruct, color));
    glEnableVertexAttribArray(GLKVertexAttribColor);


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

Buffer Usage Hints

The previous example initialized the vertex buffer once and never changed its contents afterwards. You can change the contents of a vertex buffer. A key part of the design of vertex buffer objects is that the app can inform OpenGL ES how it uses the data stored in the buffer. An OpenGL ES implementation can use this hint to alter the strategy it uses for storing the vertex data. In Listing 8-3 (page 81), each call to the `glBufferData` function provides a usage hint as the last parameter. Passing `GL_STATIC_DRAW` into `glBufferData` tells OpenGL ES that the contents of both buffers are never expected to change, which gives OpenGL ES more opportunities to optimize how and where the data is stored.

The OpenGL ES specification defines the following usage cases:

- `GL_STATIC_DRAW` is for vertex buffers that are rendered many times, and whose contents are specified once and never change.
- `GL_DYNAMIC_DRAW` is for vertex buffers that are rendered many times, and whose contents change during the rendering loop.
- `GL_STREAM_DRAW` is for vertex buffers that are rendered a small number of times and then discarded.

In iOS, `GL_DYNAMIC_DRAW` and `GL_STREAM_DRAW` are equivalent. You can use the `glBufferSubData` function to update buffer contents, but doing so incurs a performance penalty because it flushes the command buffer and waits for all commands to complete. Double or triple buffering can reduce this performance cost somewhat. (See "Use Double Buffering to Avoid Resource Conflicts" (page 59).) For better performance, use the `glMapBufferRange` function in OpenGL ES 3.0 or the corresponding function provided by the `EXT_map_buffer_range` extension in OpenGL ES 2.0 or 1.1.

If different attributes inside your vertex format require different usage patterns, split the vertex data into multiple structures and allocate a separate vertex buffer object for each collection of attributes that share common usage characteristics. Listing 8-5 modifies the previous example to use a separate buffer to hold the color data. By allocating the color buffer using the `GL_DYNAMIC_DRAW` hint, OpenGL ES can allocate that buffer so that your app maintains reasonable performance.

Listing 8-5 Drawing a model with multiple vertex buffer objects

```
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()
{
// Static position data
    glGenBuffers(1, &staticBuffer);
    glBindBuffer(GL_ARRAY_BUFFER, staticBuffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(staticVertexData), staticVertexData,
GL_STATIC_DRAW);

// Dynamic color data
// While not shown here, the expectation is that the data in this buffer changes
between frames.
    glGenBuffers(1, &dynamicBuffer);
    glBindBuffer(GL_ARRAY_BUFFER, dynamicBuffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(dynamicVertexData), dynamicVertexData,
GL_DYNAMIC_DRAW);
```

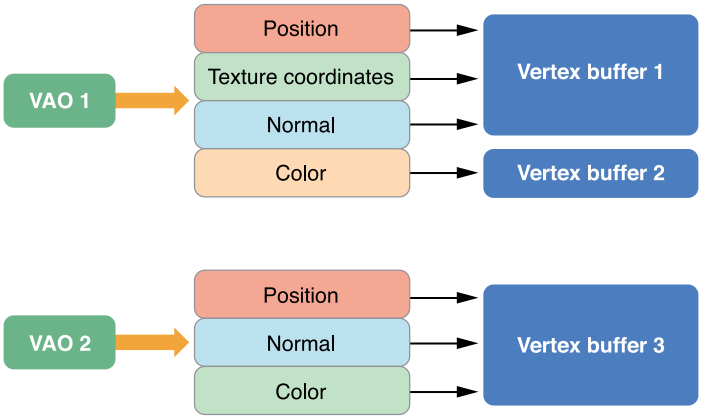
```
// Static index data
    glGenBuffers(1, &indexBuffer);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);
   glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
}

void DrawModelUsingMultipleVertexBuffers()
{
    glBindBuffer(GL_ARRAY_BUFFER, staticBuffer);
    glVertexAttribPointer(GLKVertexAttribPosition, 2, GL_FLOAT, GL_FALSE,
        sizeof(vertexStruct), (void *)offsetof(vertexStruct, position));
    glEnableVertexAttribArray(GLKVertexAttribPosition);

    glBindBuffer(GL_ARRAY_BUFFER, dynamicBuffer);
    glVertexAttribPointer(GLKVertexAttribColor, 4, GL_UNSIGNED_BYTE, GL_TRUE,
        sizeof(vertexStruct), (void *)offsetof(vertexStruct, color));
    glEnableVertexAttribArray(GLKVertexAttribColor);

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

Consolidate Vertex Array State Changes Using Vertex Array Objects

Take a closer look at the `DrawModelUsingMultipleVertexBuffers` function in Listing 8-5 (page 83). It enables many attributes, binds multiple vertex buffer objects, and configures attributes to point into the buffers. All of that initialization code is essentially static; none of the parameters change from frame to frame. If this function is called every time the app renders a frame, there's a lot of unnecessary overhead reconfiguring the graphics pipeline. If the app draws many different kinds of models, reconfiguring the pipeline may become a bottleneck. Instead, use a vertex array object to store a complete attribute configuration. Vertex array objects are part of the core OpenGL ES 3.0 specification and are available in OpenGL ES 2.0 and 1.1 through the `OES_vertex_array_object` extension.

Figure 8-7 shows an example configuration with two vertex array objects. Each configuration is independent of the other; each vertex array object can reference a different set of vertex attributes, which can be stored in the same vertex buffer object or split across several vertex buffer objects.

Figure 8-7 Vertex array object configuration



Listing 8-6 provides the code used to configure first vertex array object shown above. It generates an identifier for the new vertex array object and then binds the vertex array object to the context. After this, it makes the same calls to configure vertex attributes as it would if the code were not using vertex array objects. The configuration is stored to the bound vertex array object instead of to the context.

Listing 8-6 Configuring a vertex array object

```
void ConfigureVertexArrayObject()
{
    // Create and bind the vertex array object.
    glGenVertexArrays(1,&vao1);
    glBindVertexArray(vao1);
         // Configure the attributes in the VAO.
    glBindBuffer(GL_ARRAY_BUFFER, vbo1);
    glVertexAttribPointer(GLKVertexAttribPosition, 3, GL_FLOAT, GL_FALSE,
        sizeof(staticFmt), (void*)offsetof(staticFmt,position));
    glEnableVertexAttribArray(GLKVertexAttribPosition);
   glVertexAttribPointer(GLKVertexAttribTexCoord0, 2, GL_UNSIGNED_SHORT, GL_TRUE,
        sizeof(staticFmt), (void*)offsetof(staticFmt,texcoord));
    glEnableVertexAttribArray(GLKVertexAttribTexCoord0);
    glVertexAttribPointer(GLKVertexAttribNormal, 3, GL_FLOAT, GL_FALSE,
        sizeof(staticFmt), (void*)offsetof(staticFmt,normal));
```

```
    glEnableVertexAttribArray(GLKVertexAttribNormal);

    glBindBuffer(GL_ARRAY_BUFFER, vbo2);
    glVertexAttribPointer(GLKVertexAttribColor, 4, GL_UNSIGNED_BYTE, GL_TRUE,
        sizeof(dynamicFmt), (void*)offsetof(dynamicFmt,color));
    glEnableVertexAttribArray(GLKVertexAttribColor);

    // Bind back to the default state.
    glBindBuffer(GL_ARRAY_BUFFER,0);
    glBindVertexArray(0); }
```

To draw, the code binds the vertex array object and then submits drawing commands as before.

> **Note:** In OpenGL ES 3.0, client storage of vertex array data is not allowed—vertex array objects must use vertex buffer objects.

For best performance, your app should configure each vertex array object once, and never change it at runtime. If you need to change a vertex array object in every frame, create multiple vertex array objects instead. For example, an app that uses double buffering might configure one set of vertex array objects for odd-numbered frames, and a second set for even numbered frames. Each set of vertex array objects would point at the vertex buffer objects used to render that frame. When a vertex array object's configuration does not change, OpenGL ES can cache information about the vertex format and improve how it processes those vertex attributes.

Map Buffers into Client Memory for Fast Updates

One of the more challenging problems in OpenGL ES app design is working with dynamic resources, especially if your vertex data needs to change every frame. Efficiently balancing parallelism between the CPU and GPU requires carefully managing data transfers between your app's memory space and OpenGL ES memory. Traditional techniques, such as using the `glBufferSubData` function, can reduce performance because they force the GPU to wait while data is transferred, even if it could otherwise be rendering from data elsewhere in the same buffer.

For example, you may want to both modify a vertex buffer and draw its contents on each pass through a high frame rate rendering loop. A draw command from the last frame rendered may still be utilizing the GPU while the CPU is attempting to access buffer memory to prepare for drawing the next frame—causing the buffer update call to block further CPU work until the GPU is done. You can improve performance in such scenarios by manually synchronizing CPU and GPU access to a buffer.

The `glMapBufferRange` function provides a more efficient way to dynamically update vertex buffers. (This function is available as core API in OpenGL ES 3.0 and through the `EXT_map_buffer_range` extension in OpenGL ES 1.1 and 2.0.) Use this function to retrieve a pointer to a region of OpenGL ES memory, which you can then use to write new data. The `glMapBufferRange` function allows mapping of any subrange of the buffer's data storage into client memory. It also supports hints that allow for asynchronous buffer modification when you use the function together with a OpenGL sync object, as shown in Listing 8-7.

Listing 8-7 Dynamically updating a vertex buffer with manual synchronization

```
GLsync fence;
GLboolean UpdateAndDraw(GLuint vbo, GLuint offset, GLuint length, void *data) {
    GLboolean success;

    // Bind and map buffer.
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    void *old_data = glMapBufferRange(GL_ARRAY_BUFFER, offset, length,
        GL_MAP_WRITE_BIT | GL_MAP_FLUSH_EXPLICIT_BIT |
        GL_MAP_UNSYNCHRONIZED_BIT );

    // Wait for fence (set below) before modifying buffer.
    glClientWaitSync(fence, GL_SYNC_FLUSH_COMMANDS_BIT,
        GL_TIMEOUT_IGNORED);

    // Modify buffer, flush, and unmap.
    memcpy(old_data, data, length);
    glFlushMappedBufferRange(GL_ARRAY_BUFFER, offset, length);
    success = glUnmapBuffer(GL_ARRAY_BUFFER);

    // Issue other OpenGL ES commands that use other ranges of the VBO's data.

    // Issue draw commands that use this range of the VBO's data.
    DrawMyVBO(vbo);
```

```
    // Create a fence that the next frame will wait for.
    fence = glFenceSync(GL_SYNC_GPU_COMMANDS_COMPLETE, 0);
    return success;
}
```

The `UpdateAndDraw` function in this example uses the `glFenceSync` function to establish a synchronization point, or fence, immediately after submitting drawing commands that use a particular buffer object. It then uses the glClientWaitSync function (on the next pass through the rendering loop) to check that synchronization point before modifying the buffer object. If those drawing commands finish executing on the GPU before the rendering loop comes back around, CPU execution does not block and the `UpdateAndDraw` function continues to modify the buffer and draw the next frame. If the GPU has not finished executing those commands, the `glClientWaitSync` function blocks further CPU execution until the GPU reaches the fence. By manually placing synchronization points only around the sections of your code with potential resource conflicts, you can minimize how long the CPU waits for the GPU.

Best Practices for Working with Texture Data

Texture data is often the largest portion of the data your app uses to render a frame; textures provide the detail required to present great images to the user. To get the best possible performance out of your app, manage your app's textures carefully. To summarize the guidelines:

- Create your textures when your app is initialized, and never change them in the rendering loop.
- Reduce the amount of memory your textures use.
- Combine smaller textures into a larger texture atlas.
- Use mipmaps to reduce the bandwidth required to fetch texture data.
- Use multitexturing to perform texturing operations in a single pass.

Load Textures During Initialization

Creating and loading textures is an expensive operation. For best results, avoid creating new textures while your app is running. Instead, create and load your texture data during initialization.

After you create a texture, avoid changing it except at the beginning or end of a frame. Currently, all iOS devices use a tile-based deferred renderer, making calls to the `glTexSubImage` and `glCopyTexSubImage` functions particularly expensive. See "Tile-Based Deferred Rendering" in *OpenGL ES Hardware Platform Guide for iOS* for more information.

Use the GLKit Framework to Load Texture Data

Loading texture data is a fundamental operation that is important to get right. Using the GLKit framework, the `GLKTextureLoader` class makes creating and loading new textures easy. The `GLKTextureLoader` class can load texture data from a variety of sources, including files, URLs, in-memory representations, and CGImages. Regardless of the input source, the `GLKTextureLoader` class creates and loads a new texture from data and returns the texture information as a `GLKTextureInfo` object. Properties of `GLKTextureInfo` objects can be accessed to perform various tasks, including binding the texture to a context and enabling it for drawing.

> **Note:** A `GLKTextureInfo` object does not own the OpenGL ES texture object it describes. You must call the `glDeleteTextures` function to dispose of texture objects when you are done using them.

Listing 9-1 presents a typical strategy to load a new texture from a file and to bind and enable the texture for later use.

Listing 9-1 Loading a two-dimensional texture from a file

```
GLKTextureInfo *spriteTexture;
NSError *theError;


NSString *filePath = [[NSBundle mainBundle] pathForResource:@"Sprite" ofType:@"png"];
 // 1


spriteTexture = [GLKTextureLoader textureWithContentsOfFile:filePath options:nil
error:&theError]; // 2
glBindTexture(spriteTexture.target, spriteTexture.name); // 3
glEnable(spriteTexture.target); // 4
```

Here is what the code does, corresponding to the numbered steps in the listing:

1. Create a path to the image that contains the texture data. This path is passed as a parameter to the `GLKTextureLoader` class method `textureWithContentsOfFile:options:error:`.
2. Load a new texture from the image file and store the texture information in a `GLKTextureInfo` object. There are a variety of texture loading options available. For more information, see *GLKTextureLoader Class Reference*.
3. Bind the texture to a context, using the appropriate properties of the `GLKTextureInfo` object as parameters.
4. Enable use of the texture for drawing using the appropriate property of the `GLKTextureInfo` object as a parameter.

The `GLKTextureLoader` class can also load cubemap textures in most common image formats. And, if your app needs to load and create new textures while running, the `GLKTextureLoader` class also provides methods for asynchronous texture loading. See *GLKTextureLoader Class Reference* for more information.

Reduce Texture Memory Usage

Reducing the amount of memory your iOS app uses is always an important part of tuning your app. That said, an OpenGL ES app is also constrained in the total amount of memory it can use to load textures. Where possible, your app should always try to reduce the amount of memory it uses to hold texture data. Reducing the memory used by a texture is almost always at the cost of image quality, so you must balance any changes your app makes to its textures with the quality level of the final rendered frame. For best results, try the techniques described below, and choose the one that provides the best memory savings at an acceptable quality level.

Compress Textures

Texture compression usually provides the best balance of memory savings and quality. OpenGL ES for iOS supports multiple compressed texture formats.

All iOS devices support the the PowerVR Texture Compression (PVRTC) format by implementing the GL_IMG_texture_compression_pvrtc extension. There are two levels of PVRTC compression, 4 bits per channel and 2 bits per channel, which offer a 8:1 and 16:1 compression ratio over the uncompressed 32-bit texture format respectively. A compressed PVRTC texture still provides a decent level of quality, particularly at the 4-bit level. For more information on compressing textures into PVRTC format, see "Using texturetool to Compress Textures" (page 131).

OpenGL ES 3.0 also supports the ETC2 and EAC compressed texture formats; however, PVRTC textures are recommended on iOS devices.

Use Lower-Precision Color Formats

If your app cannot use compressed textures, consider using a lower precision pixel format. A texture in RGB565, RGBA5551, or RGBA4444 format uses half the memory of a texture in RGBA8888 format. Use RGBA8888 only when your app needs that level of quality.

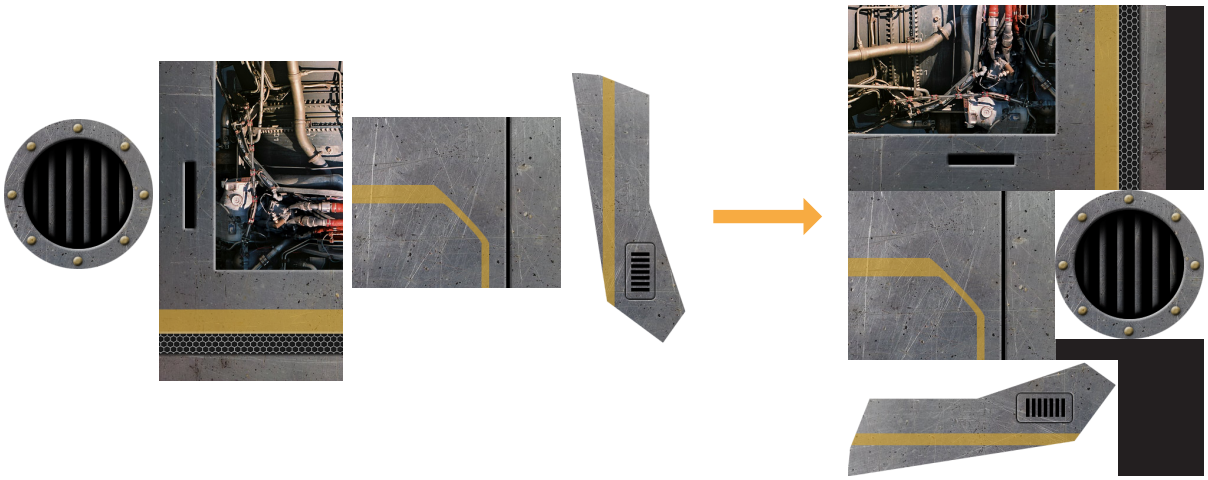
Use Properly Sized Textures

The images that an iOS-based device displays are very small. Your app does not need to provide large textures to present acceptable images to the screen. Halving both dimensions of a texture reduces the amount of memory needed for that texture to one-quarter that of the original texture.

Before shrinking your textures, attempt to compress the texture or use a lower-precision color format first. A texture compressed with the PVRTC format usually provides higher image quality than shrinking the texture—and it uses less memory too!

Combine Textures into Texture Atlases

Binding to a texture takes time for OpenGL ES to process. Apps that reduce the number of changes they make to OpenGL ES state perform better. For textures, one way to avoid binding to new textures is to combine multiple smaller textures into a single large texture, known as a **texture atlas**. When you use a texture atlas, you can bind a single texture and then make multiple drawing calls that use that texture, or even coalesce multiple drawing calls into a single draw call. The texture coordinates provided in your vertex data are modified to select the smaller portion of the texture from within the atlas.



Texture atlases have a few restrictions:

- You cannot use a texture atlas if you are using the `GL_REPEAT` texture wrap parameter.
- Filtering may sometimes fetch texels outside the expected range. To use those textures in a texture atlas, you must place padding between the textures that make up the texture atlas.
- Because the texture atlas is still a texture, it is subject to the OpenGL ES implementation's maximum texture size as well as other texture attributes.

Xcode 5 can automatically build texture atlases for you from a collection of images. For details on creating a texture atlas, see Texture Atlas Help. This feature is provided primarily for developers using the Sprite Kit framework, but any app can make use of the texture atlas files it produces. For each `.atlas` folder in your project, Xcode creates a `.atlasc` folder in your app bundle, containing one or more compiled atlas images and a property list (.plist) file. The property list file describes the individual images that make up the atlas and their locations within the atlas image—you can use this information to calculate appropriate texture coordinates for use in OpenGL ES drawing.

Use Mipmapping to Reduce Memory Bandwidth Usage

Your app should provide mipmaps for all textures except those being used to draw 2D unscaled images. Although mipmaps use additional memory, they prevent texturing artifacts and improve image quality. More importantly, when the smaller mipmaps are sampled, fewer texels are fetched from texture memory which reduces the memory bandwidth needed by the graphics hardware, improving performance.

The `GL_LINEAR_MIPMAP_LINEAR` filter mode provides the best quality when texturing but requires additional texels to be fetched from memory. Your app can trade some image quality for better performance by specifying the `GL_LINEAR_MIPMAP_NEAREST` filter mode instead.

When combining mip maps with texture atlases, use the `TEXTURE_MAX_LEVEL` parameter in OpenGL ES 3.0 to control how your textures are filtered. (This functionality is also available in OpenGL ES 1.1 and 2.0 through the `APPLE_texture_max_level` extension.)

Use Multitexturing Instead of Multiple Passes

Many apps perform multiple passes to draw a model, altering the configuration of the graphics pipeline for each pass. This not only requires additional time to reconfigure the graphics pipeline, but it also requires vertex information to be reprocessed for every pass, and pixel data to be read back from the framebuffer on later passes.

All OpenGL ES implementations on iOS support at least two texture units, and most devices support at least eight. Your app should use these texture units to perform as many steps as possible in your algorithm in each pass. You can retrieve the number of texture units available to your app by calling the `glGetIntegerv` function, passing in `GL_MAX_TEXTURE_UNITS` as the parameter.

If your app requires multiple passes to render a single object:

- Ensure that the position data remains unchanged for every pass.
- On the second and later stage, test for pixels that are on the surface of your model by calling the `glDepthFunc` function with `GL_EQUAL` as the parameter.

Best Practices for Shaders

Shaders provide great flexibility, but they can also be a significant bottleneck if you perform too many calculations or perform them inefficiently.

Compile and Link Shaders During Initialization

Creating a shader program is an expensive operation compared to other OpenGL ES state changes. Compile, link, and validate your programs when your app is initialized. Once you've created all your shaders, the app can efficiently switch between them by calling `glUseProgram`.

Check for Shader Program Errors When Debugging

Reading diagnostic information after compiling or linking a shader program is not necessary in a Release build of your app and can reduce performance. Use OpenGL ES functions to read shader compile or link logs only in development builds of your app, as shown in Listing 10-1.

Listing 10-1 Read shader compile/link logs only in development builds

```
// After calling glCompileShader, glLinkProgram, or similar

#ifdef DEBUG
// 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);
}
#endif
```

Similarly, you should call the `glValidateProgram` function only in development builds. You can use this function to find development errors such as failing to bind all texture units required by a shader program. But because validating a program checks it against the entire OpenGL ES context state, it is an expensive operation. Since the results of program validation are only meaningful during development, you should not call this function in Release builds of your app.

Use Separate Shader Objects to Speed Compilation and Linking

Many OpenGL ES apps use several vertex and fragment shaders, and it is often useful to reuse the same fragment shader with different vertex shaders or vice versa. Because the core OpenGL ES specification requires a vertex and fragment shader to be linked together in a single shader program, mixing and matching shaders results in a large number of programs, increasing the total shader compile and link time when you initialize your app.

OpenGL ES 2.0 and 3.0 contexts on iOS support the `EXT_separate_shader_objects` extension. You can use the functions provided by this extension to compile vertex and fragment shaders separately, and to mix and match precompiled shader stages at render time using program pipeline objects. Additionally, this extension provides a simplified interface for compiling and using shaders, shown in Listing 10-2.

Listing 10-2 Compiling and using separate shader objects

```
- (void)loadShaders

{

    const GLchar *vertexSourceText = " ... vertex shader GLSL source code ... ";

    const GLchar *fragmentSourceText = " ... fragment shader GLSL source code ...
 ";


    // Compile and link the separate vertex shader program, then read its uniform
 variable locations

    _vertexProgram = glCreateShaderProgramvEXT(GL_VERTEX_SHADER, 1,
&vertexSourceText);

    _uniformModelViewProjectionMatrix = glGetUniformLocation(_vertexProgram,
"modelViewProjectionMatrix");

    _uniformNormalMatrix = glGetUniformLocation(_vertexProgram, "normalMatrix");


   // Compile and link the separate fragment shader program (which uses no uniform
 variables)

    _fragmentProgram =  glCreateShaderProgramvEXT(GL_FRAGMENT_SHADER, 1,
&fragmentSourceText);
```

```
    // Construct a program pipeline object and configure it to use the shaders
    glGenProgramPipelinesEXT(1, &_ppo);
    glBindProgramPipelineEXT(_ppo);
    glUseProgramStagesEXT(_ppo, GL_VERTEX_SHADER_BIT_EXT, _vertexProgram);
    glUseProgramStagesEXT(_ppo, GL_FRAGMENT_SHADER_BIT_EXT, _fragmentProgram);
}

- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect
{
    // Clear the framebuffer
    glClearColor(0.65f, 0.65f, 0.65f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Use the previously constructed program pipeline and set uniform contents
in shader programs
    glBindProgramPipelineEXT(_ppo);
   glProgramUniformMatrix4fvEXT(_vertexProgram, _uniformModelViewProjectionMatrix,
 1, 0, _modelViewProjectionMatrix.m);
    glProgramUniformMatrix3fvEXT(_vertexProgram, _uniformNormalMatrix, 1, 0,
_normalMatrix.m);

    // Bind a VAO and render its contents
    glBindVertexArrayOES(_vertexArray);
    glDrawElements(GL_TRIANGLE_STRIP, _indexCount, GL_UNSIGNED_SHORT, 0);
}
```

Respect the Hardware Limits on Shaders

OpenGL ES limits the number of each variable type you can use in a vertex or fragment shader. The OpenGL ES specification doesn't require implementations to provide a software fallback when these limits are exceeded; instead, the shader simply fails to compile or link. When developing your app you must ensure that no errors occur during shader compilation, as shown in Listing 10-1.

Use Precision Hints

Precision hints were added to the GLSL ES language specification to address the need for compact shader variables that match the smaller hardware limits of embedded devices. Each shader must specify a default precision; individual shader variables may override this precision to provide hints to the compiler on how that variable is used in your app. An OpenGL ES implementation is not required to use the hint information, but may do so to generate more efficient shaders. The GLSL ES specification lists the range and precision for each hint.

Important: The range limits defined by the precision hints are not enforced. You cannot assume your data is clamped to this range.

Follow these guidelines:

- When in doubt, default to high precision.
- Colors in the `0.0` to `1.0` range can usually be represented using low precision variables.
- Position data should usually be stored as high precision.
- Normals and vectors used in lighting calculations can usually be stored as medium precision.
- After reducing precision, retest your app to ensure that the results are what you expect.

Listing 10-3 defaults to high precision variables, but calculates the color output using low precision variables because higher precision is not necessary.

Listing 10-3 Low precision is acceptable for fragment color

```
precision highp float; // Defines precision for float and float-derived
(vector/matrix) types.
uniform lowp sampler2D sampler; // Texture2D() result is lowp.
varying lowp vec4 color;
varying vec2 texCoord;   // Uses default highp precision.


void main()
{
    gl_FragColor = color * texture2D(sampler, texCoord);
}
```

The actual precision of shader variables can vary between different iOS devices, as can the performance of operations at each level of precision. Refer to the *iOS Device Compatibility Reference* for device-specific considerations.

Perform Vector Calculations Lazily

Not all graphics processors include vector processors; they may perform vector calculations on a scalar processor. When performing calculations in your shader, consider the order of operations to ensure that the calculations are performed efficiently even if they are performed on a scalar processor.

If the code in Listing 10-4 were executed on a vector processor, each multiplication would be executed in parallel across all four of the vector's components. However, because of the location of the parenthesis, the same operation on a scalar processor would take eight multiplications, even though two of the three parameters are scalar values.

Listing 10-4 Poor use of vector operators

```
highp float f0, f1;
highp vec4 v0, v1;
v0 = (v1 * f0) * f1;
```

The same calculation can be performed more efficiently by shifting the parentheses as shown in Listing 10-5. In this example, the scalar values are multiplied together first, and the result multiplied against the vector parameter; the entire operation can be calculated with five multiplications.

Listing 10-5 Proper use of vector operations

```
highp float f0, f1;
highp vec4 v0, v1;
v0 = v1 * (f0 * f1);
```

Similarly, your app should always specify a write mask for a vector operation if it does not use all of the components of the result. On a scalar processor, calculations for components not specified in the mask can be skipped. Listing 10-6 runs twice as fast on a scalar processor because it specifies that only two components are needed.

Listing 10-6 Specifying a write mask

```
highp vec4 v0;
highp vec4 v1;
highp vec4 v2;
v2.xz = v0 * v1;
```

Use Uniforms or Constants Instead of Computing Values in a Shader

Whenever a value can be calculated outside the shader, pass it into the shader as a uniform or a constant. Recalculating dynamic values can potentially be very expensive in a shader.

Use Branching Instructions with Caution

Branches are discouraged in shaders, as they can reduce the ability to execute operations in parallel on 3D graphics processors (although this performance cost is reduced on OpenGL ES 3.0–capable devices).

Your app may perform best if you avoid branching entirely. For example, instead of creating a large shader with many conditional options, create smaller shaders specialized for specific rendering tasks. There is a tradeoff between reducing the number of branches in your shaders and increasing the number of shaders you create. Test different options and choose the fastest solution.

If your shaders must use branches, follow these recommendations:

- Best performance: Branch on a constant known when the shader is compiled.
- Acceptable: Branch on a uniform variable.
- Potentially slow: Branch on a value computed inside the shader.

Eliminate Loops

You can eliminate many loops by either unrolling the loop or using vectors to perform operations. For example, this code is very inefficient:

```
int i;
float f;
vec4 v;

for(i = 0; i < 4; i++)
    v[i] += f;
```

The same operation can be done directly using a component-wise add:

```
float f;
vec4 v;
v += f;
```

When you cannot eliminate a loop, it is preferred that the loop have a constant limit to avoid dynamic branches.

Avoid Computing Array Indices in Shaders

Using indices computed in the shader is more expensive than a constant or uniform array index. Accessing uniform arrays is usually cheaper than accessing temporary arrays.

Be Aware of Dynamic Texture Lookups

Dynamic texture lookups, also known as *dependent texture reads*, occur when a fragment shader computes texture coordinates rather than using the unmodified texture coordinates passed into the shader. Dependent texture reads are supported at no performance cost on OpenGL ES 3.0–capable hardware; on other devices, dependent texture reads can delay loading of texel data, reducing performance. When a shader has no dependent texture reads, the graphics hardware may prefetch texel data before the shader executes, hiding some of the latency of accessing memory.

Listing 10-7 shows a fragment shader that calculates new texture coordinates. The calculation in this example can easily be performed in the vertex shader, instead. By moving the calculation to the vertex shader and directly using the vertex shader's computed texture coordinates, you avoid the dependent texture read.

> **Note:** It may not seem obvious, but any calculation on the texture coordinates counts as a dependent texture read. For example, packing multiple sets of texture coordinates into a single varying parameter and using a swizzle command to extract the coordinates still causes a dependent texture read.

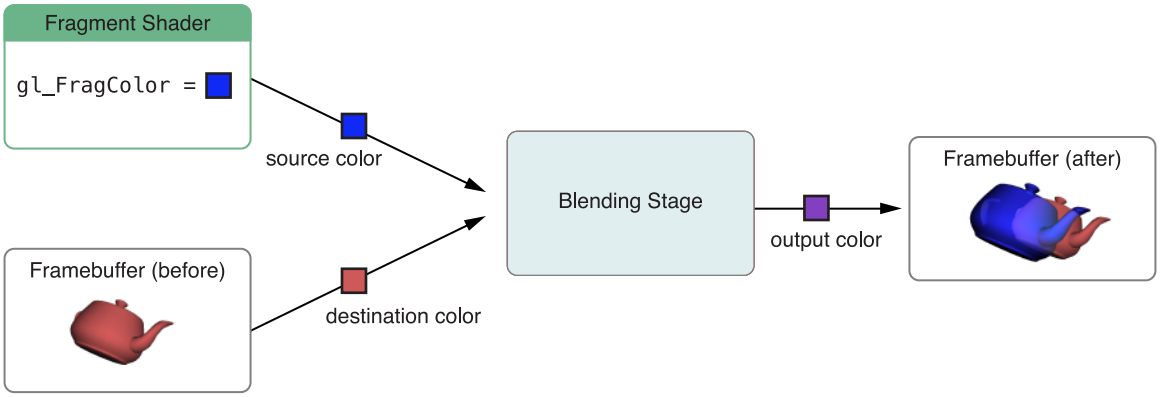
Listing 10-7 Dependent Texture Read

```
varying vec2 vTexCoord;
uniform sampler2D textureSampler;

void main()
{
    vec2 modifiedTexCoord = vec2(1.0 - vTexCoord.x, 1.0 - vTexCoord.y);
    gl_FragColor = texture2D(textureSampler, modifiedTexCoord);
}
```

Fetch Framebuffer Data for Programmable Blending

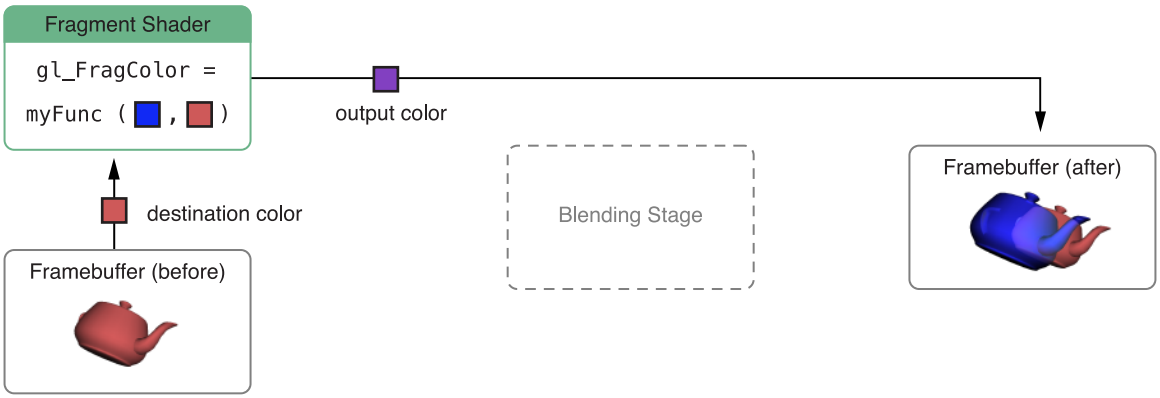
Traditional OpenGL and OpenGL ES implementations provide a fixed-function blending stage, illustrated in Figure 10-1. Before issuing a draw call, you specify a blending operation from a fixed set of possible parameters. After your fragment shader outputs color data for a pixel, the OpenGL ES blending stage reads color data for the corresponding pixel in the destination framebuffer, then combines the two according to the specified blending operation to produce an output color.

Figure 10-1 Traditional fixed-function blending



In iOS 6.0 and later, you can use the `EXT_shader_framebuffer_fetch` extension to implement programmable blending and other effects. Instead of supplying a source color to be blended by OpenGL ES, your fragment shader reads the contents of the destination framebuffer corresponding to the fragment being processed. Your fragment shader can then use whatever algorithm you choose to produce an output color, as shown in Figure 10-2.

Figure 10-2 Programmable blending with framebuffer fetch



This extension enables many advanced rendering techniques:

- **Additional blending modes.** By defining your own GLSL ES functions for combining source and destination colors, you can implement blending modes not possible with the OpenGL ES fixed-function blending stage. For example, Listing 10-8 (page 102) implements the Overlay and Difference blending modes found in popular graphics software.
- **Post-processing effects.** After rendering a scene, you can draw a full-screen quad using a fragment shader that reads the current fragment color and transforms it to produce an output color. The shader in Listing 10-9 (page 103) can be used with this technique to convert a scene to grayscale.
- **Non-color fragment operations.** Framebuffers may contain non-color data. For example, deferred shading algorithms use multiple render targets to store depth and normal information. Your fragment shader can read such data from one (or more) render targets and use them to produce an output color in another render target.

These effects are possible without the framebuffer fetch extension—for example, grayscale conversion can be done by rendering a scene into a texture, then drawing a full-screen quad using that texture and a fragment shader that converts texel colors to grayscale. However, using this extension generally results in better performance.

To enable this feature, your fragment shader must declare that it requires the `EXT_shader_framebuffer_fetch` extension, as shown in Listing 10-8 and Listing 10-9. The shader code to implement this feature differs between versions of the OpenGL ES Shading Language (GLSL ES).

Using Framebuffer Fetch in GLSL ES 1.0

For OpenGL ES 2.0 contexts and OpenGL ES 3.0 contexts not using `#version 300 es` shaders, you use the `gl_FragColor` builtin variable for fragment shader output and the `gl_LastFragData` builtin variable to read framebuffer data, as illustrated in Listing 10-8.

Listing 10-8 Fragment shader for programmable blending in GLSL ES 1.0

```
#extension GL_EXT_shader_framebuffer_fetch : require

#define kBlendModeDifference 1
#define kBlendModeOverlay    2
#define BlendOverlay(a, b) ( (b<0.5) ? (2.0*b*a) : (1.0-2.0*(1.0-a)*(1.0-b)) )

uniform int blendMode;
varying lowp vec4 sourceColor;
```

```
void main()
{
    lowp vec4 destColor = gl_LastFragData[0];
    if (blendMode == kBlendModeDifference) {
        gl_FragColor = abs( destColor - sourceColor );
    } else if (blendMode == kBlendModeOverlay) {
        gl_FragColor.r = BlendOverlay(sourceColor.r, destColor.r);
        gl_FragColor.g = BlendOverlay(sourceColor.g, destColor.g);
        gl_FragColor.b = BlendOverlay(sourceColor.b, destColor.b);
        gl_FragColor.a = sourceColor.a;
    } else { // normal blending
        gl_FragColor = sourceColor;
    }
}
```

Using Framebuffer Fetch in GLSL ES 3.0

In GLSL ES 3.0, you use user-defined variables declared with the `out` qualifier for fragment shader outputs. If you declare a fragment shader output variable with the `inout` qualifier, it will contain framebuffer data when the fragment shader executes. Listing 10-9 illustrates a grayscale post-processing technique using an `inout` variable.

Listing 10-9 Fragment shader for color post-processing in GLSL ES 3.0

```
#version 300 es
#extension GL_EXT_shader_framebuffer_fetch : require

layout(location = 0) inout lowp vec4 destColor;

void main()
{
    lowp float luminance = dot(vec3(0.3, 0.59, 0.11), destColor.rgb);
    destColor.rgb = vec3(luminance);
}
```

Use Textures for Larger Memory Buffers in Vertex Shaders

In iOS 7.0 and later, vertex shaders can read from currently bound texture units. Using this technique you can access much larger memory buffers during vertex processing, enabling high performance for some advanced rendering techniques. For example:

- **Displacement mapping.** Draw a mesh with default vertex positions, then read from a texture in the vertex shader to alter the position of each vertex. Listing 10-10 demonstrates using this technique to generate three-dimensional geometry from a grayscale height map texture.
- **Instanced drawing.** As described in "Use Instanced Drawing to Minimize Draw Calls" (page 70), instanced drawing can dramatically reduce CPU overhead when rendering a scene that contains many similar objects. However, providing per-instance information to the vertex shader can be a challenge. A texture can store extensive information for many instances. For example, you could render a vast cityscape by drawing hundreds of instances from vertex data describing only a simple cube. For each instance, the vertex shader could use the `gl_InstanceID` variable to sample from a texture, obtaining a transformation matrix, color variation, texture coordinate offset, and height variation to apply to each building.

Listing 10-10 Vertex shader for rendering from a height map

```
attribute vec2 xzPos;

uniform mat4 modelViewProjectionMatrix;
uniform sampler2D heightMap;

void main()
{
    // Use the vertex X and Z values to look up a Y value in the texture.
    vec4 position = texture2D(heightMap, xzPos);
    // Put the X and Z values into their places in the position vector.
    position.xz = xzPos;

    // Transform the position vector from model to clip space.
    gl_Position = modelViewProjectionMatrix * position;
}
```

You can also use uniform arrays and uniform buffer objects (in OpenGL ES 3.0) to provide bulk data to a vertex shader, but vertex texture access offers several potential advantages. You can store much more data in a texture than in either a uniform array or uniform buffer object, and you can use texture wrapping and filtering options to interpolate the data stored in a texture. Additionally, you can render to a texture, taking advantage of the GPU to produce data for use in a later vertex processing stage.

To determine whether vertex texture sampling is available on a device (and the number of texture units available to vertex shaders), check the value of the `MAX_VERTEX_TEXTURE_IMAGE_UNITS` limit at run time. (See "Verifying OpenGL ES Capabilities" (page 16).)

Concurrency and OpenGL ES

In computing, concurrency usually refers to executing tasks on more than one processor at the same time. By performing work in parallel, tasks complete sooner, and apps become more responsive to the user. A well-designed OpenGL ES app already exhibits a specific form of concurrency—concurrency between app processing on the CPU and OpenGL ES processing on the GPU. Many techniques introduced in "OpenGL ES Design Guidelines" (page 48) are aimed specifically at creating OpenGL apps that exhibit great CPU-GPU parallelism. Designing a concurrent app means decomposing the work 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 iOS consists of one or more threads. A **thread** is a stream of execution that runs code for the process. Apple offers both traditional threads and a feature called **Grand Central Dispatch (GCD)**. Using Grand Central Dispatch, you can decompose a task into subtasks without manually managing threads. GCD allocates threads based on the number of cores available on the device and automatically schedules tasks to those threads.

At a higher level, Cocoa Touch offers `NSOperation` and `NSOperationQueue` to provide an Objective-C abstraction for creating and scheduling units of work.

This chapter does not describe these technologies in detail. Before you consider how to add concurrency to your OpenGL ES app, consult *Concurrency Programming Guide*. If you plan to manage threads manually, also see *Threading Programming Guide*. Regardless of which technique you use, there are additional restrictions when calling OpenGL ES on multithreaded systems. This chapter helps you understand when multithreading improves your OpenGL ES app's performance, the restrictions OpenGL ES places on multithreaded app, and common design strategies you might use to implement concurrency in an OpenGL ES app.

Deciding Whether You Can Benefit from Concurrency

Creating a multithreaded app requires significant effort in the design, implementation, and testing of your app. Threads also add complexity and overhead. Your app 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 ES app, optimize your OpenGL ES code in a single-threaded environment using the techniques described in "OpenGL ES Design Guidelines" (page 48). Focus on achieving great CPU-GPU parallelism first and then assess whether concurrent programming can provide additional performance.

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

- The app performs many tasks on the CPU that are independent of OpenGL ES 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 ES drawing code.
- Profiling your app has shown that your OpenGL ES rendering code spends a lot of time in the CPU. In this scenario, the GPU is idle because your app 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 app is blocked waiting for the GPU, and has no work it can perform in parallel with its OpenGL ES drawing, then it is not a good candidate for concurrency. If the CPU and GPU are both idle, then your OpenGL ES needs are probably simple enough that no further tuning is needed.

OpenGL ES Restricts Each Context to a Single Thread

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

OpenGL ES is not reentrant. If you modify the same context from multiple threads simultaneously, the results are unpredictable. Your app 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 ES calls to the context. OpenGL ES commands that block—such as `glFinish`—do not synchronize threads.

GCD and `NSOperationQueue` objects can 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 ES app, that means:

- Each task must set the context before executing any OpenGL ES commands.
- Two tasks that access the same context may never execute simultaneously.
- Each task should clear the thread's context before exiting.

Strategies for Implementing Concurrency in OpenGL ES Apps

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

- Decompose your app into OpenGL ES and non-OpenGL ES tasks that can execute concurrently. Your OpenGL ES drawing code executes as a single task, so it still executes in a single thread. This strategy works best when your app has other tasks that require significant CPU processing.
- If performance profiling reveals that your application spends a lot of CPU time inside OpenGL, move some of that processing to another thread by enabling multithreading for your OpenGL ES context. The advantage is simplicity; enabling multithreading takes a single line of code. See "Multithreaded OpenGL ES" (page 108).
- If your app spends a lot of CPU time preparing data to send to OpenGL ES, divide the work between tasks that prepare rendering data and tasks that submit rendering commands to OpenGL ES. See "Perform OpenGL ES Computations in a Worker Task" (page 109)
- If your app has multiple scenes it can render simultaneously or work it can perform in multiple contexts, it can create multiple tasks, with one OpenGL ES context per task. If the contexts need access to the same art assets, use a sharegroup to share OpenGL ES objects between the contexts. See "Use Multiple OpenGL ES Contexts" (page 110).

Multithreaded OpenGL ES

Whenever your application calls an OpenGL ES function, OpenGL ES 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 overhead in preparing commands for the hardware.

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

To enable OpenGL ES multithreading, set the value of the `multiThreaded` property of your `EAGLContext` object to `YES`.

> **Note:** Enabling or disabling multithreaded execution causes OpenGL ES to flush previous commands and incurs the overhead of setting up the additional thread. Enable or disable multithreading in an initialization function rather than in the rendering loop.

Enabling multithreading means OpenGL ES must copy parameters to transmit them to the worker thread. Because of this overhead, always test your application with and without multithreading enabled to determine whether it provides a substantial performance improvement. You can minimize this overhead by implementing your own strategy for x OpenGL ES use in a multithreaded app, as described in the remainder of this chapter.

Perform OpenGL ES Computations in a Worker Task

Some app perform lots of calculations on their data before passing the data down to OpenGL ES. For example, the app might create new geometry or animate existing geometry. Where possible, such calculations should be performed inside OpenGL ES. This takes advantage of the greater parallelism available inside the GPU, and reduces the overhead of copying results between your app and OpenGL ES.

The approach described in Figure 6-6 (page 56) alternates between updating OpenGL ES objects and executing rendering commands that use those objects. OpenGL ES renders on the GPU in parallel with your app'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 ES rendering code into separate calculation and processing tasks, and run them in parallel. One task produces data that is consumed by the second and submitted to OpenGL.

For best performance, avoid copying data between tasks. 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 you can further decompose the modifications task into subtasks, you may see better benefits. For example, assume two or more vertex buffer objects, 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.

Use Multiple OpenGL ES Contexts

One common approach for using multiple contexts is to have one context that updates OpenGL ES objects while the other consumes those resources, with each context running on a separate thread. Because each context runs on a separate thread, its actions are rarely blocked by the other context. To implement this, your app would create two contexts and two threads; each thread controls one context. Further, any OpenGL ES objects your app intends to update on the second thread must be double buffered; a consuming thread may not access an OpenGL ES object while the other thread is modifying it. The process of synchronizing the changes between the contexts is described in detail in "An EAGL Sharegroup Manages OpenGL ES Objects for the Context" (page 21).

The `GLKTextureLoader` class implements this strategy to provide asynchronous loading of texture data. (See "Use the GLKit Framework to Load Texture Data" (page 89).)

Guidelines for Threading OpenGL ES Apps

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

- Use only one thread per context. OpenGL ES commands for a specific context are not thread safe. Never have more than one thread accessing a single context simultaneously.
- When using GCD, use a dedicated serial queue to dispatch commands to OpenGL ES; this can be used to replace the conventional mutex pattern.
- 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 and clear the current context before leaving the thread.

Adopting OpenGL ES 3.0

OpenGL ES 3.0 is a superset of the OpenGL ES 2.0 specification, so adopting it in your app is easy. You can continue to use your OpenGL ES 2.0 code while taking advantage of the higher resource limits available to OpenGL ES 3.0 contexts on compatible devices, and add support for OpenGL ES 3.0–specific features where it makes sense for your app's design.

Checklist for Adopting OpenGL ES 3.0

To use OpenGL ES 3.0 in your app:

1. Create an OpenGL ES context (as described in "Configuring OpenGL ES Contexts" (page 19)), and specify the API version constant for OpenGL ES 3.0:

   ```
   EAGLContext *context = [[EAGLContext alloc]
   initWithAPI:kEAGLRenderingAPIOpenGLES3];
   ```

   If you plan to make your app available for devices that do not support OpenGL ES 3.0, follow the procedure in Listing 2-1 (page 20) to fall back to OpenGL ES 2.0 when necessary.

2. Include or import the OpenGL ES 3.0 API headers in source files that use OpenGL ES 3.0 API:

   ```
   #import <OpenGLES/ES3/gl.h>
   #import <OpenGLES/ES3/glext.h>
   ```

3. Update code that uses OpenGL ES 2.0 extensions incorporated into or changed by the OpenGL ES 3.0 specifications, as described in "Updating Extension Code" below.

4. (Optional.) You can use the same shader programs in both OpenGL ES 2.0 and 3.0. However, if you choose to port shaders to GLSL ES 3.0 to use new features, see the caveats in "Adopting OpenGL ES Shading Language version 3.0" (page 114).

5. Test your app on an OpenGL ES 3.0–compatible device to verify that it behaves correctly.

Updating Extension Code

OpenGL ES 3.0 is a superset of the OpenGL ES 2.0 specification, so apps that use only core OpenGL ES 2.0 features can be used in an OpenGL ES 3.0 context without changes. However, some apps also use OpenGL ES 2.0 extensions. The features provided by these extensions are also available in OpenGL ES 3.0, but using them in an OpenGL ES 3.0 context may require at least minor code changes.

Remove Extension Suffixes

The OpenGL ES 2.0 extensions listed below define APIs that are incorporated into the core OpenGL ES 3.0 specification. To use these features in an OpenGL ES 3.0 context, simply remove the extension suffixes from function and constant names. For example, the name of the `glMapBufferRangeEXT` function becomes `glMapBufferRange`, and the `DEPTH_COMPONENT24_OES` constant (used in the `internalformat` parameter of the `glRenderbufferStorage` function) becomes `DEPTH_COMPONENT24`.

- `OES_depth24`
- `OES_element_index_uint`
- `OES_fbo_render_mipmap`
- `OES_rgb8_rgba8`
- `OES_texture_half_float_linear`
- `OES_vertex_array_object`
- `EXT_blend_minmax`
- `EXT_draw_instanced`
- `EXT_instanced_arrays`
- `EXT_map_buffer_range`
- `EXT_occlusion_query_boolean`
- `EXT_texture_storage`
- `APPLE_sync`
- `APPLE_texture_max_level`

Modify Use of Extension APIs

Some features defined by OpenGL ES 2.0 extensions are in the core OpenGL ES 3.0 specification, but with changes to their API definitions. To use these features in an OpenGL ES 3.0 context, make the changes described below.

Working with Texture Formats

The OES_depth_texture, OES_packed_depth_stencil, OES_texture_float, OES_texture_half_float, EXT_texture_rg, and EXT_sRGB extensions define constants for use in the internalformat and type parameters of the glTexImage family of functions. The functionality defined by these extensions is available in the OpenGL ES 3.0 core API, but with some caveats:

- The glTexImage functions do not support internalformat constants without explicit sizes. Use explicitly sized constants instead:

```
// Replace this OpenGL ES 2.0 code:
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA,
GL_HALF_FLOAT_OES, data);
// With this OpenGL ES 3.0 code:
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, width, height, 0, GL_RGBA,
GL_HALF_FLOAT, data);
```

- OpenGL ES 3.0 does not define float or half-float formats for LUMINANCE or LUMINANCE_ALPHA data. Use the corresponding RED or RG formats instead.
- The vector returned by depth and depth/stencil texture samplers no longer repeats the depth value in its first three components in OpenGL ES 3.0. Use only the first (.r) component in shader code that samples such textures.
- The sRGB format is only valid when used for the internalformat parameter in OpenGL ES 3.0. Use GL_RGB or GL_RGBA for the format parameter for sRGB textures.

Alternatively, replace calls to glTexImage functions with calls to the corresponding glTexStorage functions. Texture storage functions are available in as core API in OpenGL ES 3.0, and through the EXT_texture_storage extension in OpenGL ES 1.1 and 2.0. These functions offer an additional benefit: using a glTexStorage function completely specifies an immutable texture object in one call; it performs all consistency checks and memory allocations immediately, guaranteeing that the texture object can never be incomplete due to missing mipmap levels or inconsistent cube map faces.

Mapping Buffer Objects into Client Memory

The OES_mapbuffer extension defines the glMapBuffer function for mapping the entire data storage of a buffer object into client memory. OpenGL ES 3.0 instead defines the glMapBufferRange function, which provides additional functionality: it allows mapping a subset of a buffer object's data storage and includes options for asynchronous mapping. The glMapBufferRange function is also available in OpenGL ES 1.1 and 2.0 contexts through the EXT_map_buffer_range extension.

Discarding Framebuffers

The `glInvalidateFramebuffer` function in OpenGL ES 3.0 replaces the `glDiscardFramebufferEXT` function provided by the `EXT_discard_framebuffer` extension. The parameters and behavior of both functions are identical.

Using Multisampling

OpenGL ES 3.0 incorporates all features of the `APPLE_framebuffer_multisample` extension, except for the `glResolveMultisampleFramebufferAPPLE` function. Instead the `glBlitFramebuffer` function provides this and other other framebuffer copying options. To resolve a multisampling buffer, set the read and draw framebuffers (as in "Using Multisampling to Improve Image Quality" (page 40)) and then use `glBlitFramebuffer` to copy the entire read framebuffer into the entire draw framebuffer:

```
glBlitFramebuffer(0,0,w,h, 0,0,w,h, GL_COLOR_BUFFER_BIT, GL_NEAREST);
```

Continue Using Most Other Extensions in OpenGL ES 3.0

Several key features of iOS device graphics hardware are not part of the core OpenGL ES 3.0 specification, but remain available as OpenGL ES 3.0 extensions. To use these features, continue to check for extension support using the procedures described in "Verifying OpenGL ES Capabilities" (page 16). (See also the *iOS Device Compatibility Reference* to determine which features are available on which devices.)

Most code written for OpenGL ES 2.0 extensions that are also present as OpenGL ES 3.0 extensions will work in an OpenGL ES 3.0 context without changes. However, additional caveats apply to extensions which modify the vertex and fragment shader language—for details, see the next section.

Adopting OpenGL ES Shading Language version 3.0

OpenGL ES 3.0 includes a new version of the OpenGL ES Shading Language (GLSL ES). OpenGL ES 3.0 contexts can use shader programs written in either version 1.0 or version 3.0 of GLSL ES, but version 3.0 shaders (marked with a `#version 300 es` directive in shader source code) are required to access certain new features, such as uniform blocks, 32-bit integers and additional integer operations.

Some language conventions have changed between GLSL ES version 1.0 and 3.0. These changes make shader source code more portable between OpenGL ES 3.0 and desktop OpenGL ES 3.3 or later, but they also require minor changes to existing shader source code when porting to GLSL ES 3.0:

- The `attribute` and `varying` qualifiers are replaced in GLSL ES 3.0 by by the keywords `in` and `out`. In a vertex shader, use the `in` qualifier for vertex attributes and the `out` qualifier for varying outputs. In a fragment shader, use the `in` qualifier for varying inputs.

- GLSL ES 3.0 removes the `gl_FragData` and `gl_FragColor` builtin fragment output variables. Instead, you declare your own fragment output variables with the `out` qualifier.
- Texture sampling functions have been renamed in GLSL ES 3.0—all sampler types use the same texture function name. For example, you can use the new `texture` function with either a `sampler2D` or `samplerCube` parameter (replacing the texture2D and textureCube functions from GLSL ES 1.0).
- The features added to GLSL ES 1.0 by the `EXT_shader_texture_lod`, `EXT_shadow_samplers`, and `OES_standard_derivatives` extensions are part of the core GLSL ES specification. When porting shaders that use these features to GLSL ES 3.0, use the corresponding GLSL ES 3.0 functions.
- The `EXT_shader_framebuffer_fetch` extension works differently. GLSL ES 3.0 removes the `gl_FragData` and `gl_FragColor` builtin fragment output variables in favor of requiring fragment outputs to be declared in the shader. Correspondingly, the `gl_LastFragData` builtin variable is not present in GLSL ES 3.0 fragment shaders. Instead, any fragment output variables you declare with the `inout` qualifier contain previous fragment data when the shader runs. For more details, see "Fetch Framebuffer Data for Programmable Blending" (page 101).

For a complete overview of GLSL ES 3.0, see the *OpenGL ES Shading Language 3.0 Specification* , available from the OpenGL ES API Registry.

Xcode OpenGL ES Tools Overview

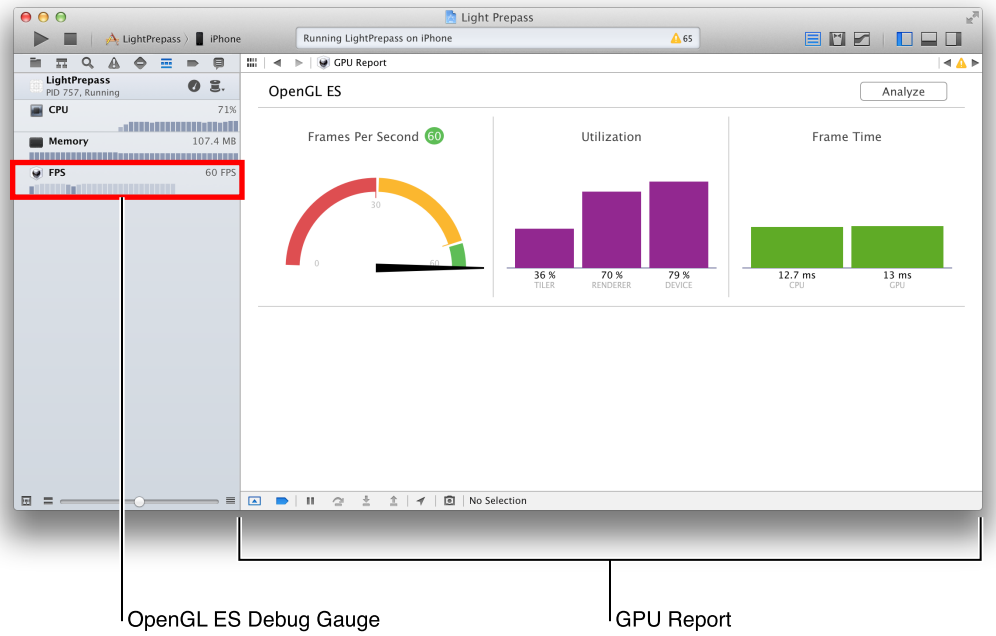
Xcode tools for debugging, analyzing, and tuning OpenGL ES applications are useful during all stages of development. The FPS Debug Gauge and GPU report summarize your app's GPU performance every time you run it from Xcode, so you can quickly spot performance issues while designing and building your renderer. Once you've found a trouble spot, capture a frame and use Xcode's OpenGL ES Frame Debugger interface to pinpoint rendering problems and solve performance issues.

Effectively using the Xcode OpenGL ES features requires some familiarity with Xcode's debugging interface. For background information, read *Xcode Overview*.

Using the FPS Debug Gauge and GPU Report

The FPS debug gauge and accompanying GPU report, shown in Figure B-1, provide a high-level summary of your app's OpenGL ES performance while it runs. By monitoring these displays when developing your app, you can discover performance issues as they arise and consider where to focus your tuning efforts.

Figure B-1 FPS Debug Gauge and GPU Report



> **Note:** Some features of the FPS gauge and GPU report rely on a display link timer. If you do not use the `CADisplayLink` or `GLKViewController` classes to animate your OpenGL ES displays, the gauge and report cannot show performance relative to a target frame rate or provide accurate CPU frame time information.

The debug gauge and report contain the following displays:

- **FPS Gauge.** Shows the current animation rate of your app, in frames per second (FPS), and a recent history of FPS readings. Click this gauge to display the GPU report in Xcode's primary editor.
- **Frames Per Second.** Shows the current frame rate, relative to the target frame rate set by your app (often 30 or 60 FPS). A blue arc indicates the recent range of FPS readings.
- **Utilization.** Shows three bars, breaking down your app's use of the different processing resources on the GPU and indicating the possible locations of performance bottlenecks in your use of graphics hardware.

  The Tiler bar measures use of the GPU's geometry processing resources. High tiler utilization can indicate performance bottlenecks in the vertex and primitive processing stages of the OpenGL ES pipeline, such as using inefficient vertex shader code or drawing an excessive number of vertices or primitives each frame.

  The Renderer bar measures use of the GPU's pixel processing resources. High renderer utilization can indicate performance bottlenecks in the fragment and pixel processing stages of the OpenGL ES pipeline, such as using inefficient fragment shader code or processing additional fragments each frame for color blending.

  The Device bar shows overall GPU usage, incorporating both tiler and renderer usage.
- **Frame Time.** Shows the time spent processing each frame on both the CPU and GPU. This graph can indicate whether your app makes effective use of CPU/GPU parallelism.

  If your app spends more time in CPU processing, you may be able to improve performance by moving work to the GPU. For example, if each frame requires many similar `glDrawArrays` or `glDrawElements` calls, you can use hardware instancing to reduce CPU overhead. (For details, see "Use Instanced Drawing to Minimize Draw Calls" (page 70).)

  If your app spends more time in GPU processing, you may be able to improve performance by moving work to the CPU. For example, if a shader performs the same calculation with the same result for every vertex or fragment during a particular draw call, you can perform that computation once on the CPU and pass its result to the shader in a uniform variable. (See "Use Uniforms or Constants Instead of Computing Values in a Shader" (page 99).)
- **Program Performance.** Only appears after you capture a frame (see "Capturing and Analyzing an OpenGL ES Frame" (page 118) below), showing the time spent in each shader program while rendering the captured frame, both in milliseconds and as a percentage of the total frame rendering time. Expanding the listing

for a program shows the draw calls made using that program and the rendering time contribution from each. Select a program in the list to view its shader source code in the assistant editor, or click the arrow icon next to a draw call to select that call in the frame navigator (see "Navigator Area" (page 121) below).

> **Note:** The Program Performance view only appears when debugging on devices that support OpenGL ES 3.0 (regardless of whether your app uses an OpenGL ES 3.0 or 2.0 context).

When tuning your app, you can use this graph to find opportunities for optimization. For example, if one program takes 50% of the frame rendering time, you gain more performance by optimizing it than by improving the speed of a program that accounts for only 10% of frame time. Though this view organizes frame time by shader program, remember that improving your shader algorithms isn't the only way to optimize your app's performance—for example, you can also reduce the number of draw calls that use a costly shader program, or reduce the number of fragments processed by a slow fragment shader.

- **Problems & Solutions.** Only appears after Xcode analyzes a frame capture (see "Capturing and Analyzing an OpenGL ES Frame" (page 118)), this area lists possible issues found during analysis and recommendations for improving performance.

When you make changes to a GLSL shader program in a captured frame (see "Editing Shader Programs" (page 125) below), the Frame Time and Program Performance graphs expand to show both the baseline rendering time of the frame as originally captured and the current rendering time using your edited shaders.

Capturing and Analyzing an OpenGL ES Frame

For a detailed look at your app's OpenGL ES usage, capture the sequence of OpenGL ES commands used to render a single frame of animation. Xcode offers several ways to begin a frame capture:

- **Manual capture.** While running your app in Xcode, click the camera icon in the debug bar (shown in Figure B-2) or choose Capture OpenGL ES Frame from the Debug menu.

Figure B-2 Debug Bar with Capture OpenGL ES Frame button

> **Note:** The Capture OpenGL ES Frame button automatically appears only if your project links against the OpenGL ES or Sprite Kit framework. You can choose whether it appears for other projects by editing the active scheme. (See About the Scheme Editing Dialog.)

- **Breakpoint action.** Choose Capture OpenGL ES Frame as an action for any breakpoint. When the debugger reaches a breakpoint with this action, Xcode automatically captures a frame. (See Setting Breakpoint Actions and Options.) If you use this action with an OpenGL ES Error breakpoint while developing your app (see Adding an OpenGL ES Error Breakpoint), you can use the OpenGL ES Frame Debugger to investigate the causes of OpenGL ES errors whenever they occur.
- **OpenGL ES event marker.** Programmatically trigger a frame capture by inserting an event marker in the OpenGL ES command stream. The following command inserts such a marker:

  ```
  glInsertEventMarkerEXT(0, "com.apple.GPUTools.event.debug-frame")
  ```

  When the OpenGL ES client reaches this marker, it finishes rendering the frame, then Xcode automatically captures the entire sequence of commands used to render that frame.

After Xcode has captured the frame, it shows the OpenGL ES Frame Debugger interface. Use this interface to inspect the sequence of OpenGL ES commands that render the frame and examine OpenGL ES resources, as discussed in "Touring the OpenGL ES Frame Debugger" (page 120).

In addition, Xcode can perform an automated analysis of your app's OpenGL ES usage to determine which parts of your renderer and shader architecture can benefit most from performance optimizations. To use this option, click the Analyze button at the top of the GPU report (shown at the top right in Figure B-1 (page 116)).

When you click the Analyze button, Xcode captures a frame (if one hasn't been captured already), then runs your rendering code through a series of experiments using the attached iOS device. For example, to see if your rendering speed is limited by texture sizes, Xcode runs the captured sequence of OpenGL ES commands both with the texture data your app submitted to the GPU and with a size-reduced texture set. After Xcode finishes its analysis, the Problems & Solutions area of the GPU report lists any issues it found and suggestions for possible performance improvements.

Touring the OpenGL ES Frame Debugger

After Xcode captures a frame, it automatically reconfigures its interface for OpenGL ES debugging. The OpenGL ES Frame Debugger interface modifies several areas of the Xcode workspace window to provide information about the OpenGL ES rendering process, as shown in Figure B-3 and Figure B-4 and summarized below. (The frame debugger does not use the inspector or library panes, so you may wish to hide Xcode's utility area during OpenGL ES debugging.)

Figure B-3 Frame debugger examining draw calls and resources

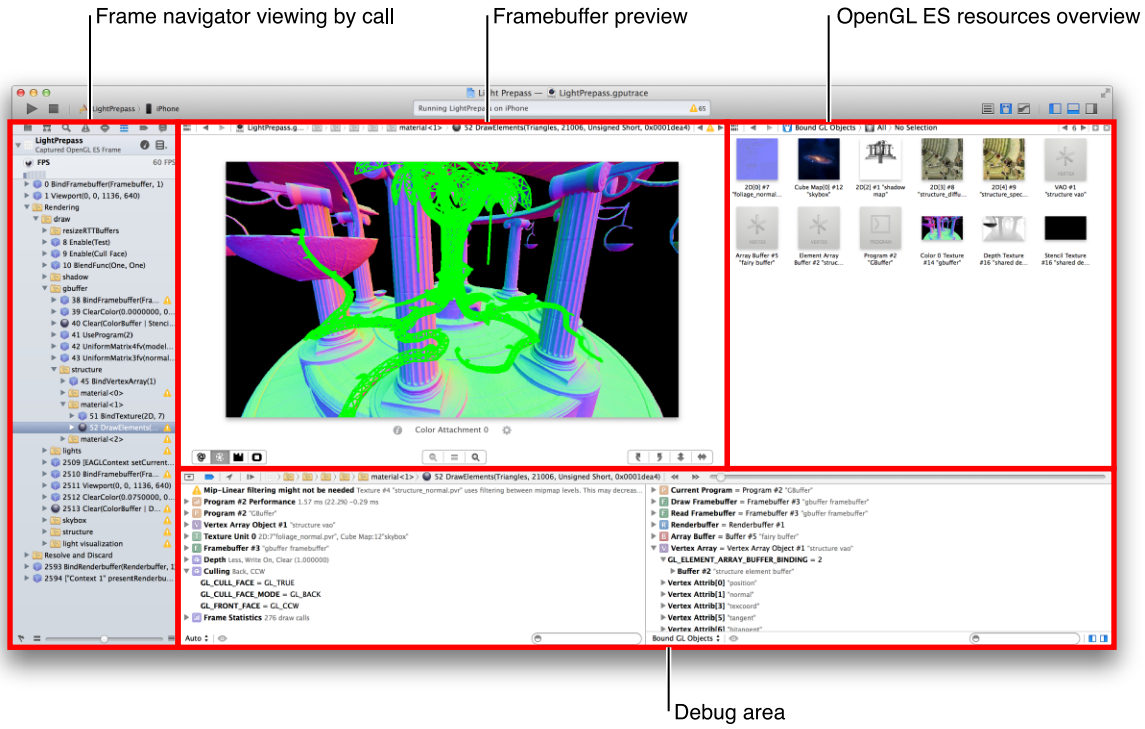
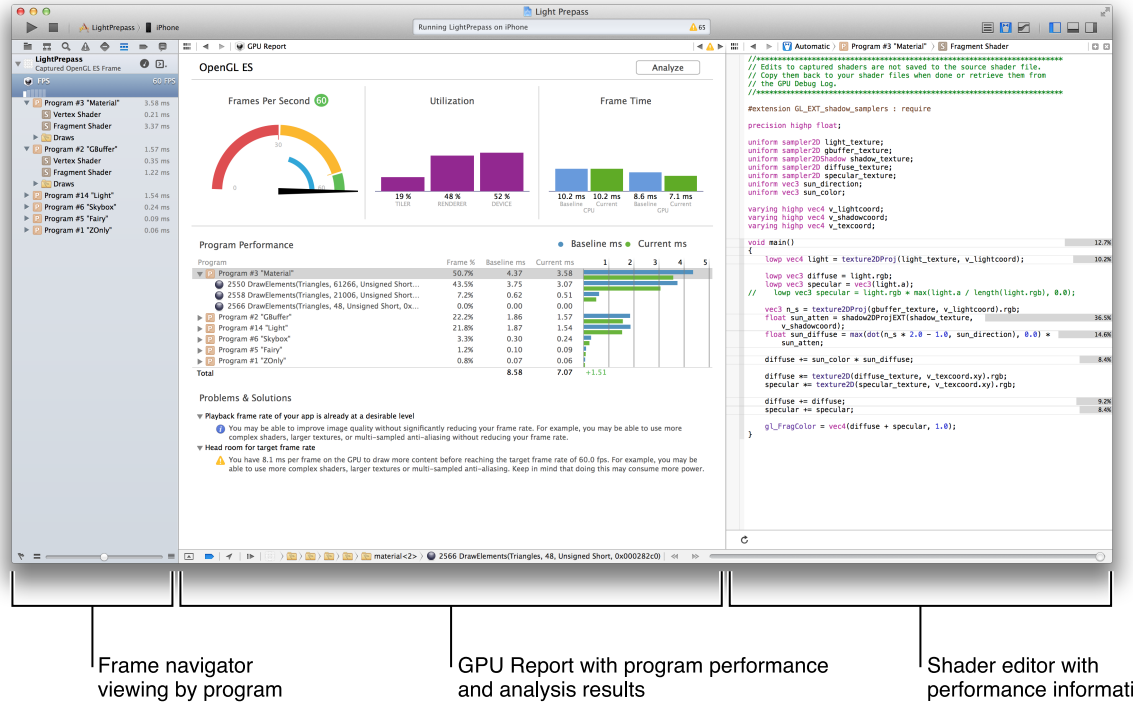


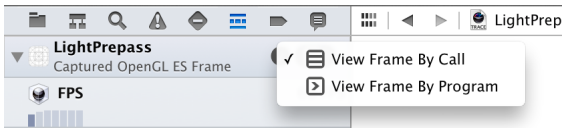
Figure B-4 Frame debugger examining shader program performance and analysis results



Navigator Area

In the OpenGL ES frame debugger interface, the debug navigator is replaced by the OpenGL ES frame navigator. This navigator shows the OpenGL ES commands that render the captured frame, organized sequentially or according to their associated shader program. Use the Frame View Options popup menu at the top of the frame navigator to switch between view styles.

Figure B-5 View Frame By popup menu in navigator



View Frame By Call

View the captured frame by call when you want to study OpenGL ES commands in sequence to pinpoint errors, diagnose rendering problems, or identify common performance issues. In this mode, the frame navigator lists commands in the order your app called them. Error or warning icons appear next to commands that result in OpenGL ES errors or that may indicate performance issues.

You can add structure to this list by using the `glPushGroupMarkerEXT` and `glPopGroupMarkerEXT` functions to annotate groups of OpenGL ES commands—these groups appear as folders you can expand or collapse to show more or less detail. (For details, see "Annotate Your OpenGL ES Code for Informative Debugging and Profiling" (page 64).) You can also expand an OpenGL ES command to show a stack trace indicating where in your application code the command was issued.

Use the context menu to choose whether to abbreviate command names and which commands, groups, and warnings to show. Use the flag icon at the bottom of the navigator to switch between showing all OpenGL ES commands and showing only those which draw into the framebuffer.

Clicking an OpenGL ES command in the list navigates to that point in the OpenGL ES command sequence, affecting the contents of other areas of the frame debugger interface, as discussed below, and showing the effects of the OpenGL ES calls up to that point on the attached device's display.

View Frame By Program

View the captured frame by program when you want to analyze the GPU time spent on each shader program and draw command.

Expand the listing for a program to see the time contribution from each shader in the program and each draw call. Expand the listing for a draw call to show a stack trace indicating where in your application code that command was issued.

Use the context menu to refine the display—you can choose whether programs are sorted by their time contributions and whether timing information is displayed as a percentage of the total rendering time.

Clicking a program or shader shows the corresponding GLSL source code in the primary editor. Clicking an OpenGL ES command navigates to that point in the frame capture sequence.

> **Note:** The View Frame By Program option is only available when debugging on devices that support OpenGL ES 3.0 (regardless of whether your app uses an OpenGL ES 3.0 or 2.0 context). On other devices, the Frame View Options popup menu is disabled.

Editor Area

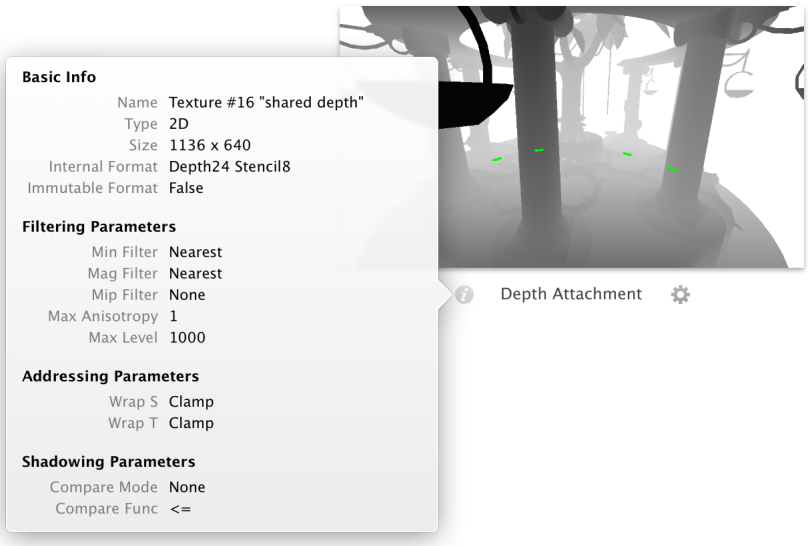
When working with a frame capture, you use the primary editor to preview the framebuffer being rendered to, and the assistant editor to examine OpenGL ES resources and edit GLSL shader programs. By default, the assistant editor shows a graphical overview of all resources currently owned by the OpenGL ES context, as shown in Figure B-3 (page 120). Use the assistant editor's jump bar to show only those resources bound for

use as of the call selected in the frame navigator, or to select an individual resource for further inspection. You can also double-click a resource in the overview to inspect it. When you select a resource, the assistant editor changes to a format suited for tasks appropriate to that resource's type.

Previewing Framebuffer Contents

The primary editor shows the contents of the framebuffer as rendered by the draw call currently selected in the frame navigator. (If the selected OpenGL ES command in the frame navigator is not a drawing command—for example, a command that sets state such as `glUseProgram`—the framebuffer reflects the rendering done by the most recent draw call prior to the selection.) You can also navigate the sequence of OpenGL ES commands using the jump bar at the top of the primary editor.

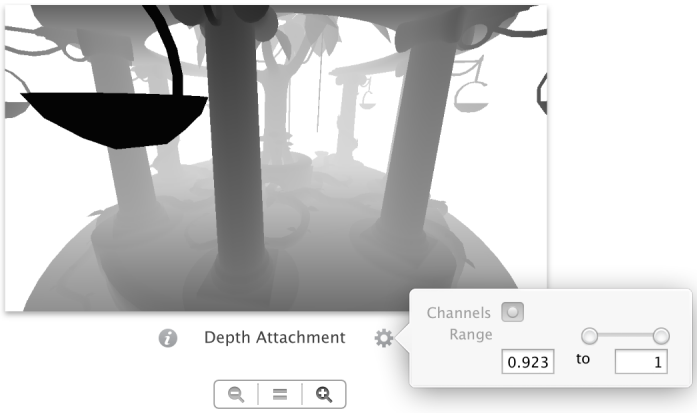
Figure B-6 Framebuffer info popover



The editor shows a preview for each framebuffer attachment currently bound for drawing. For example, most approaches to 3D rendering use a framebuffer with attachments for both color and depth, as illustrated in . Use the controls in the lower left of the editor to choose which framebuffer attachments are currently shown. Clicking the info button, left of each framebuffer attachment's name, shows a popover detailing the attachment's

properties, as shown in Figure B-6. Click the settings button, right of the framebuffer attachment's name, to show a popover with controls that adjust the preview image. For example, you can use these controls to make a certain range of Z values in a depth buffer more visible in its grayscale preview, as shown in Figure B-7.

Figure B-7 Framebuffer settings popover

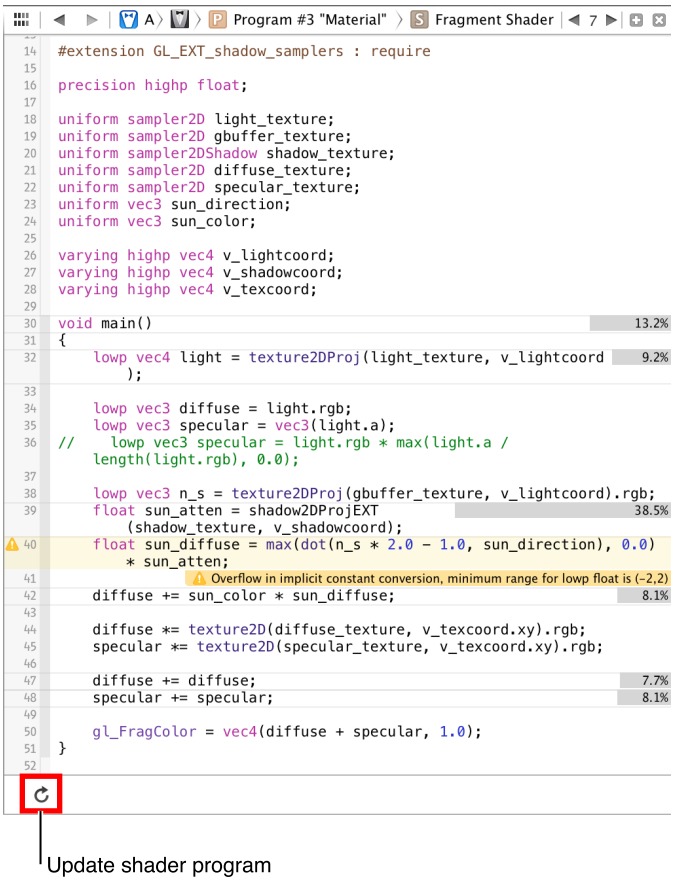


Each framebuffer attachment preview also shows a green wireframe highlighting the effect of the current draw call (as illustrated in Figure B-3 (page 120)). Use the context menu in a preview image to choose whether the highlight appears in the preview or on the display of the attached device.

Editing Shader Programs

When you select a shader program in the assistant editor's jump bar or resource overview, the assistant editor shows the GLSL source code for that program's fragment shader (as shown in Figure B-8). When you select a program in the frame navigator (see "View Frame By Program" (page 122)), the primary editor shows the program's fragment shader and the assistant editor shows its vertex shader. In any editor showing a fragment shader, you can use the jump bar to switch to its counterpart vertex shader, and vice versa.

Figure B-8 GLSL shader source editor with update button



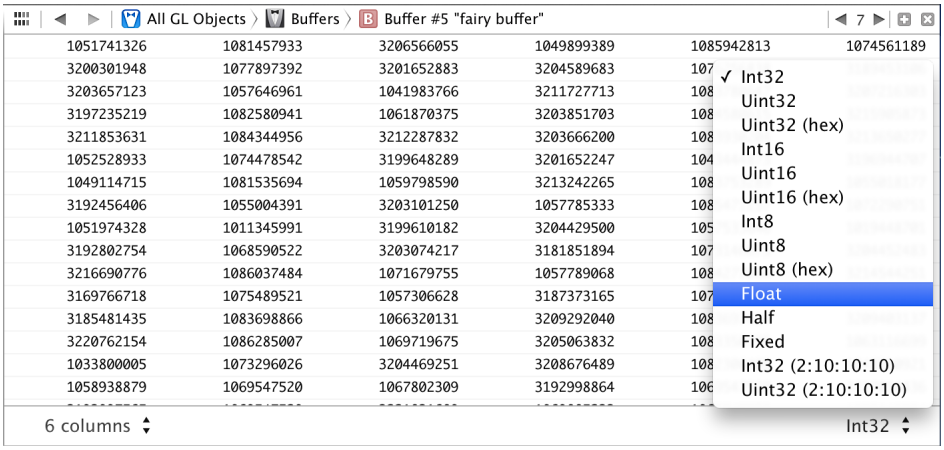
Each line of the shader source code is highlighted in the right margin with a bar representing its relative contribution to rendering time. Use these to focus your shader optimization efforts—if a few lines account for a greater share of rendering time, look into faster alternatives for those lines. (For shader performance tips, see "Best Practices for Shaders" (page 94).)

You can make changes to the shader source code in the editor. Then, click the Update button below the editor (shown in Figure B-8 (page 125)) to recompile the shader program and see its effects on the captured frame. If compiling the shader results in error or warning messages from the GLSL compiler, Xcode annotates the shader source code for each issue. The recompiled shader program remains in use on the device, so you can resume running your app. Click the Continue button in the debug bar to see your shader changes in action.

Inspecting Vertex Data

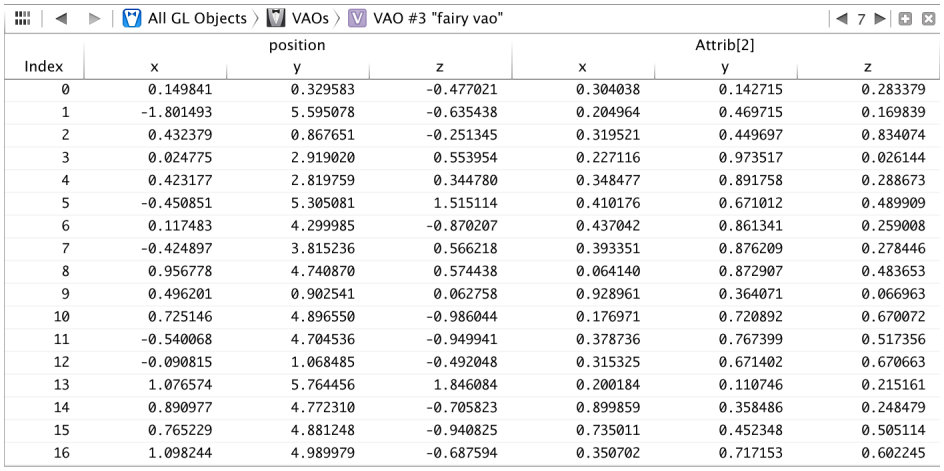
When you inspect an array buffer, the assistant editor shows the contents of the buffer (see Figure B-9). Because a buffer in OpenGL ES memory has no defined format, you use the pop-up menus at the bottom of the editor to choose how its contents appear (for example, as 32-bit integers or floating-point values, or as twice as many 16-bit integers or half-float values), and how many columns Xcode uses to display the data.

Figure B-9 Assistant editor previewing array buffer contents



A vertex array object (VAO) encapsulates one or more data buffers in OpenGL ES memory and the attribute bindings used for supplying vertex data from the buffers to a shader program. (For details on using VAOs, see "Consolidate Vertex Array State Changes Using Vertex Array Objects" (page 84).) Because the VAO bindings include information about the format of the buffers' contents, inspecting a VAO shows its contents as interpreted by OpenGL ES (see Figure B-10).

Figure B-10 Assistant editor previewing vertex array object

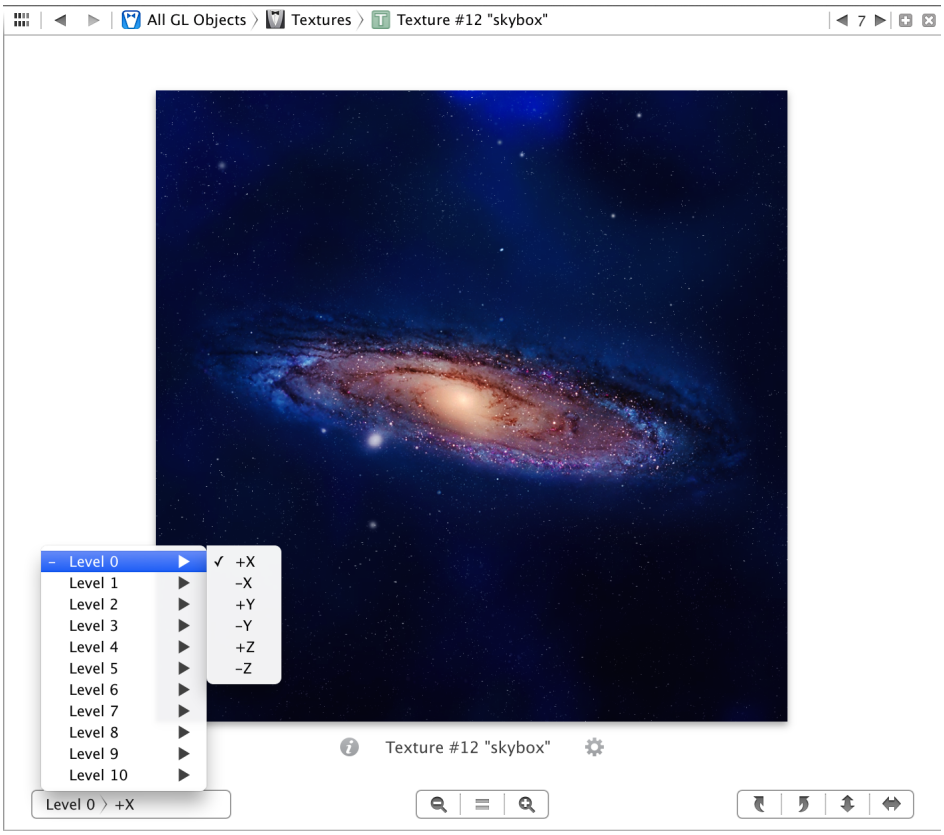


	position			Attrib[2]		
Index	x	y	z	x	y	z
0	0.149841	0.329583	-0.477021	0.304038	0.142715	0.283379
1	-1.801493	5.595078	-0.635438	0.204964	0.469715	0.169839
2	0.432379	0.867651	-0.251345	0.319521	0.449697	0.834074
3	0.024775	2.919020	0.553954	0.227116	0.973517	0.026144
4	0.423177	2.819759	0.344780	0.348477	0.891758	0.288673
5	-0.450851	5.305081	1.515114	0.410176	0.671012	0.489909
6	0.117483	4.299985	-0.870207	0.437042	0.861341	0.259008
7	-0.424897	3.815236	0.566218	0.393351	0.876209	0.278446
8	0.956778	4.740870	0.574438	0.064140	0.872907	0.483653
9	0.496201	0.902541	0.062758	0.928961	0.364071	0.066963
10	0.725146	4.896550	-0.986044	0.176971	0.720892	0.670072
11	-0.540068	4.704536	-0.949941	0.378736	0.767399	0.517356
12	-0.090815	1.068485	-0.492048	0.315325	0.671402	0.670663
13	1.076574	5.764456	1.846084	0.200184	0.110746	0.215161
14	0.890977	4.772310	-0.705823	0.899859	0.358486	0.248479
15	0.765229	4.881248	-0.940825	0.735011	0.452348	0.505114
16	1.098244	4.989979	-0.687594	0.350702	0.717153	0.602245

Viewing Textures or Renderbuffers

When you inspect a texture or renderbuffer, the assistant editor shows an image preview of its contents. You can use the same controls found in the primary editor to get more information about the texture object or renderbuffer and to adjust the image preview. For textures, you can use an additional control in the lower left corner of the assistant editor to preview each mipmap level of the texture and (if applicable) each face of a cube map texture (as shown in Figure B-11).

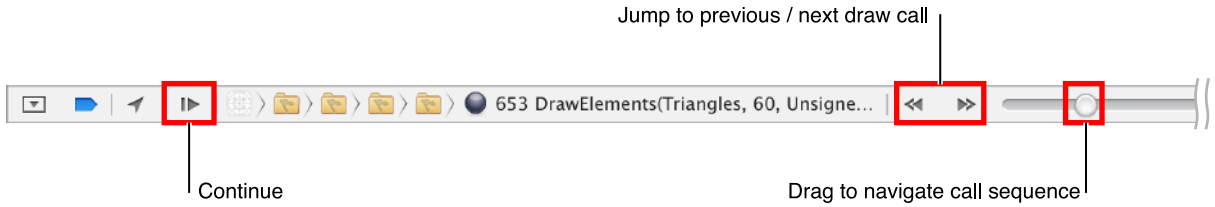
Figure B-11 Assistant editor previewing cube map texture



Debug Area

The debug bar provides multiple controls for navigating the captured sequence of OpenGL ES commands (shown in Figure B-12). You can use its menus to follow the hierarchy shown in the frame navigator and choose a command, or you can use the arrows and slider to move back and forth in the sequence. Press the Continue button to end frame debugging and return to running your application.

Figure B-12 OpenGL ES debug bar



The frame debugger has no debug console. Instead, Xcode offers multiple variables views, each of which provides a different summary of the current state of the OpenGL ES rendering process. Use the popup menu to choose between the available variables views, discussed in the following sections.

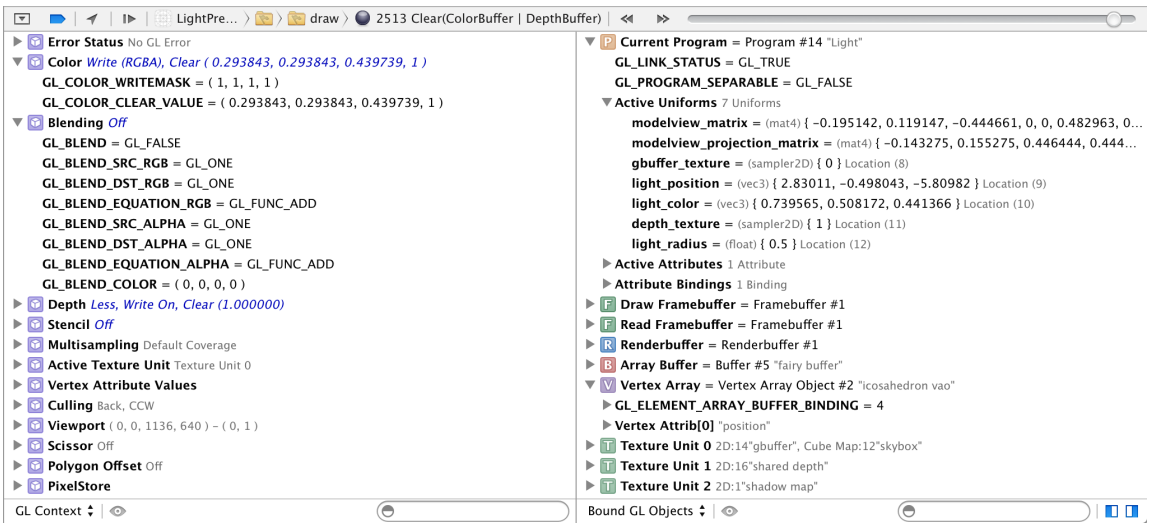
The All GL Objects View

The All GL Objects view, similar to the Bound GL Objects view shown on the right in Figure B-13 (page 129), lists the same OpenGL ES resources as the graphical overview in the assistant editor. Unlike the graphical overview, however, this view can provide more detailed information about a resource when you expand its disclosure triangle. For example, expanding the listing for a framebuffer or buffer object shows information otherwise available only through OpenGL ES query functions such as `glGetBufferParameter` and `glGetFramebufferAttachmentParameter`. Expanding the listing for a shader program shows its status, attribute bindings, and the currently bound value for each uniform variable.

The Bound GL Objects View

The Bound GL Objects view, shown on the right in Figure B-13, behaves identically to the *All GL Objects* view, but lists only resources currently bound for use as of the selected OpenGL ES command in the frame navigator.

Figure B-13 Debug area with GL Context and Bound GL Objects views



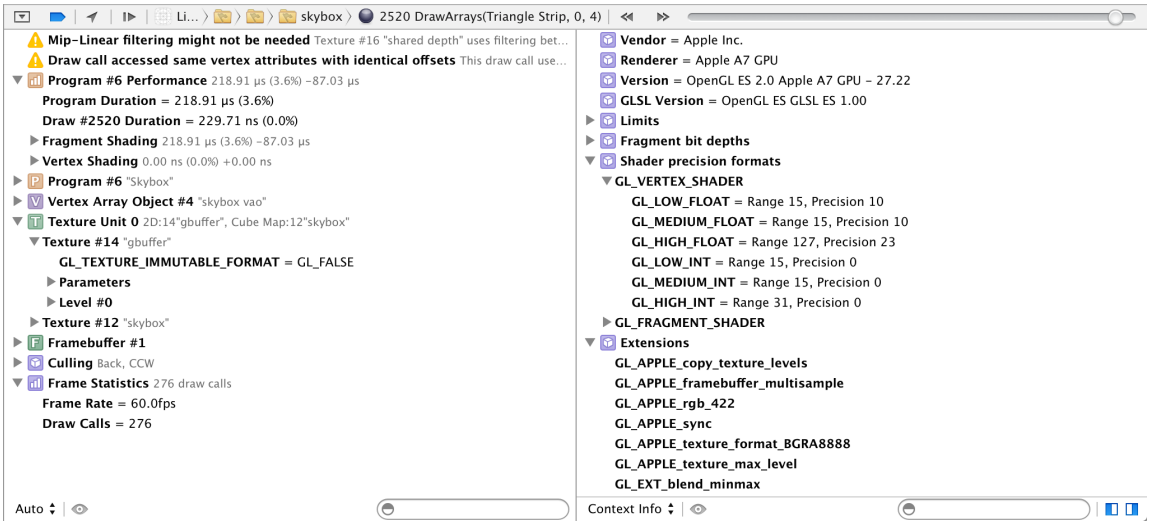
The GL Context View

The GL Context view, shown on the left in Figure B-13, lists the entire state vector of the OpenGL ES renderer, organized into functional groups. When you select a call in the frame navigator that changes OpenGL ES state, the changed values appear highlighted. For example, calling the `glCullFace` or `glFrontFace` function changes and highlights values in the Culling section of the state list. Enabling blending with the `glEnable(GL_BLEND)` call or changing blending parameters with the `glBlendFunc` function changes and highlights values in the Blending section of the state list.

The Context Info View

The Context Info view, shown on the right in Figure B-14, lists static information about the OpenGL ES renderer in use: name, version, capabilities, extensions and similar data. You can look through this data instead of writing your own code to query renderer attributes such as `GL_MAX_TEXTURE_IMAGE_UNITS` and `GL_EXTENSIONS`.

Figure B-14 Debug area with Auto and Context Info views



The Auto View

The Auto view, shown on the left in Figure B-14, automatically lists a subset of items normally found in the other variables views and other information appropriate to the selected call in the frame navigator. For example:

- If the selected call results in an OpenGL ES error, or if Xcode has identified possible performance issues with the selected call, the view lists the errors or warnings and suggested fixes for each.
- If the selected call changes part of the OpenGL ES context state, or its behavior is dependent on context state, the view automatically lists relevant items from the *GL Context* view.
- If the selected call binds a resource or makes use of bound resources such as vertex array objects, programs, or textures, the view automatically lists relevant items from the *Bound GL Objects* view.
- If a draw call is selected, the view lists program performance information, including the total time spent in each shader during that draw call and, if you've changed and recompiled shaders since capturing the frame, the difference from the baseline time spent in each shader. (Program performance information is only available when debugging on an OpenGL ES 3.0–capable device.)

In addition, this view lists aggregate statistics about frame rendering performance, including the number of draw calls and frame rate.

Using texturetool to Compress Textures

The iOS SDK includes a tool to compress your textures into the PVR texture compression format, aptly named `texturetool`. If you have Xcode installed with the iOS 7.0 SDK, then `texturetool` is located at: `/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/texturetool`.

`texturetool` provides various compression options, with tradeoffs between image quality and size. You need to experiment with each texture to determine which setting provides the best compromise.

> **Note:** The encoders, formats, and options available with `texturetool` are subject to change. This document describes those options available as of iOS 7.

texturetool Parameters

The parameters that may be passed to `texturetool` are described in the rest of this section.

```
user$ texturetool -h

Usage: texturetool [-hl]
       texturetool -c <reference_image> <input_image>
      texturetool [-ms] [-e <encoder>] [-p <preview_file>] -o <output> [-f <format>]
 <input_image>

first form:
    -h       Display this help menu.
   -l      List available encoders, individual encoder options, and file formats.

second form:
    -c       Compare <input_image> to <reference_image> and report differences.

third form:
    -m               Generate a complete mipmap chain from the input image.
```

```
    -s                Report numerical differences between <input_image> and the
encoded image.
    -e <encoder>      Encode texture levels with <encoder>.
   -p <preview_file> Output a PNG preview of the encoded output to <preview_file>.
 Requires -e option
    -o <output>       Write processed image to <output>.
    -f <format>       Set file <format> for <output> image.
```

Note: The -p option indicates that it requires the -e option. It also requires the -o option.

Listing C-1 Encoding options

```
user$ texturetool -l
Encoders:

PVRTC
 --channel-weighting-linear
 --channel-weighting-perceptual
 --bits-per-pixel-2
 --bits-per-pixel-4
 --alpha-is-independent
 --alpha-is-opacity
 --punchthrough-unused
 --punchthrough-allowed
 --punchthrough-forced

Formats:

Raw
PVR
```

`texturetool` defaults to `--bits-per-pixel-4`, `--channel-weighting-linear` and `-f Raw` if no other options are provided.

The `--bits-per-pixel-2` and `--bits-per-pixel-4` options create PVRTC data that encodes source pixels into 2 or 4 bits per pixel. These options represent a fixed 16:1 and 8:1 compression ratio over the uncompressed 32-bit RGBA image data. There is a minimum data size of 32 bytes; the compressor never produces files smaller than this, and at least that many bytes are expected when uploading compressed texture data.

When compressing, specifying `--channel-weighting-linear` spreads compression error equally across all channels. By contrast, specifying `--channel-weighting-perceptual` attempts to reduce error in the green channel compared to the linear option. In general, PVRTC compression does better with photographic images than with line art.

The `-m` option automatically generates mipmap levels for the source image. These levels are provided as additional image data in the archive created. If you use the Raw image format, then each level of image data is appended one after another to the archive. If you use the PVR archive format, then each mipmap image is provided as a separate image in the archive.

The (`-f`) parameter controls the format of its output file. The default format is Raw. This format is raw compressed texture data, either for a single texture level (without the `-m` option) or for each texture level concatenated together (with the `-m` option). Each texture level stored in the file is at least 32 bytes in size and must be uploaded to the GPU in its entirety. The PVR format matches the format used by the PVRTexTool found in Imagination Technologies's PowerVR SDK. To load a PVR-compressed texture, use the `GLKTextureLoader` class.

The `-s` and `-c` options print error metrics during encoding. The `-s` option compares the input (uncompressed) image to the output (encoded) image, and the `-c` option compares any two images. Results of the comparison include root-mean-square error (`rms`), perceptually weighted `pRms`, worst-case texel error (`max`), and compositing-based versions of each statistic (`rmsC`, `pRmsC`, and `maxC`). Compositing-based errors assume that the image's alpha channel is used for opacity and that the color in each texel is blended with the worst-case destination color.

The error metrics used with the `-s` and `-c` options and by the encoder when optimizing a compressed image treat the image's alpha channel as an independent channel by default (or when using the `--alpha-is-independent` option). The `--alpha-is-opacity` option changes the error metric to one based on a standard blending operation, as implemented by calling `glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA )`.

PVR Texture compression supports a special punchthrough mode which can be enabled on a per 4x4 block basis. This mode limits the color gradations that can be used within the block, but introduces the option of forcing the pixel's alpha value to 0. It can defeat PVRTC smooth color interpolation, introducing block boundary artifacts, so it should be used with care. The three punchthrough options are:

- `--punchthrough-unused` — No punchthrough (the default option).

- `--punchthrough-allowed` — The encoder may enable punchthrough on a block by block basis when optimizing for image quality. This option generally improves the objective (per-pixel) error metrics used by the compression algorithm, but may introduce subjective artifacts.
- `--punchthrough-forced` — Punchthrough is enabled on every block, limiting color gradation but making it possible for any pixel in the block to have an alpha of 0. This option is provided principally for completeness, but may be useful when the results can be compared visually to the other options.

Important: Source images for the encoder must satisfy these requirements:

- Height and width must be at least 8.
- Height and width must be a power of 2.
- Must be square (height==width).
- Source images must be in a format that Image IO accepts in OS X. For best results, your original textures should begin in an uncompressed data format.

Important: If you are using PVRTexTool to compress your textures, then you must create textures that are square and a power of 2 in length. If your app attempts to load a non-square or non-power-of-two texture in iOS, an error is returned.

Listing C-2 Encoding images into the PVRTC compression format

```
Encode Image.png into PVRTC using linear weights and 4 bpp, and saving as
ImageL4.pvrtc
user$ texturetool -e PVRTC --channel-weighting-linear --bits-per-pixel-4 -o
ImageL4.pvrtc Image.png

Encode Image.png into PVRTC using perceptual weights and 4 bpp, and saving as
ImageP4.pvrtc
user$ texturetool -e PVRTC --channel-weighting-perceptual --bits-per-pixel-4 -o
ImageP4.pvrtc Image.png

Encode Image.png into PVRTC using linear weights and 2 bpp, and saving as
ImageL2.pvrtc
user$ texturetool -e PVRTC --channel-weighting-linear --bits-per-pixel-2 -o
ImageL2.pvrtc Image.png

Encode Image.png into PVRTC using perceptual weights and 2 bpp, and saving as
ImageP2.pvrtc
user$ texturetool -e PVRTC --channel-weighting-perceptual --bits-per-pixel-2 -o
ImageP2.pvrtc Image.png
```

Listing C-3 Encoding images into the PVRTC compression format while creating a preview

```
Encode Image.png into PVRTC using linear weights and 4 bpp, and saving the output
 as ImageL4.pvrtc and a PNG preview as ImageL4.png
user$ texturetool -e PVRTC --channel-weighting-linear --bits-per-pixel-4 -o
ImageL4.pvrtc -p ImageL4.png Image.png

Encode Image.png into PVRTC using perceptual weights and 4 bpp, and saving the
output as ImageP4.pvrtc and a PNG preview as ImageP4.png
user$ texturetool -e PVRTC --channel-weighting-perceptual --bits-per-pixel-4 -o
ImageP4.pvrtc -p ImageP4.png Image.png

Encode Image.png into PVRTC using linear weights and 2 bpp, and saving the output
 as ImageL2.pvrtc and a PNG preview as ImageL2.png
user$ texturetool -e PVRTC --channel-weighting-linear --bits-per-pixel-2 -o
ImageL2.pvrtc -p ImageL2.png Image.png

Encode Image.png into PVRTC using perceptual weights and 2 bpp, and saving the
output as ImageP2.pvrtc and a PNG preview as ImageP2.png
user$ texturetool -e PVRTC --channel-weighting-perceptual --bits-per-pixel-2 -o
ImageP2.pvrtc -p ImageP2.png Image.png
```

> **Note:** It is not possible to create a preview without also specifying the `-o` parameter and a valid output file. Preview images are always in PNG format.

To load a PVR-compressed texture, use the `GLKTextureLoader` class.

For an example of working with PVR-compressed data directly, see the *PVRTextureLoader* sample.

Document Revision History

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

Date	Notes
2014-03-10	Added and expanded discussions of OpenGL ES 3.0 and important extensions, new multithreading support in iOS 7.1, Xcode OpenGL ES tools, GPU architecture and performance issues.
2013-09-18	Updated to include more information about OpenGL ES 3.0, GLKit, and the Xcode debugger.
2013-04-23	Moved the platform notes to OpenGL ES Hardware Platform Guide for iOS. Removed the “Platform Notes” chapter and moved the information into its own book, *OpenGL ES Hardware Platform Guide for iOS*.
2011-02-24	Added information about new OpenGL ES tools provided in Xcode 4. Clarified that context sharing can only be used when all of the contexts share the same version of the OpenGL ES API.
2010-11-15	Significantly revised and expanded all the material in the document. Added a glossary of commonly used graphics and OpenGL ES terminology. Added a detailed explanation of the rendering loop, including enhancements added in iOS 4 (renderbuffer discards).
2010-09-01	Fixed an incorrect link. Clarified some performance guidelines. Added links to more new extensions added in iOS 4.
2010-07-09	Changed the title from "OpenGL ES Programming Guide for iPhone OS."
2010-06-14	Added new extensions exposed by iOS 4.
2010-01-20	Corrected code for creating a framebuffer object that draws to the screen.

Date	Notes
2009-11-17	Minor updates and edits.
2009-09-02	Edited for clarity. Updated extensions list to reflect what's currently available. Clarified usage of triangle strips for best vertex performance. Added a note to the platforms chapter about texture performance on the PowerVR SGX.
2009-06-11	First version of a document that describes how to use the OpenGL ES 1.1 and 2.0 programming interfaces to create high performance graphics within an iPhone Application.

Glossary

This glossary contains terms that are used specifically for the Apple implementation of OpenGL ES as well as terms that are common in OpenGL ES graphics programming.

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.

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.

buffer A block of memory managed by OpenGL ES dedicated to storing a specific kind of data, such as vertex attributes, color data or indices.

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.

completeness A state that indicates whether a framebuffer object meets all the requirements for drawing.

context A set of OpenGL ES state variables that affect how drawing is performed to 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 ES routes commands issued by your app.

current matrix A matrix used by OpenGL ES 1.1 to transform coordinates in one system to those of another system, such as the modelview matrix, the perspective matrix, and the texture matrix. GLSL ES uses user-defined matrices instead.

depth In OpenGL, the *z* coordinate that 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. All fragments rasterized by OpenGL ES must pass a depth test that compares the incoming depth value to the value stored in the depth buffer; only fragments that pass the depth test are stored to framebuffer.

double buffering The practice of using two buffers to avoid resource conflicts between two different parts of the graphic subsystem. The front buffer is

used by one participant and the back buffer is modified by the other. When a swap occurs, the front and back buffer change places.

drawable object An object allocated outside of OpenGL ES that can be used as part of an OpenGL ES framebuffer object. On iOS, the only type of drawable object is the `CAEAGLLayer` class that integrates OpenGL ES rendering into Core Animation.

extension A feature of OpenGL ES that's not part of the OpenGL ES core API and therefore not guaranteed to be supported by every implementation of OpenGL ES. 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 `OES` 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.

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 calculated when rasterizing a primitive. Each fragment must past a series of tests before being blended with the pixel stored in the framebuffer.

system framebuffer A framebuffer provided by an operating system. This type of framebuffer supports integrating OpenGL ES into an operating system's windowing system. iOS does not use system framebuffers. Instead, it provides framebuffer objects that are associated with a Core Animation layer.

framebuffer attachable image The rendering destination for a framebuffer object.

framebuffer object A framebuffer that is managed entirely by OpenGL ES. A framebuffer object contains state information for an OpenGL ES framebuffer and its set of images, called *renderbuffers* . Framebuffers are built into OpenGL ES 2.0 and later, and all iOS implementations of OpenGL ES 1.1 are guaranteed to support framebuffer objects (through the `OES_framebuffer_object` extension).

frustum The region of space that is seen by the observer and that is warped by perspective division.

image A rectangular array of pixels.

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 transform points, lines, polygons, and positions from object coordinates to eye coordinates.

multisampling A technique that takes multiple samples at a pixel and combines them with coverage values to arrive at a final fragment.

mutex A mutual exclusion object in a multithreaded app.

packing Converting pixel color components from a buffer into the format needed by an app.

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 depth In a pixel image, the number of bits per pixel.

pixel format A format used to store pixel data in memory. The format describes the pixel components (red, green, blue, alpha), the number and order of components, and other relevant information, such as whether a pixel contains stencil and depth values.

premultiplied alpha A pixel whose other components have been multiplied by the alpha value. For example, a pixel whose RGBA values start as (1.0, 0.5, 0.0, 0.5) would, when premultiplied, be (0.5, 0.25, 0.0, 0.5).

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 `OES_framebuffer_object` extension.

renderer A combination of hardware and software that OpenGL ES uses to create an image from a view and a model.

rendering context A container for state information.

rendering pipeline The order of operations used by OpenGL ES 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.

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. iOS avoids tearing by processing all visible OpenGL ES content through Core Animation.

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. The data can be one-, two-, or three- dimensional or it can be a cube map.

texture mapping The process of applying a texture to a primitive.

texture matrix A 4 x 4 matrix that OpenGL ES 1.1 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.

vertex array object An OpenGL ES object that records a list of active vertex attributes, the format each attribute is stored in, and the location of the data describing vertices and attributes. Vertex array objects simplify the effort of reconfiguring the graphics pipeline.

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