OpenGL ES Programming Guide for iOS

Developer

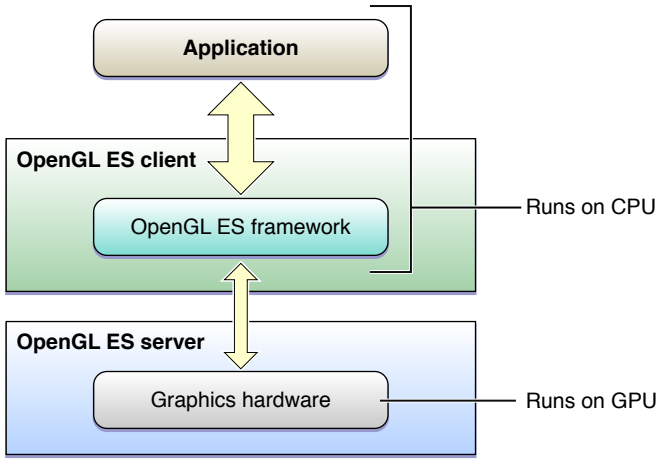
Contents

Figures, Tables, and Listings

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. OpenGL allows application developers 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

Apple provides implementations of both OpenGL ES 1.1 and OpenGL ES 2.0:

- OpenGL ES 1.1 implements a well-defined **fixed-function pipeline**. The fixed-function pipeline implements a traditional lighting and shading model that allows each stage of the pipeline to be configured to perform specific tasks, or disabled when its functionality is not required.

- OpenGL ES 2.0 shares many functions in common with OpenGL ES 1.1, but removes all functions that target the fixed-function vertex and fragment pipeline stages. Instead, it introduces new functions that provide access to a general-purpose **shader**-based pipeline. Shaders allow you to author custom vertex and fragment functions that are executed directly on the graphics hardware. With shaders, your application can more precisely and more clearly customize the inputs to the pipeline and the computations that are performed on each vertex and fragment.

OpenGL ES is a C-based, Platform-Neutral API

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 based Cocoa Touch applications. 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.

Relevant Chapters "OpenGL ES on iOS" (page 13), "Configuring OpenGL ES Contexts" (page 20)

OpenGL ES is Integrated into Core Animation

Core Animation is fundamental to iOS graphics, and that includes OpenGL ES content that your application delivers to the screen. When you develop an OpenGL ES application, your OpenGL ES content is rendered to a special Core Animation layer, known as a `CAEAGLLayer` object. The images you render using OpenGL ES are stored in the `CAEAGLLayer`. Core Animation composites these images with content in other layers and delivers the final image to the screen.

Relevant Chapters "Drawing With OpenGL ES" (page 28)

Framebuffer Objects are Always the Rendering Destination

The OpenGL ES specification assumes there are two kinds of framebuffers: system-provided framebuffers and framebuffer objects. A system framebuffer is allocated using an API provided by the host operating system, and connects to the screen or windowing environment. Framebuffer objects target offscreen rendering without a direct connection to the screen or windowing environment. iOS only uses framebuffer objects; rather than create a separate system framebuffer, iOS extends the OpenGL ES API to allow a framebuffer object to be allocated so that its contents are shared with Core Animation.

Relevant Chapters "Drawing With OpenGL ES" (page 28)

Functionality May Vary on Different Devices

iOS devices support a variety of graphics processors with varying capabilities. Some devices support both OpenGL ES 1.1 and OpenGL ES 2.0; older devices only support OpenGL ES 1.1. Even devices that support the same version of OpenGL ES may have different capabilities depending on the version of iOS and the underlying graphics hardware. Apple offers many OpenGL ES extensions to provide new capabilities to your applications.

To allow your application to run on as many devices as possible and to ensure compatibility with future devices and iOS versions, your application must always test the capabilities of the underlying OpenGL ES implementation at run time, disabling any features that do not have the required support from iOS.

Relevant Chapters "Configuring OpenGL ES Contexts" (page 20), "Determining OpenGL ES Capabilities" (page 25), "Platform Notes" (page 88)

Applications Require Additional Performance Tuning

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

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

Xcode 4 provides new tools to help you improve the performance of your OpenGL ES applications.

Relevant Chapters "OpenGL ES Application Design Guidelines" (page 45), "Best Practices for Working with Vertex Data" (page 65), "Best Practices for Working with Texture Data" (page 78), "Best Practices for Shaders" (page 82), "Tuning Your OpenGL ES Application" (page 55), "Platform Notes" (page 88)

OpenGL ES May Not Be Used in Background Applications

Applications that are running in the background may not call OpenGL ES functions. If your application accesses the graphics processor while it is in the background, it is automatically terminated by iOS. To avoid this, your application 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 "Implementing a Multitasking-aware OpenGL ES Application" (page 43)

OpenGL ES Places Additional Restrictions on Multithreaded Applications

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

Relevant Chapters "Concurrency and OpenGL ES" (page 60)

How to Use This Document

If you are new to OpenGL ES or iOS development, begin by reading "OpenGL ES on iOS" (page 13), which provides an overview of the architecture of OpenGL ES and how it integrates into iOS. Read the remaining chapters in order.

Experienced iOS developers should focus on reading "Drawing With OpenGL ES" (page 28) to learn new details about how to integrate OpenGL ES into your application as well as how to correctly implement your rendering loop. Then, read "OpenGL ES Application Design Guidelines" (page 45) to dig deeper into how to design efficient OpenGL ES applications. "Platform Notes" (page 88) provides more detailed information on the available hardware and software available to your application.

Prerequisites

Before attempting to create an OpenGL ES application, you should already be familiar with views, and how views interact with Core Animation. See *View Programming Guide for iOS*.

Although this document introduces important OpenGL ES concepts, it is not a tutorial or a reference for the OpenGL ES API. To learn more about creating OpenGL ES applications, consult the references below.

See Also

OpenGL ES is an open standard defined by the Khronos Group. For more information about OpenGL ES 1.1 and 2.0, 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 application. 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 1.1 and OpenGL ES 2.0 specifications, the OpenGL ES shading language specification, and documentation for OpenGL ES extensions.
- OpenGL ES 1.1 Reference Pages provides a complete reference to the OpenGL ES 1.1 specification, indexed alphabetically.
- OpenGL ES 2.0 Reference Pages provides a complete reference to the OpenGL ES 2.0 specification, indexed alphabetically.
- *OpenGL ES Framework Reference* describes the platform-specific functions and classes provided by Apple to integrate OpenGL ES into iOS.

OpenGL ES on iOS

OpenGL ES provides a procedural API for submitting primitives to be rendered by a hardware accelerated graphics pipeline. Graphics commands are consumed by OpenGL to generate images that can be displayed to the user or retrieved for further processing outside of OpenGL ES.

The OpenGL ES specification defines the precise behavior for each function. Most commands in OpenGL ES perform one of the following activities:

- Reading detailed information provided by the implementation about its capabilities. See "Determining OpenGL ES Capabilities" (page 25).
- Reading and writing state variables defined by the specification. OpenGL ES state typically represents the current configuration of the graphics pipeline. For example, OpenGL ES 1.1 uses state variables extensively to configure lights, materials, and the computations performed by the fixed-function pipeline.
- Creating, modifying or destroying OpenGL ES objects. OpenGL ES objects are not Objective-C objects; they are OpenGL ES resources manipulated through OpenGL ES's procedural API. See "OpenGL ES Objects Encapsulate Resources on Behalf of Your Application" (page 17) for more information.
- Drawing primitives. Vertices are submitted to the pipeline where they are processed, assembled into primitives and rasterized to a **framebuffer**.

Which Version(s) of OpenGL ES Should I Target?

When designing your OpenGL ES application, a critical question you must answer is whether your application should support OpenGL ES 2.0, OpenGL ES 1.1 or both.

- OpenGL ES 2.0 is more flexible and more powerful than OpenGL ES 1.1 and is the best choice for new applications. Custom vertex or fragment calculations can be implemented more clearly and concisely in shaders, with better performance. To perform the same calculations in OpenGL ES 1.1 often requires multiple rendering passes or complex state configurations that obscure the intent of the algorithm. As your algorithms grow in complexity, shaders convey your calculations more clearly and concisely and with better performance. OpenGL ES 2.0 requires more work up front to build your application; you need to recreate some of the infrastructure that OpenGL ES 1.1 provides by default.

- OpenGL ES 1.1 provides a standard fixed-function pipeline that provides good baseline behavior for a 3D application, from transforming and lighting vertices to blending fragments into the framebuffer. If your needs are simple, OpenGL ES 1.1 requires less coding to add OpenGL ES support to your application. Your application should target OpenGL ES 1.1 if your application needs to support all iOS devices

If you choose to implement an OpenGL ES 1.1 application primarily for compatibility with older devices, consider adding an OpenGL ES 2.0 rendering option that takes advantage of the greater power of OpenGL ES 2.0-capable graphics processors found on newer iOS devices.

Important If your application does not support both an OpenGL ES 1.1 and an OpenGL ES 2.0 rendering path, restrict your application to devices that support the version you selected. See "Declaring the Required Device Capabilities" in *iOS App Programming Guide*.

Understanding the OpenGL ES Architecture

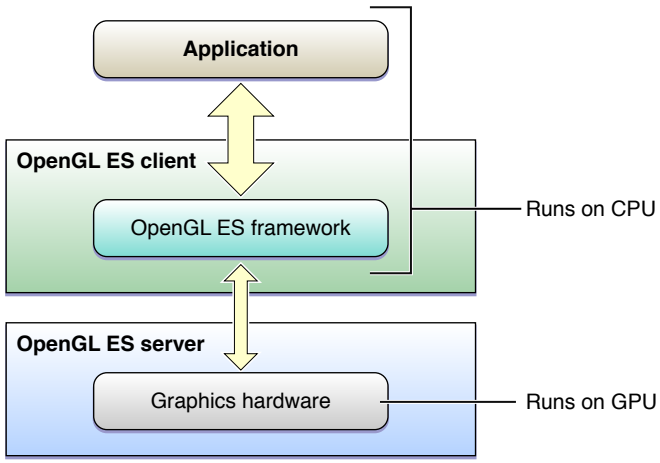
OpenGL ES works on a few key principles. To design efficient OpenGL ES applications, you need to understand the underlying architecture.

Client-Server Model

OpenGL ES uses a client-server model, as shown in Figure 1-1. When your application calls an OpenGL ES function, it talks to an OpenGL ES client. The client processes the function call and, where necessary, converts it into commands to deliver to the OpenGL server. A client-server model allows the work to process a function

call to be divided between the client and the server. The nature of the client, the server, and the communication path between them is specific to each implementation of OpenGL ES; on an iOS device, the workload is divided between the CPU and a dedicated graphics processor.

Figure 1-1 OpenGL ES client-server model



OpenGL ES Relies on Platform-Specific Libraries For Critical Functionality

The OpenGL ES specification defines how OpenGL ES works, but does not define functions to manage the interaction between OpenGL ES and the hosting operating system. Instead, the specification expects each implementation to provide functions to allocate **rendering contexts** and **system framebuffers**.

A rendering context stores the internal state for the OpenGL ES state machine. Rendering contexts allow each OpenGL ES application to each maintain a copy of the state data without worrying about other applications. See "Configuring OpenGL ES Contexts" (page 20). You can also use multiple rendering contexts in a single application.

A system framebuffer is a destination for OpenGL ES drawing commands, and is typically associated with the host operating system's graphics subsystem. iOS does not provide system framebuffers. Instead, iOS extends the framebuffer object provided by OpenGL ES to allow framebuffers that share data with Core Animation. See "Framebuffer Objects are the Only Rendering Target on iOS" (page 18) for more information on framebuffer objects and "Drawing With OpenGL ES" (page 28) for a detailed discussion on creating and using framebuffers in your application.

Commands May Be Executed Asynchronously

A benefit of the OpenGL ES client-server model is that an OpenGL ES function can return control to the application before the requested operation completes. If OpenGL ES required every function to complete before returning control to your application, the CPU and GPU would run in lockstep, eliminating many

opportunities for parallelism in your application. On iOS, deferring execution of drawing commands is quite common. By deferring several drawing commands and handling them simultaneously, the graphics hardware can remove hidden surfaces before performing costly fragment calculations.

Many OpenGL ES functions implicitly or explicitly flush commands to the graphics hardware. Other OpenGL functions flush commands to the graphics processor and wait until some or all pending commands have completed. Whenever possible, design your application to avoid client-server synchronizations.

Commands Are Executed In Order

OpenGL ES guarantees that the results of function calls made to a rendering context act as if they executed in the same order that the functions were called by the client application. When your application makes a function call to OpenGL ES, your application can assume the results from previous functions are available, even if some of the commands have not finished executing.

Parameters are Copied at Call-Time

To allow commands to be processed asynchronously, when your application calls an OpenGL ES function, any parameter data required to complete the function call must be copied by OpenGL ES before control is returned to your application. If a parameter points at an array of vertex data stored in application memory, OpenGL ES must copy the vertex data before returning. This has a couple of important implications. First, an application is free to change any memory it owns regardless of the calls it makes to OpenGL ES, because OpenGL ES and your application never access the same memory simultaneously. Second, copying parameters and reformatting them so that the graphics hardware can read the data adds overhead to every function call. For best performance, your application should define its data in format that are optimized for the graphics hardware, and it should use buffer objects to explicitly manage memory copies between your application and OpenGL ES.

Implementations May Extend the Capabilities Defined in the Specification

An OpenGL ES implementation may extend the OpenGL ES specification in one of two ways. First, the specification sets specific minimum requirements that implementations must meet, such as the size of a texture or the number of texture units that the application may access. An OpenGL ES implementation is free to support larger values — a larger texture, or more texture units. Second, OpenGL ES **extensions** allow an implementation to provide new OpenGL ES functions and constants. Extensions allow an implementation to add entirely new features. Apple implements many extensions to allow applications to take advantage of hardware features and to help you improve the performance of your applications. The actual hardware limits and the list of extensions offered by each implementation may vary depending on which device your application runs on and the version of iOS running on the device. Your application must test the capabilities at runtime and alter its behavior to match.

OpenGL ES Objects Encapsulate Resources on Behalf of Your Application

Objects are opaque containers that your application uses to hold configuration state or data needed by the renderer. Because the only access to objects is through the procedural API, the OpenGL ES implementation can choose different strategies when allocating objects for your application. It can store your data in a format or memory location that is optimal for the graphics processor. Another advantage to objects is that they are *reusable*, allowing your application to configure the object once and use it multiple times.

The most important OpenGL ES object types include:

- A **texture** is an image that can be sampled by the graphics pipeline. This is typically used to map a color image onto primitives but can also be used to map other data, such as a normal map or pre-calculated lighting information. The chapter "Best Practices for Working with Texture Data" (page 78) discusses critical topics for using textures on iOS.
- A **buffer** object is a block of memory owned by OpenGL ES used to store data for your application. Buffers are used to precisely control the process of copying data between your application and OpenGL ES. For example, if you provide a vertex array to OpenGL ES, it must copy the data every time you submit a drawing call. In contrast, if your application stores its data in a *vertex buffer object*, the data is copied only when your application sends commands to modify the contents of the vertex buffer object. Using buffers to manage your vertex data can significantly boost the performance of your application.
- A **vertex array** object holds a configuration for the vertex attributes that are to be read by the graphics pipeline. Many applications require different pipeline configurations for each entity it intends to render. By storing a configuration in a vertex array, you avoid the cost of reconfiguring the pipeline and may allow the implementation to optimize its handling of that particular vertex configuration.
- **Shader programs**, also known as shaders, are also objects. An OpenGL ES 2.0 application creates vertex and fragment shaders to specify the calculations that are to be performed on each vertex or fragment, respectively.
- A **renderbuffer** is a simple 2D graphics image in a specified format. This format usually is defined as color, depth or stencil data. Renderbuffers are not usually used in isolation, but are instead used as attachments to a framebuffer.
- **Framebuffer** objects are the ultimate destination of the graphics pipeline. A framebuffer object is really just a container that attaches textures and renderbuffers to itself to create a complete configuration needed by the renderer. A later chapter, "Drawing With OpenGL ES" (page 28), describes strategies for creating and using framebuffers in iOS applications.

Although each object type in OpenGL ES has its own functions to manipulate it, all objects share a similar programming model:

1. Generate an object identifier.

An identifier is a plain integer used to identify a specific object instance. Whenever you need a new object, call OpenGL ES to create a new identifier. Creating the object identifier does not actually allocate an object, it simply allocates a reference to it.

2. Bind your object to the OpenGL ES context.

   Most OpenGL ES functions act implicitly on an object, rather than requiring you to explicitly identify the object in every function call. You set the object to be configured by *binding* it to the context. Each object type uses different functions to bind it to the context. The first time you bind an object identifier, OpenGL ES allocates and initializes the object.

3. Modify the state of the object.

   Your application makes one or more function calls to configure the object. For example, after binding a texture object, you typically would configure how the texture is filtered and then load image data into the texture object.

   Changing an object can potentially be expensive, as it may require new data to be sent to the graphics hardware. Where reasonable, create and configure your objects once, and avoid changing them afterwards for the duration of your application.

4. Use the object for rendering.

   Once you've created and configured all the objects needed to render a scene, you bind the objects needed by the pipeline and execute one or more drawing functions. OpenGL ES uses the data stored in the objects to render your primitives. The results are sent to the bound framebuffer object.

5. Delete the object.

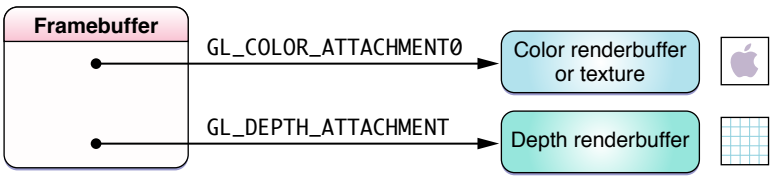
   When you are done with an object, your application should delete it. When an object is deleted, its contents are destroyed and the object identifier is recycled.

Framebuffer Objects are the Only Rendering Target on iOS

Framebuffer objects are the destination for rendering commands. OpenGL ES 2.0 provides framebuffer objects as part of the core specification; they are provided on OpenGL ES 1.1 by the `OES_framebuffer_object` extension. Because framebuffer objects are the only rendering target on iOS, Apple guarantees that the `OES_framebuffer_object` extension will always be provided by every OpenGL ES 1.1 implementation on iOS.

Framebuffer objects provide storage for color, depth and/or stencil data by attaching images to the framebuffer, as shown in Figure 1-2. The most common image attachment is a renderbuffer object. However, a OpenGL ES texture can be attached to the color attachment point of a framebuffer instead, allowing image to be rendered directly into a texture. Later, the texture can act as an input to future rendering commands.

Figure 1-2 Framebuffer with color and depth renderbuffers



Creating a framebuffer uses the following steps:

1. Generate and bind a framebuffer object.
2. Generate, bind, and configure an image.
3. Attach the image to one of the framebuffer's attachment points.
4. Repeat steps 2 and 3 for other images.
5. Test the framebuffer for completeness. The rules for completeness are defined in the OpenGL ES specification. These rules ensure the framebuffer and its attachments are well-defined.

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, it makes it easy for multiple applications to share the graphics hardware without interfering with the other's state.

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

An EAGL Context is the iOS Implementation of an OpenGL ES Rendering Context

Before your application can call any OpenGL ES functions, it must initialize an `EAGLContext` object and set it as the **current context**. The `EAGLContext` class also provides methods your application uses to integrate OpenGL ES content with Core Animation. Without these methods, your application would be limited to working with offscreen images.

The Current Context Acts as the Target for OpenGL ES Function Calls Made on a Thread

Every thread in an iOS application maintains a current context. When your application makes an OpenGL ES function call, that thread's context is modified by the call.

To set the current context, you call the `EAGLContext` class method `setCurrentContext:`.

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

Your application can retrieve a thread's current context by calling the `EAGLContext` class method `currentContext`.

When your application sets a new context, EAGL releases the previous context (if any) and retains the new context.

Note If your application 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 either OpenGL ES 1.1 or OpenGL ES 2.0, but never both. The reason for this lies in the design of OpenGL ES 2.0. OpenGL ES 2.0 removed all functions from OpenGL ES 1.1 that dealt with the fixed-function graphics pipeline and added many new functions to provide a clean shader-based interface. If your application attempts to call an OpenGL ES 2.0 function on an OpenGL ES 1.1 context (or vice versa), the results are undefined.

Your application decides which version of OpenGL ES to support when it creates and initializes the `EAGLContext` object. To create an OpenGL ES 2.0 context, your application would initialize it as shown:

```
EAGLContext* myContext = [[EAGLContext alloc]
initWithAPI:kEAGLRenderingAPIOpenGLES2];
```

If your application wants to use OpenGL ES 1.1 instead, it initializes it using a different constant:

```
EAGLContext* myContext = [[EAGLContext alloc]
initWithAPI:kEAGLRenderingAPIOpenGLES1];
```

If the device does not support the requested version of OpenGL ES, the `initWithAPI:` method returns `nil`. Your application should always test to ensure a context was initialized successfully before using it.

To support both an OpenGL ES 2.0 and OpenGL ES 1.1 rendering option, your application should first attempt to initialize an OpenGL ES 2.0 rendering context. If the returned object is `nil`, it should initialize an OpenGL ES 1.1 context instead. Listing 2-1 demonstrates how to do this.

Listing 2-1 Supporting OpenGL ES 1.1 and OpenGL ES 2.0 in the same application

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

```
        context = [[EAGLContext alloc] initWithAPI:kEAGLRenderingAPIOpenGLES1];
    }
    return context;
}
```

A context's `API` property states which version of OpenGL ES the context supports. Your application 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 application tests the context and creates a renderer once, on initialization.

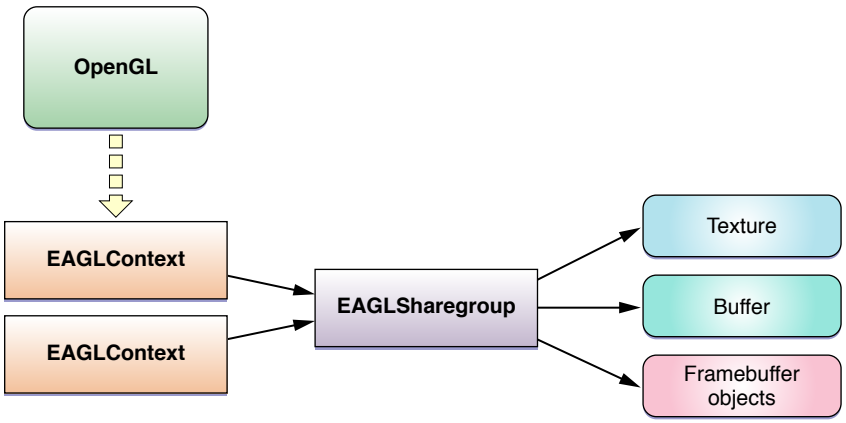
An EAGL Sharegroup Manages OpenGL ES Objects for the Context

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.

Your application can treat a sharegroup as an opaque object; it has no methods or properties that your application can call, and it is retained and released automatically by the referencing contexts.

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 application 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.

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 21). 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 application's responsibility to manage state changes to OpenGL ES objects when the sharegroup is shared by multiple contexts. Here are the rules:

- Your application 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 application 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 application 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 applications where you can carefully control the state of the context.

Determining OpenGL ES Capabilities

Both the OpenGL ES 1.1 and OpenGL ES 2.0 specifications define a minimum standard that every implementation must support. However, the OpenGL ES specification does not limit an implementation to just those capabilities. The OpenGL ES specification allows implementations to extend its capabilities in multiple ways. A later chapter, "Platform Notes" (page 88), drills down into the specific capabilities of each OpenGL ES implementation provided by iOS. The precise capabilities of an implementation may vary based on the version of the specification implemented, the underlying graphics hardware, and the version of iOS running on the device.

Whether you have chosen to build an OpenGL ES 1.1 or OpenGL ES 2.0 application, the first thing your application should do is determine the exact capabilities of the underlying implementation. To do this, your application sets a current context and calls one or more OpenGL ES functions to retrieve the specific capabilities of the implementation. The capabilities of the context do not change after it is created; your application can test the capabilities once and use them to tailor its drawing code to fit within those capabilities. For example, depending on the number of texture units provided by the implementation, you might perform your calculations in a single pass, perform them in multiple passes, or choose a simpler algorithm. A common pattern is to design a class for each rendering path in your application, with the classes sharing a common superclass. At runtime you instantiate the class that best matches the capabilities of the context.

Read Implementation-Dependent Values

The OpenGL ES specification defines implementation-dependent values that define limits of what an OpenGL ES implementation is capable of. For example, the maximum size of a texture and the number of texture units are both common implementation-dependent values that an application is expected to check. iOS devices that support the PowerVR MBX graphics hardware support textures up to 1024 x 1024 in size, while the PowerVR SGX software supports textures up to 2048 x 2048; both sizes greatly exceed 64 x 64, which is the minimum size required in the OpenGL ES specification. If your application's needs exceeds the minimum capabilities required by the OpenGL ES specification, it should query the implementation to check the actual capabilities of the hardware and fail gracefully; you may load a smaller texture or choose a different rendering strategy.

Although the specification provides a comprehensive list of these limitations, a few stand out in most OpenGL applications. Table 3-1 lists values that both OpenGL ES 1.1 and OpenGL ES 2.0 applications should test.

Table 3-1 Common OpenGL ES implementation-dependent values

Maximum size of the texture	GL_MAX_TEXTURE_SIZE

Number of depth buffer planes	GL_DEPTH_BITS
Number of stencil buffer planes	GL_STENCIL_BITS

In an OpenGL ES 2.0 application, your application primarily needs to read the limits placed on its shaders. All graphics hardware supports a limited number of attributes that can be passed into the vertex and fragment shaders. An OpenGL ES 2.0 implementation is not required to provide a software fallback if your application exceeds the values provided by the implementation; your application must either keep its usage below the minimum values in the specification, or it must check the shader limitations documented in Table 3-2 and choose shaders whose usage fits within those limits.

Table 3-2 OpenGL ES 2.0 shader values

Maximum number of vertex attributes	GL_MAX_VERTEX_ATTRIBS
Maximum number of uniform vertex vectors	GL_MAX_VERTEX_UNIFORM_VECTORS
Maximum number of uniform fragment vectors	GL_MAX_FRAGMENT_UNIFORM_VECTORS
Maximum number of varying vectors	GL_MAX_VARYING_VECTORS
Maximum number of texture units usable in a vertex shader	GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS
Maximum number of texture units usable in a fragment shader	GL_MAX_TEXTURE_IMAGE_UNITS

For all of the vector types, the query returns the number of 4-component floating-point vectors available.

OpenGL ES 1.1 applications should check the number of texture units and the number of available clipping planes, as shown in Table 3-3.

Table 3-3 OpenGL ES 1.1 values

Maximum number of texture units available to the fixed function pipeline	GL_MAX_TEXTURE_UNITS
Maximum number of clip planes	GL_MAX_CLIP_PLANES

Check for Extensions Before Using Them

An OpenGL ES implementation adds functionality to the OpenGL ES API by implementing OpenGL ES extensions. Your application must check for the existence of any OpenGL ES extension whose features it intends to use. The sole exception is the OES_framebuffer_object extension, which is always provided on all iOS implementations of OpenGL ES 1.1. iOS uses framebuffer objects as the only kind of framebuffer your application may draw into.

Listing 3-1 provides code you can use to check for the existence of extensions.

Listing 3-1 Checking for OpenGL ES extensions.

```
BOOL CheckForExtension(NSString *searchName)
{
// For performance, the array can be created once and cached.
    NSString *extensionsString = [NSString
stringWithCString:glGetString(GL_EXTENSIONS) encoding: NSASCIIStringEncoding];
    NSArray *extensionsNames = [extensionsString componentsSeparatedByString:@"
"];
    return [extensionsNames containsObject: searchName];
}
```

Call glGetError to Test for Errors

The debug version of your application should periodically call the `glGetError` function and flag any error that is returned. If an error is returned from the `glGetError` function, it means the application is using the OpenGL ES API incorrectly or the underlying implementation is not capable of performing the requested operation.

Note that repeatedly calling the `glGetError` function can significantly degrade the performance of your application. Call it sparingly in the release version of your application.

Drawing With OpenGL ES

This chapter digs into the process of creating framebuffers and rendering images into them. It describes different techniques for creating framebuffer objects, how to implement a rendering loop to perform animation, and working with Core Animation. Finally, it covers advanced topics such as rendering high-resolution images on Retina displays, using multisampling to improve image quality, and using OpenGL ES to render images on external displays.

Framebuffer Objects Store Rendering Results

The OpenGL ES specification requires that each implementation provide a mechanism that an application can use to create a **framebuffer** to hold rendered images. On iOS, all framebuffers are implemented using framebuffer objects, which are built-in to OpenGL ES 2.0, and provided on all iOS implementations of OpenGL ES 1.1 by the `GL_OES_framebuffer_object` extension.

Framebuffer objects allow your application to precisely control the creation of color, depth, and stencil targets. You can also create multiple framebuffer objects on an single context, possibly sharing resources between the frame buffers.

To correctly create a framebuffer:

1. Create a framebuffer object.
2. Create one or more targets (renderbuffers or textures), allocate storage for them, and attach each to an attachment point on the framebuffer object.
3. Test the framebuffer for completeness.

Depending on what task your application intends to perform, your application 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:

- If the framebuffer is used to perform offscreen image processing, attach a renderbuffer. See "Creating Offscreen Framebuffer Objects" (page 29)
- If the framebuffer image is used as an input to a later rendering step, attach a texture. "Using Framebuffer Objects to Render to a Texture" (page 30)

- If the framebuffer is intended to be displayed to the user, use a special Core Animation-aware renderbuffer. See "Rendering to a Core Animation Layer" (page 30)

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);
}
```

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.
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);
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).

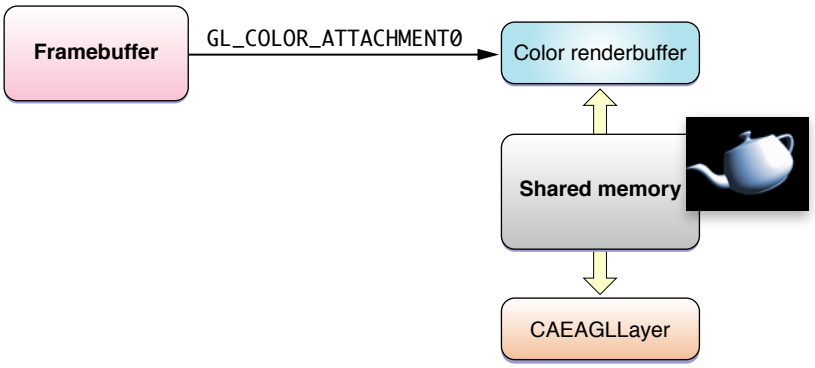
Although this example assumes you are rendering to a color texture, other options are possible. For example, using the OES_depth_texture extension, you can attach a texture to the depth attachment point to store depth information from the scene into a texture. You might use this depth information to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

Rendering to a Core Animation Layer

Most applications that draw using OpenGL ES want to display the contents of the framebuffer to the user. On iOS, all images displayed on the screen are handled by Core Animation. Every view is backed by a corresponding Core Animation layer. OpenGL ES connects to Core Animation through a special Core Animation layer, a `CAEAGLLayer`. A `CAEAGLLayer` allows the contents of an OpenGL ES renderbuffer to also act as the contents

of a Core Animation layer. This allows the renderbuffer contents to be transformed and composited with other layer content, including content rendered using UIKit or Quartz. Once Core Animation composites the final image, it is displayed on the device's main screen or an attached external display.

Figure 4-1 Core Animation shares the renderbuffer with OpenGL ES



In most cases, your application never directly allocates a `CAEAGLLayer` object. Instead, your application defines a subclass of `UIView` that allocates a `CAEAGLLayer` object as its backing layer. At runtime, your application instantiates the view and places it into the view hierarchy. When the view is instantiated, your application initializes an OpenGL ES context and creates a framebuffer object that connects to Core Animation.

The `CAEAGLLayer` provides this support to OpenGL ES by implementing the `EAGLDrawable` protocol. An object that implements the `EAGLDrawable` works closely with an `EAGLContext` object. A drawable object provides two key pieces of functionality. First, it allocates shared storage for a renderbuffer. Second, it works closely with the context to *present* that renderbuffer's content. Presenting the contents of a renderbuffer is loosely defined by EAGL; for a `CAEAGLLayer` object, presenting means that the renderbuffer's contents replace any previous contents presented to Core Animation. An advantage of this model is that the contents of the Core Animation layer do not need to be rendered every frame, only when the rendered image would actually change.

It is illustrative to walk through the steps used to create an OpenGL ES-aware view. The OpenGL ES template provided by Xcode implements this code for you.

1. Subclass `UIView` to create an OpenGL ES view for your iOS application.
2. Override the `layerClass` method so that your view creates a `CAEAGLLayer` object as its underlying layer. To do this, your `layerClass` method returns the `CAEAGLLayer` class.

```
+ (Class) layerClass
{
    return [CAEAGLLayer class];
```

```
}
```

3. In your view's initialization routine, read the `layer` property of your view. Your code uses this when it creates the framebuffer object.

```
myEAGLLayer = (CAEAGLLayer*)self.layer;
```

4. Configure the layer's properties.

   For optimal performance, mark the layer as opaque by setting the `opaque` property provided by the `CALayer` class to `YES`. See "Improving Compositing Performance in Core Animation" (page 38).

   Optionally, configure the surface properties of the rendering surface by assigning a new dictionary of values to the `drawableProperties` property of the `CAEAGLLayer` object. EAGL allows you to specify the pixel format for the renderbuffer and whether it retains its contents after they are presented to the Core Animation. For a list of the keys you can set, see *EAGLDrawable Protocol Reference*.

5. Allocate a context and make it the current context. See "Configuring OpenGL ES Contexts" (page 20).

6. Create the framebuffer object (see above).

7. Create a color renderbuffer. Allocate its storage by calling the context's `renderbufferStorage:fromDrawable:` method, 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 application should reallocate the renderbuffer's storage. If you do not reallocate the renderbuffers, the renderbuffer size won't match the size of the view; in this case, Core Animation may scale the image's contents to fit in the view. To prevent this, the Xcode template reallocates the framebuffer and renderbuffer whenever the view layout changes.

8. 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 was 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 application does this because the actual dimensions of the color renderbuffer are calculated based on the view'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 as well as to help determine the level of detail required in your application's textures and models. See "Supporting High-Resolution Displays Using OpenGL ES" (page 40).

9. Allocate and attach a depth buffer.
10. Test the framebuffer object.

To summarize, the steps to create the framebuffer are almost identical. Each allocates a color attachment and a depth attachment, differing primarily in how the color attachment is allocated.

Table 4-1 Mechanisms for allocating the color attachment of the framebuffer

Offscreen renderbuffer	`glRenderbufferStorage`
Drawable renderbuffer	`renderbufferStorage: fromDrawable:`
Texture	`glFramebufferTexture2D`

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 application uses the final frame.

Generally, applications render new frames in one of two situations:

- On demand; it renders a new frame when it recognizes that the data used to render the frame changed.
- In an animation loop; it assumes that data used to render the frame changes for every frame.

Rendering on Demand

Rendering on demand is appropriate when the data used to render a frame does not change very often, or only changes in response to user action. OpenGL ES on iOS is well suited to this model. When you present a frame, Core Animation caches the frame and uses it until a new frame is presented. By only rendering new frames when you need to, you conserve battery power on the device, and leave more time for the device to perform other actions.

> **Note** An OpenGL ES-aware view should not implement a `drawRect:` method to render the view's contents; instead, implement your own method to draw and present a new OpenGL ES frame and call it when your data changes. Implementing a `drawRect:` method causes other changes in how UIKit handles your view.

Rendering Using an Animation Loop

Rendering using an animation loop is appropriate when your data is extremely likely to change in every frame. For example, games and simulations rarely present static images. Smooth animation is more important than implementing an on-demand model.

On iOS, the best way to set up your animation loop is with a `CADisplayLink` object. A display link is a Core Animation object that synchronizes drawing to the refresh rate of a screen. This allows a smooth update to the screen contents, without stuttering or tearing. Listing 4-1 (page 34) shows how your view can retrieve the screen it is viewed on, use that screen to create a new display link object and add the display link object to the run loop.

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, your application should read the display link's `timestamp` property to get the time stamp for the next frame to be rendered. It can use that value to calculate the positions of objects in the next frame.

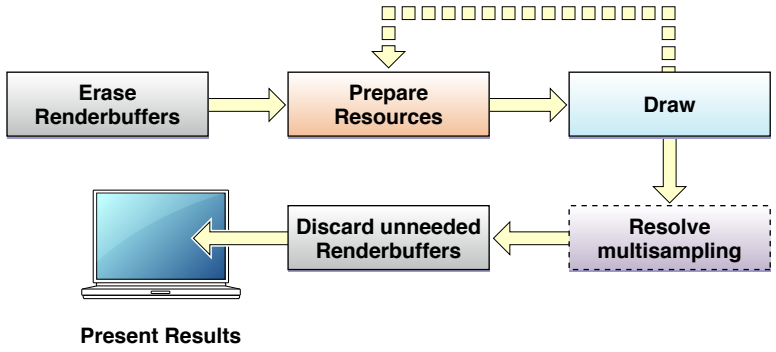
Normally, the display link object is fired every time the screen refreshes; that value is usually 60 hz, but may vary on different devices. Most applications do not need to update the screen sixty times per second. You can set the display link's `frameInterval` property to the number of actual frames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

Important For best results, choose a frame rate your application 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 4-2 (page 35) shows the steps an OpenGL ES application should take on iOS to render and present a frame. These steps include many hints to improve performance in your application.

Figure 4-2 iOS OpenGL Rendering Steps



Erase the Renderbuffers

At the start of every frame, erase all renderbuffers 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 (page 35).

Listing 4-2 Erasing the renderbuffers

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

Not only is using `glClear` more efficient than erasing the buffers manually, but using `glClear` hints to OpenGL ES that the existing contents can be discarded. On some graphics hardware, this avoids costly memory operations to load the previous contents into memory.

Prepare OpenGL ES Objects

This step and the next step is the heart of your application, 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 Application Design Guidelines" (page 45). For now, the most important performance optimization to note is that your application runs faster if it only modifies OpenGL ES objects at the start of rendering a new frame. Although your application 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 application uses multisampling to improve image quality, your application must resolve the pixels before they are presented to the user. Multisampling applications are covered in detail in "Using Multisampling to Improve Image Quality" (page 38).

Discard Unneeded Renderbuffers

A *discard* operation is defined by the `EXT_discard_framebuffer` extension and is available on iOS 4.0 and later. Discard operations should be omitted when your application is running on earlier versions of iOS, but included whenever they are available. A discard is a performance hint to OpenGL ES; it tells OpenGL ES that the contents of one or more renderbuffers are not used by your application after the discard command completes. By hinting to OpenGL ES that your application does not need the contents of a renderbuffer, the data in the buffers can be discarded or expensive tasks to keep the contents of those buffers updated can be avoided.

At this stage in the rendering loop, your application has submitted all of its drawing commands for the frame. While your application 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);
```

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 37) 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 application presents the renderbuffer. This means that every time your application presents a frame, it must completely recreate the frame's contents when it renders a new frame. The code above always erases the color buffer for this reason.

If your application 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 application's performance.

Supporting View Controllers in Your OpenGL ES Application

Many OpenGL ES applications are immersive; they take over the entire screen to display their content. However, those same applications often need to interact with other features of iOS. For example, applications that want to display iAd advertisements or use Game Center's built-in view controllers must provide a view controller. This view controller is used to modally display the contents provided by those system view controllers. For this reason, the Xcode template now provides a corresponding view controller to manage the view. Your application should do the same.

An important use for a view controller is to handle view rotations. On PowerVR SGX-equipped devices running iOS 4.2 and later, the performance of Core Animation rotation and scaling of OpenGL ES content has been significantly improved. Your application should use a view controller to set the allowed orientations and to transition between orientations when the user rotates the device, as described in *View Controller Programming Guide for iOS*. By using a view controller, other view controllers presented modally by your view controller are presented in the same orientation as your view controller.

If your application supports PowerVR MBX-equipped devices, on those devices you may need to avoid Core Animation; instead, perform the rotations directly inside OpenGL ES. When a rotation occurs, change the modelview and projection matrices and swap the width and height arguments to the `glViewport` and `glScissor` functions.

Improving Compositing Performance in Core Animation

The contents of renderbuffers are animated and composited along with any other Core Animation 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 application may perform too slowly to reach interactive frame rates. The performance penalties for mixing and matching content vary depending on the underlying graphics hardware on the iOS device; devices that use the PowerVR MBX graphics processor incur more severe penalties when compositing complex scenes. For best results, alway test your application on all iOS devices you intend it to ship on.

For the absolute best performance, your application should rely solely on OpenGL ES to render your content. To do this, size the view that holds your `CAEAGLLayer` object to match the screen, set its `opaque` property to `YES`, and ensure that no other Core Animation layers or views are visible. If your OpenGL ES layer 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.

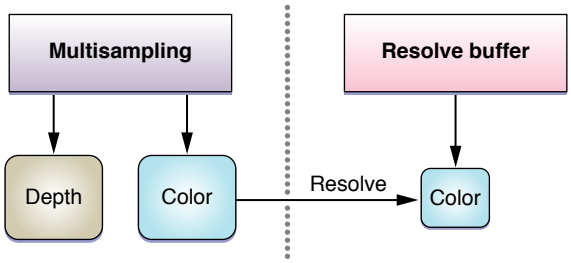
Using Multisampling to Improve Image Quality

Starting in iOS 4, OpenGL ES supports the `APPLE_framebuffer_multisample` extension. Multisampling is a form of **antialiasing**, allowing your application to smooth jagged edges and improve image quality. Multisampling uses more resources (memory and fragment processing), but improves image quality in most 3D applications.

Figure 4-3 (page 39) shows how multisampling works in concept. Instead of creating one framebuffer object, your application now creates two. The first framebuffer object is the **resolve buffer**; it contains a color renderbuffer, but otherwise is allocated exactly as you did before. The resolve buffer is where the final image is rendered. The second framebuffer object, the **multisampling buffer**, contains both depth and color attachments. 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 application 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 4-3 How multisampling works



Listing 4-5 (page 39) 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 Erase 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 Presentation step presents 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 application, always test your application's performance to ensure that it remains acceptable.

Supporting High-Resolution Displays Using OpenGL ES

If your application uses OpenGL ES for rendering, your existing drawing code should continue to work without any changes. When drawn on a high-resolution screen, though, your content is scaled accordingly and will appear as it does on non-Retina displays. The reason for this is that the default behavior of the `CAEAGLLayer`

class is to set the scale factor to `1.0`. To enable high-resolution drawing, you must change the scale factor of the view you use to present your OpenGL ES content. For more information on how high-resolution displays are supported in UIKit, see "Supporting High-Resolution Screens".

Changing the `contentScaleFactor` property of your view triggers a matching change to the scale factor of the underlying `CAEAGLLayer` object. The `renderbufferStorage:fromDrawable:` method, which you use to create the storage for the renderbuffer, calculates the size of the renderbuffer by multiplying the layer's bounds by its scale factor. The code provided in "Rendering to a Core Animation Layer" (page 30) allocates the renderbuffer's storage in Core Animation, then retrieves the width and height of renderbuffer; this allows it to automatically pick up any changes to the scale of its content. If you double the scale factor, the resulting renderbuffer has four times as many pixels to render. After that, it is up to you to provide the content to fill those additional pixels.

> **Important** A view that is backed by a `CAEAGLLayer` object should not implement a custom `drawRect:` method. Implementing a `drawRect:` method implicitly changes the default scale factor of the view to match the scale factor of the screen. If your drawing code is not expecting this behavior, your application content will not be rendered correctly.

If you opt in to high-resolution drawing, adjust the model and texture assets of your application accordingly. When running on an iPad or a high-resolution device, you might want to choose more detailed models and textures to render a better image. Conversely, on a standard-resolution iPhone, you can continue to use smaller models and textures.

An important factor when determining whether to support high-resolution content is performance. The quadrupling of pixels that occurs when you change the scale factor of your layer from 1.0 to 2.0 causes four times as many fragments to be processed. If your application performs many per-fragment calculations, the increase in pixels may reduce your application's frame rate. If you find your application 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 multisampling instead. An added advantage is that multisampling also provides higher quality on devices that do not support high-resolution displays.
- Create your color and depth renderbuffers using the lowest precision types that still provide good results; this reduces the memory bandwidth required to operate on the renderbuffers.

The best solution depends on the needs of your OpenGL ES application; during development, test these options and choose the approach that provides the best balance between performance and image quality.

Creating OpenGL ES Contexts on an External Display

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

The procedure for creating your context is almost identical to that running on the main screen. Follow these steps:

1. Follow the steps found in "Displaying Content on an External Display" in *View Programming Guide for iOS* to create a new window on the external display.
2. Add your OpenGL ES view to the window.
3. Create a display link object by retrieving the `screen` property of the window and calling its `displayLinkWithTarget:selector:`. This creates a display link optimized for that display.

Implementing a Multitasking-aware OpenGL ES Application

Starting in iOS 4, applications can continue to run when a user switches to another application. An application must be modified to run in the background; applications designed for iOS 3.x and earlier are terminated when a developer switches to another application. For an overall discussion of multitasking on iOS, see "iCloud Storage".

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

Background Applications May Not Execute Commands on the Graphics Hardware

An OpenGL ES application is terminated if it attempts to execute OpenGL ES commands on the graphics hardware. This not only refers to calls made to OpenGL ES while your application is in the background, but also refers to previously submitted commands that have not yet completed. The main reason for preventing background applications from processing OpenGL ES commands is to make the graphics processor completely available to the frontmost application. The frontmost application should always present a great experience to the user. Allowing background applications to hog the graphics processor might prevent that. Your application must ensure that all previously submitted commands have been finished prior to moving into the background.

Here's how to implement this in your application:

1. In your application delegate's `applicationWillResignActive:` method, your application should stop its animation timer (if any), place itself into a known good state, and then call the `glFinish` function.
2. In your application delegate's `applicationDidEnterBackground:` method, your application may want to delete some of its OpenGL ES objects to make memory and resources available to the foreground application. Call the `glFinish` function to ensure that the resources are removed immediately.
3. Once your application exits its `applicationDidEnterBackground:` method, it must not make any new OpenGL ES calls. If your application makes an OpenGL ES call, it is terminated by iOS.
4. In your application's `applicationWillEnterForeground:` method, recreate any objects your application disposed of and restart your animation timer.

To summarize, your application 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, it must avoid all use of OpenGL ES until it moves back to the foreground.

Delete Easily Recreated Resources Before Moving to the Background

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

- A user is playing your game and exits your application briefly to check their calendar. When they return to your game, the game's resources are still in memory, allowing your game to resume immediately.
- Your OpenGL ES application is in the background when the user launches another OpenGL ES application. If that application needs more memory than is available on the devices, your application is silently and automatically terminated without requiring your application to perform any additional work.

Your goal should be to design your application to be a good citizen. Your application should keep the time it takes to move to the foreground short while also reducing its memory footprint while it is in the background.

Here's how your application should handle the two scenarios:

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

A class of easy targets is the framebuffers your application allocates to hold rendering results. When your application 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 application's framebuffers is allocated, but is not useful. Also, the contents of the framebuffers are *transitory*; most applications recreate the contents of the framebuffer every time they render a new frame. This makes renderbuffers a memory-intensive resource that can be easily recreated—a good candidate for an object that can be disposed of when moving into the background.

OpenGL ES Application Design Guidelines

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

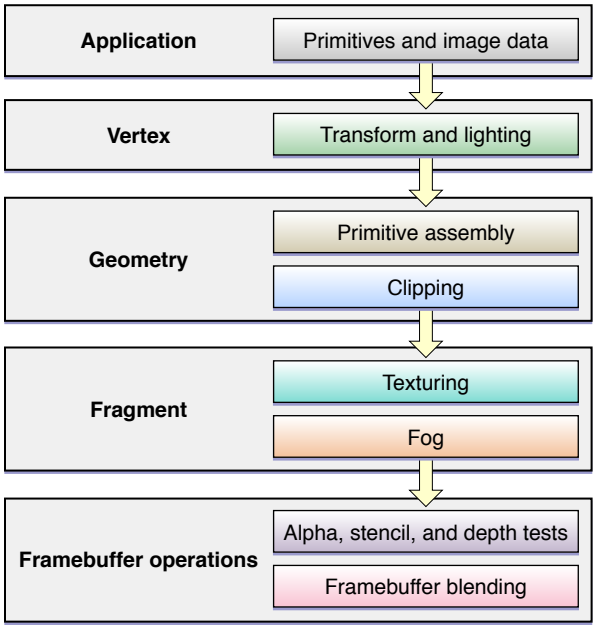
How to Visualize OpenGL ES

There are a few ways you can visualize OpenGL ES, and each provides a slightly different context in which to design and observe your application. The most common way to visualize OpenGL ES is as a graphics pipeline such as the one shown in Figure 6-1. Your application configures the graphics pipeline, and then executes one or more drawing commands. The drawing commands send vertex data down the pipeline, where it is processed, assembled into primitives, and rasterized into fragments. Each fragment calculates color and depth values which are then merged into the framebuffer. Using the pipeline as a mental model is essential for identifying exactly what work your application performs to generate a new frame. In a typical OpenGL ES 2.0 application, your design consists of writing customized shaders to handle the vertex and fragment stages of the pipeline. In an OpenGL ES 1.1 application, you modify the state machine that drives the fixed-function pipeline to perform the desired calculations.

Another benefit of the pipeline model is that individual stages can calculate their results independently and simultaneously. This is a key point. Your application might prepare new primitives while separate portions of the graphics hardware perform vertex and fragment calculations on previously submitted geometry. If any pipeline stage performs too much work or performs too slowly, other pipeline stages sit idle until the slowest stage completes its work. Your design needs to balance the work performed by each pipeline stage by matching calculations to the capabilities of the graphics hardware on the device.

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

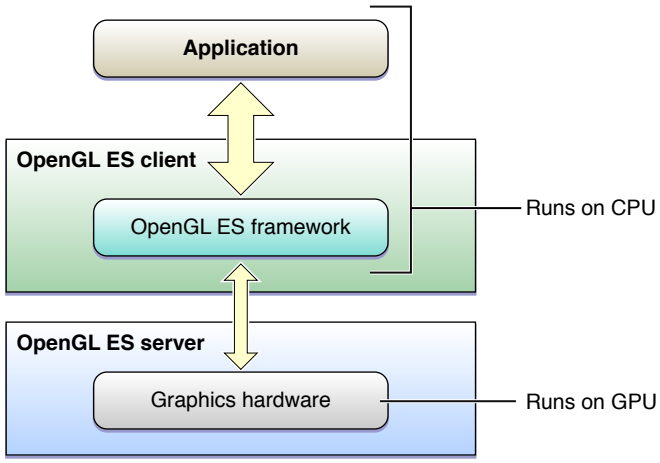
Figure 6-1 OpenGL ES graphics pipeline



Another way to visualize OpenGL ES is as a client-server architecture, as shown in Figure 6-2. OpenGL ES state changes, texture and vertex data, and rendering commands all have to travel from the application to the OpenGL ES client. The client transforms these data into a format that the graphics hardware understands, and forwards them to the GPU. Not only do these transformations add overhead, but the process of transferring the data to the graphics hardware takes time.

To achieve great performance, an application must reduce the frequency of calls it makes to OpenGL ES, minimize the transformation overhead, and carefully manage the flow of data between itself and OpenGL ES.

Figure 6-2 OpenGL client-server architecture



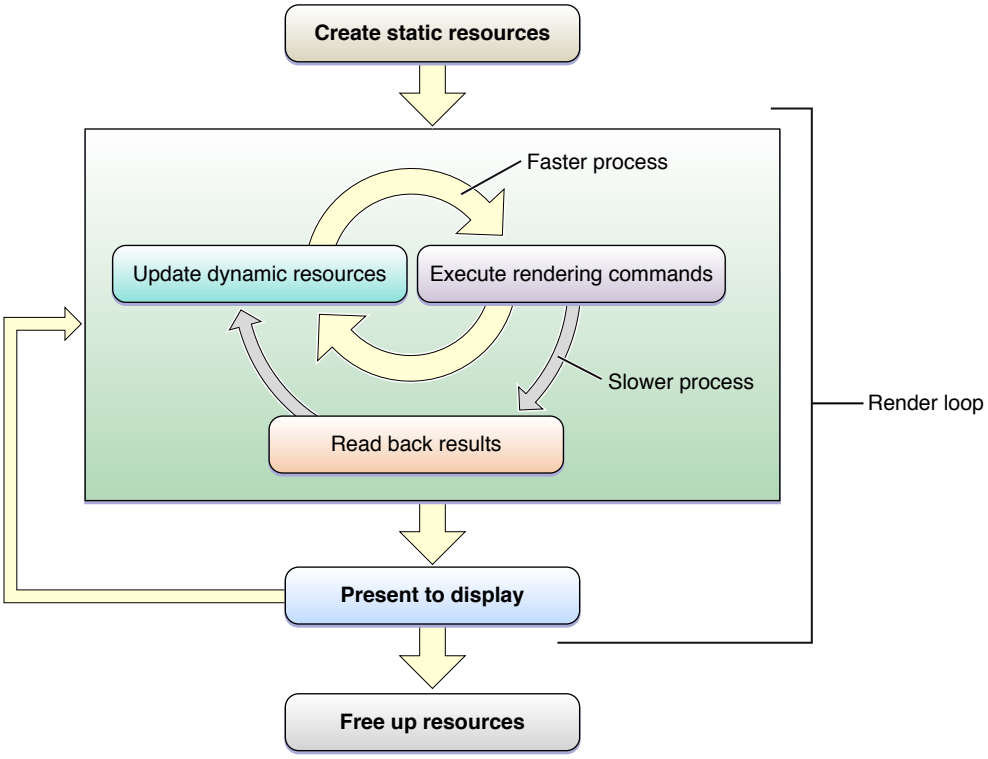
Designing a High-Performance OpenGL ES Application

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

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

Figure 6-3 suggests a process flow for an application that uses OpenGL ES to perform animation to the display.

Figure 6-3 Application model for managing resources



When the application launches, the first thing it does is initialize resources that it does not intend to change over the lifetime of the application. Ideally, the application encapsulates those resources into OpenGL ES objects. The goal is to create any object that can remain unchanged for the runtime of the application (or even a portion of the application'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 6-3, the application 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 application 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 application. 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 application blocks until all previously submitted drawing commands are completed.

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

Finally, when your application 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 applications.

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 application.

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

OpenGL ES is not required to execute most 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 application has queued up a significant number of commands before sending the buffer to the hardware—allowing the graphics hardware to execute commands in batches is often more efficient. However, some OpenGL ES functions must flush the buffer immediately. Other functions not only flush the buffer, but also block until previously submitted commands have completed before returning control over the application. Your application should

restrict the use of flushing and synchronizing commands only to those cases where that behavior is necessary. Excessive use of flushing or synchronizing commands may cause your application to stall waiting 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 hardare. It blocks until commands are submitted to the hardware but does not wait for the commands to finish executing.
- The function `glFinish` waits for all previously submitted commands to finish executing on the graphics hardware.
- Functions that retrieve OpenGL state (such as `glGetError`), also wait for submitted commands to complete.
- The command buffer is full.

Using glFlush Effectively

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

- If your application submits rendering commands that use a particular OpenGL ES object, and it intends to modify that object in the near future (or vice versa). If you attempt to modify an OpenGL ES object that has pending drawing commands, your application may stall until those drawing commands are completed. In this situation, calling `glFlush` ensures that the hardware begins processing commands immediately. After flushing the command buffer, your application should perform work that can operate in parallel with the submitted commands.
- When two contexts share an OpenGL ES object. After submitting any OpenGL ES commands that modify the object, call `glFlush` before switching to the other context.
- If multiple threads are accessing the same context, only one thread should send commands to OpenGL ES at a time. After submitting commands it must call `glFlush`.

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 that you can retrieve with the function `glGetError`. During development, it's crucial that your code contains error checking routines that call `glGetError`. If you are developing a performance-critical application, retrieve error information only while debugging your application. Calling `glGetError` excessively in a release build degrades performance.

Allow OpenGL ES to Manage Your Resources

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

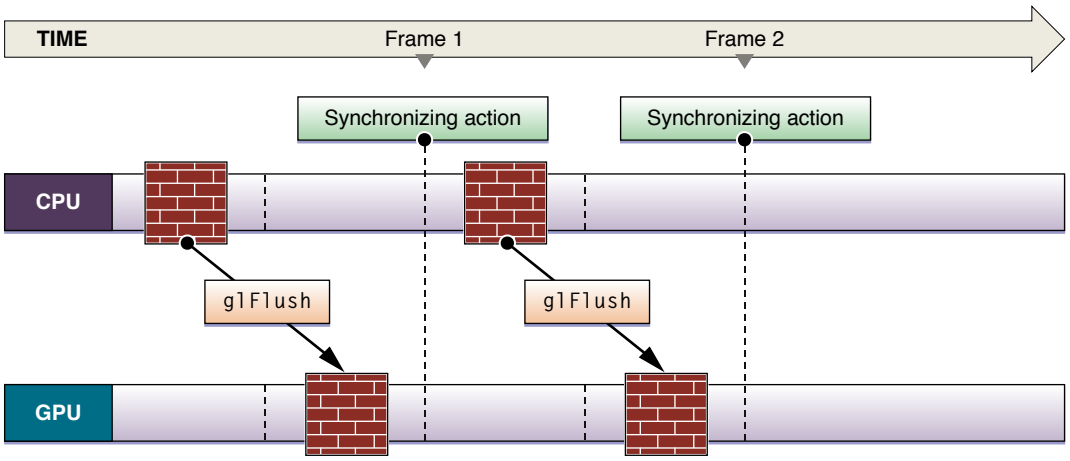
OpenGL ES allows your application to hint how it intends to use the data. These hints allow OpenGL ES to make an informed choice of how to process your data. For example, static data might be placed in memory that the graphics processor can readily fetch, or even into dedicated graphics memory.

Use Double Buffering to Avoid Resource Conflicts

Resource conflicts occur when your application 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 is not allowed to 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 application. Figure 6-4 shows this problem. In this example, there is a single texture

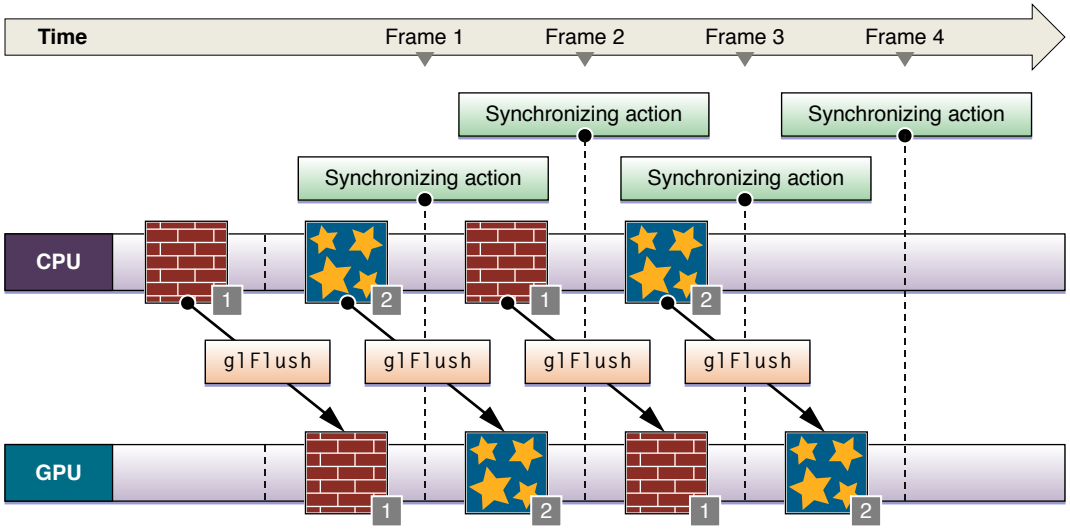
object, which both OpenGL ES and your application want to use. When the application attempts to change the texture, it must wait until previously submitted drawing commands complete—the CPU synchronizes to the GPU.

Figure 6-4 Single-buffered texture data



To solve this problem, your application could perform additional work between changing the object and drawing with it. But, if your application 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 6-5 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 6-5 Double-buffered texture data



Double buffering is sufficient for most applications, 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 application. On an iOS device, memory can be scarce; your design may need to balance using more memory with other application optimizations.

Be Mindful of OpenGL ES State Variables

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

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

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

If you are drawing 2D images, disable all irrelevant state variables, similar to what's shown in Listing 6-1.

Listing 6-1 Disabling state variables on OpenGL ES 1.1

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

Replace State Changes with OpenGL ES Objects

The "Be Mindful of OpenGL ES State Variables" (page 53) section suggests that reducing the number of state changes can improve performance. Some OpenGL ES extensions also allow you to create objects that collect multiple OpenGL state changes into an object that can be bound with a single function call. Where such techniques are available, they are recommended. For example, configuring the fixed-function pipeline requires many function calls to change the state of the various operators. Not only does this incur overhead for each function called, but the code is more complex and difficult to manage. Instead, use a shader. A shader, once compiled, can have the same effect but requires only a single call to `glUseProgram`.

For another example, vertex array objects allow you configure your vertex attributes once and store them in a vertex array object. See "Consolidate Vertex Array State Changes Using Vertex Array Objects" (page 75).

Tuning Your OpenGL ES Application

The performance of OpenGL ES applications in iOS differs from that of OpenGL in Mac 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 will touch on many techniques to improve your application's performance; this chapter talks about overall strategies you may want to follow in your application.

General Performance Recommendations

Test Your Application With Xcode and Use Common Sense to Guide Your Efforts

Don't optimize your application until you have tested its performance under a variety of scenarios on a variety of devices. Also, use common sense and avoid unnecessary optimizations. For example, it is unlikely that your application is bottlenecked inside OpenGL ES if it only draws a few dozen triangles per frame. Rewriting how your application submits the vertex data in such an application is unlikely to improve your application's performance.

Xcode provides many tools to help you analyze and tune your OpenGL ES applications:

- **OpenGL ES Performance Detective** — First introduced in Xcode 4, OpenGL ES Performance Detective quickly helps you to determine if OpenGL ES is the main bottleneck in your application; it should be the first tool you run to test your OpenGL ES code. A key advantage of OpenGL ES Performance Detective is that it can automatically direct you immediately to the critical location in your application that slows OpenGL ES performance the most. To use the OpenGL ES Performance Detective, launch the tool and use it to select your application on the iOS-based device connected to your development machine. When your application reaches a location in your application that is of interest to you, click the Collect Evidence button. OpenGL ES Performance Detective records OpenGL ES commands your application generates, analyzes the commands to discover the key bottlenecks, and delivers specific performance recommendations.

The OpenGL ES Performance Detective can be found inside the `/Developer/Applications/Graphics Tools/` directory.

- **Instruments (OpenGL ES Analysis)** — Also introduced in Xcode 4, the OpenGL ES Analysis tool provides a number of features to help you study your application's usage of OpenGL ES. The OpenGL ES Analysis tool records the OpenGL ES commands generated by your application and warns you when your application does not follow the best practices described in this programming guide; it recommends specific changes you can make to follow the best practices. The OpenGL ES Analysis tool allows you to see all the commands used to generate each frame of animation. Finally, the OpenGL ES Analysis tool allows you to selectively disable portions of the graphics pipeline to determine if that part of the pipeline is a significant bottleneck in your application.

  The OpenGL ES Analysis tool provides you a great set of tools to manually analyze your application and understand its inner workings. It does not, however, automatically point you at the location where your application is currently bottlenecked. For example, even when it offers a suggestion on how to improve your OpenGL ES coding practices, that suggestion does not mean that changing your code is automatically going to improve the performance of your application.

  For more information about Instruments, see *Instruments User Guide*.

- **Instruments (OpenGL ES Driver)** — The OpenGL ES Driver tool is provided on Xcode 3 and later. It does not directly analyze the OpenGL ES commands submitted by your application. Instead, it allows you to monitor key statistics about how the graphics hardware is utilized by your application. For example, you can use it to track the number of bytes used to hold texture data and how those numbers change from frame to frame.

  For more information about Instruments, see *Instruments User Guide*.

Redraw Scenes Only When The Scene Data Changes

Your application 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.

iOS Devices Support Native Floating-Point Arithmetic

3D applications (especially games) require physics, collision detection, lighting, animation, and other processing to create a compelling and interesting 3D world. All of these tasks boil down to a collection of mathematical functions that are evaluated for every frame. This imposes a serious amount of arithmetic on the CPU.

The ARM processor in iOS devices processes floating-point instructions natively. Your application should use floating-point instead of fixed point math whenever possible. If you are porting an application that uses fixed-point operations, rewrite the code to use floating-point types.

> **Note** iOS devices all support both ARM and Thumb instruction sets. Although Thumb reduces the code size, be sure to use ARM instructions for floating-point intensive code for better performance. To turn off the default Thumb setting in Xcode, open the project properties and deselect the Compile for Thumb build setting.

Disable Unused OpenGL ES Features

Whether you are using the fixed-function pipeline of OpenGL ES 1.1 or shaders in OpenGL ES 2.0, the best calculation is one that your application never performs. For example, if a calculation can be pre-calculated and stored in your model data, you can avoid performing that calculation at runtime.

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

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

Minimize the Number of Draw Calls

Every time your application submits primitives to be processed by OpenGL ES, the CPU spends time preparing the commands for the graphics hardware. To reduce this overhead, batch your drawing into fewer calls. For example, you might merge multiple triangle strips into a single strip, as described in "Use Triangle Strips to Batch Vertex Data" (page 69).

Consolidating models to use a common set of OpenGL state has other advantages in that it reduces the overhead of changing OpenGL ES state. See "Be Mindful of OpenGL ES State Variables" (page 53).

For best results, consolidate primitives that are drawn in close spacial proximity. Large, sprawling models are more difficult for your application to efficiently cull when they are not visible in the frame.

Memory is a Scarce Resource on iOS Devices

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

- After loading an image into an OpenGL ES texture, free the original image.
- Allocate a depth buffer only when your application requires it.
- If your application 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 application the memory it needs. Your application should strive to use as little memory as possible and be prepared to release cached data that is not essential to your application. Responding to low-memory conditions is covered in detail in the *iOS App Programming Guide*.

The PowerVR MBX processor found in some iOS devices has additional memory restrictions. See "PowerVR MBX" (page 94) for more information.

Do Not Sort Rendered Objects Except Where Necessary For Correct Rendering

- Do not waste time sorting objects front to back. OpenGL ES on all iOS devices implements a tile-based deferred rendering model that makes this unnecessary. See "Tile-Based Deferred Rendering" (page 89) for more information.
- Do sort objects by their opacity:
  1. Draw opaque objects first.
  2. Next draw objects that require alpha testing (or in an OpenGL ES 2.0 based application, objects that require the use of `discard` in the fragment shader). Note that these operations have a performance penalty, as described in "Avoid Alpha Test and Discard" (page 59).
  3. Finally, draw alpha-blended objects.

Simplify Your Lighting Models

This advice applies 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.

- Use the fewest lights possible and the simplest lighting type for your application. Consider using directional lights instead of spot lighting, which require more calculations. OpenGL ES 2.0 calculations 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.

Avoid Alpha Test and Discard

Graphics hardware often performs depth testing early in the graphics pipeline, before calculating the fragment's color value. If your application uses an alpha test in OpenGL ES 1.1 or the `discard` instruction in an OpenGL ES 2.0 fragment shader, some hardware depth-buffer optimizations must be disabled. In particular, this may require a fragment's color to be completely calculated only to be discarded because the fragment is not visible.

An alternative to using alpha test or discard to kill pixels is to use alpha blending with alpha forced to zero. This effectively eliminates any contribution to the framebuffer color while retaining the Z-buffer optimizations. This does change the value stored in the depth buffer and so may require back-to-front sorting of the transparent primitives.

If you need to use alpha testing or a `discard` instruction, draw these objects separately in the scene after processing any primitives that do not require it. Place the `discard` instruction early in the fragment shader to avoid performing calculations whose results are unused.

Concurrency and OpenGL ES

Concurrency is the notion of multiple things happening at the same time. In the context of computers, concurrency usually refers to executing tasks on more than one processor at the same time. By performing work in parallel, tasks complete sooner, and applications become more responsive to the user. A well-designed OpenGL ES application already exhibits a specific form of concurrency—concurrency between application processing on the CPU and OpenGL ES processing on the GPU. Many of the techniques introduced in "OpenGL ES Application Design Guidelines" (page 45) are aimed specifically at creating OpenGL applications that exhibit great CPU-GPU parallelism. Designing a concurrent application means decomposing the work your application performs into subtasks and identifying which tasks can safely operate in parallel and which tasks must be executed sequentially—that is, which tasks are dependent on either resources used by other tasks or results returned from those tasks.

Each process in iOS is made up 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)**. Grand Central Dispatch allows you to decompose your application into smaller tasks without requiring the application to manage threads. GCD allocates threads based on the number of cores available on the 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 attempt to describe these technologies in detail. Before you consider how to add concurrency to your OpenGL ES application, first read *Concurrency Programming Guide*. If you plan on managing threads manually, also read *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 application's performance, the restrictions OpenGL ES places on multithreaded applications, and common design strategies you might use to implement concurrency in an OpenGL ES application.

Identifying Whether an OpenGL Application Can Benefit from Concurrency

Creating a multithreaded application requires significant effort in the design, implementation, and testing of your application. Threads also add complexity and overhead to an application. Your application may need to copy data so that it can be handed to a worker thread, or multiple threads may need to synchronize access to

the same resources. Before you attempt to implement concurrency in an OpenGL ES application, first optimize your OpenGL ES code in a single-threaded environment using the techniques described in "OpenGL ES Application Design Guidelines" (page 45). Focus on achieving great CPU-GPU parallelism first and then assess whether concurrent programming can provide an additional performance benefit.

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

- The application performs many tasks on the CPU that are independent of OpenGL 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 application 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 application is incapable of feeding it commands fast enough. If your CPU-bound code has already been optimized, you may be able to improve its performance further by splitting the work into tasks that execute concurrently.

If your application is blocked waiting for the GPU, and has no work it can perform in parallel with its OpenGL 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 application 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 application might crash or it might render improperly. If for some reason you decide to set more than one thread to target the same context, then you must synchronize threads by placing a mutex around all OpenGL 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 application, that means:

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

Strategies for Implementing Concurrency in OpenGL ES Applications

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

- Decompose your application into OpenGL 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 application has other tasks that require significant CPU processing.
- If your application spends a lot of CPU time preparing data to send to OpenGL ES, you can divide the work between tasks that prepare rendering data and tasks that submit rendering commands to OpenGL ES. See "OpenGL ES Restricts Each Context to a Single Thread" (page 61)
- If your application 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 "An EAGL Sharegroup Manages OpenGL ES Objects for the Context" (page 22).

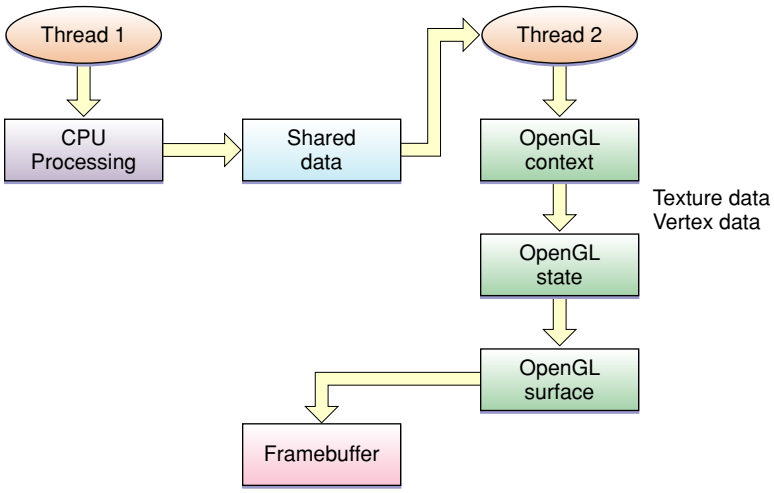
Perform OpenGL ES Computations in a Worker Task

Some applications perform lots of calculations on their data before passing the data down to OpenGL ES. For example, the application 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 application and OpenGL ES.

The approach described in Figure 6-3 (page 48) alternates between updating OpenGL ES objects and executing rendering commands that use those objects. OpenGL ES renders on the GPU in parallel with your application's updates running on the CPU. If the calculations performed on the CPU take more processing time than those on the GPU, then the GPU spends more time idle. In this situation, you may be able to take advantage of

parallelism on systems with multiple CPUs. Split your OpenGL ES rendering code into separate calculation and processing tasks, and run them in parallel. Figure 8-1 shows a clear division of labor. One task produces data that is consumed by the second and submitted to OpenGL.

Figure 8-1 CPU processing and OpenGL on separate threads



For best performance, your application should avoid copying data between the 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 your application 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

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

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 application would create two contexts and two threads; each thread controls one context. Further, any OpenGL ES objects your application 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 22).

Guidelines for Threading OpenGL ES Applications

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

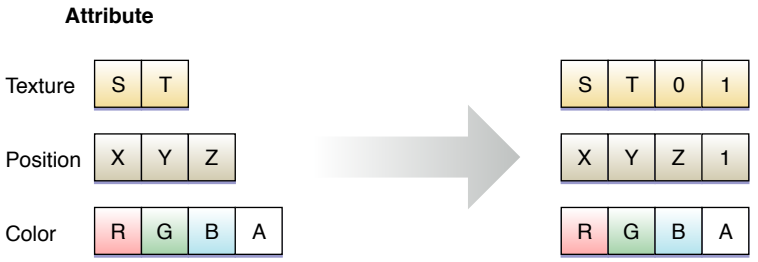
- Use only one thread per context. OpenGL ES commands for a specific context are not thread safe. Never have more than one thread accessing a single context simultaneously.
- When using GCD, use a dedicated serial queue to dispatch commands to OpenGL ES; this can be used to replace the conventional mutex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

Best Practices for Working with Vertex Data

To render a frame using OpenGL ES your application configures the graphics pipeline and submits graphics primitives to be drawn. In some applications, all primitives are drawn using the same pipeline configuration; other applications may render different elements of the frame using different techniques. But no matter which primitives you use in your application or how the pipeline is configured your application 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 application 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 application uses attributes defined by the fixed-function pipeline.

Your application defines 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 GLbyte components (alpha, red, green, blue). When an attribute is loaded into a shader variable, any components that are not provided in the application data are filled in with default values by OpenGL ES. The last component is filled with `1`, while other unspecified components are filled with `0`.



Your application 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 application calls a function in OpenGL ES to draw a set of vertices, the vertex data is copied from your application 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.

Applications 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 application uses OpenGL ES 2.0, then your application 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 application is blocked on vertex processing, the calculation is inexpensive and the vertex count can be significantly reduced by the change.

- If your application 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 applications can either set a constant vertex attribute or use a uniform shader value to hold the value instead. OpenGL ES 1.1 application 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:

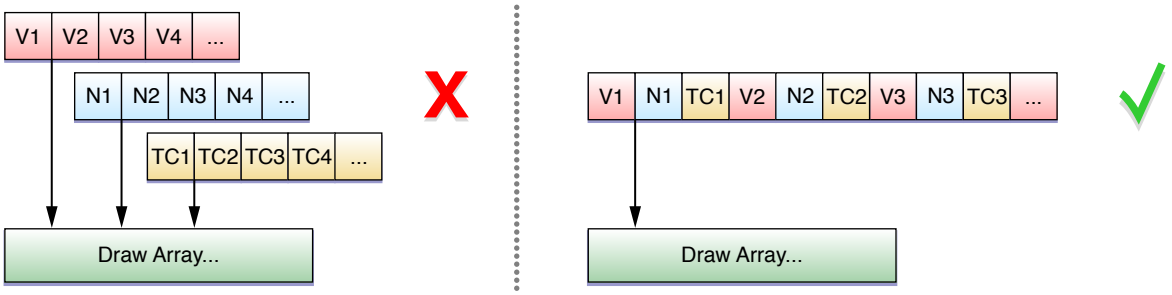
- Specify vertex colors using four unsigned byte components (`GL_UNSIGNED_BYTE`).
- Specify texture coordinates using 2 or 4 unsigned bytes (`GL_UNSIGNED_BYTE`) or unsigned short (`GL_UNSIGNED_SHORT`). Do not pack multiple sets of texture coordinates into a single attribute.
- Avoid using the OpenGL 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.

If you specify smaller components, be sure you reorder your vertex format to avoid misaligning your vertex data. See "Avoid Misaligned Vertex Data" (page 68).

Use Interleaved Vertex Data

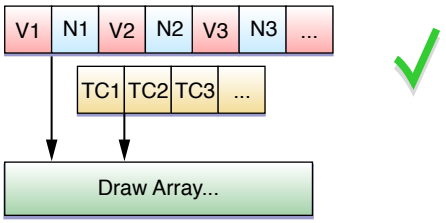
OpenGL ES allows your application to either specify vertex data as a series of arrays (also known as a *struct of arrays*) or as an array where each element includes multiple attributes (an *array of structs*). The preferred format on iOS is an array of structs with a single interleaved vertex format.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

Figure 9-1 Interleaved memory structures place all data for a vertex together in memory



An exception to this rule is when your application 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 9-2 Use multiple vertex structures when some data is used differently

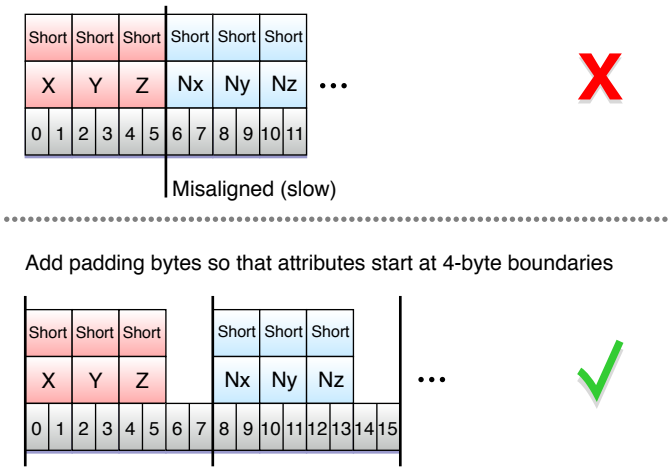


Avoid Misaligned Vertex Data

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 9-3 (page 69), 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.

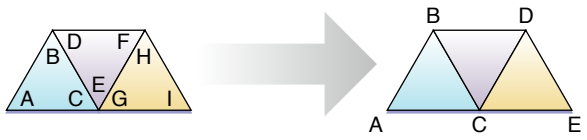
Figure 9-3 Align Vertex Data to avoid additional processing



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 9-4, 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 9-4 Triangle strip

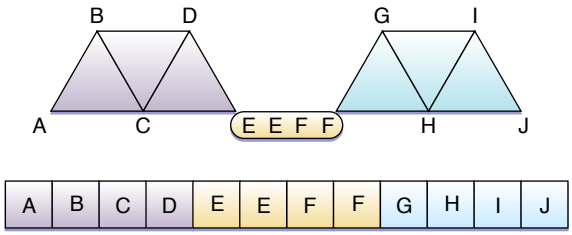


Sometimes, your application can combine more than one triangle strip into a single larger triangle strip. All of the strips must share the same rendering requirements. This means:

- An OpenGL ES 2.0 application must use the same shader to draw all of the triangle strips.
- An OpenGL ES 1.1 application 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 9-5. When this strip is submitted to OpenGL ES, triangles DEE, EEF, EFF, and FFG are considered degenerate and not processed or rasterized.

Figure 9-5 Use degenerate triangles to merge triangle strips



For best performance, your models should be submitted as a single unindexed triangle strip using `glDrawArrays` with as few duplicated vertices as possible. If your models require many vertices to be duplicated (because many vertices are shared by triangles that do not appear sequentially in the triangle strip or because your application merged many smaller triangle strips), you may obtain better performance using a separate index buffer and calling `glDrawElements` instead. There is a trade off: an unindexed triangle strip must periodically duplicate entire vertices, while an indexed triangle list requires additional memory for the indices and adds overhead to look up vertices. For best results, test your models using both indexed and unindexed triangle strips, and use the one that performs the fastest.

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, so locality of reference may allow the hardware to avoid calculating a vertex multiple times.

Use Vertex Buffer Objects to Manage Copying Vertex Data

Listing 9-1 provides a function that a simple application 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 9-1 Submitting vertex data to OpenGL ES 2.0

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

```
enum {
   ATTRIB_POSITION,
   ATTRIB_COLOR,
   NUM_ATTRIBUTES };


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


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


    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 application 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.

Listing 9-2 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 application is initialized.

Listing 9-2 Creating vertex buffer objects

```
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 9-3 modifies Listing 9-1 (page 70) to use the vertex buffer objects. The key difference in Listing 9-3 is that the parameters to the `glVertexPointer` and `glColorPointer` functions no longer point to the vertex arrays. Instead, each is an offset into the vertex buffer object.

Listing 9-3 Drawing using Vertex Buffer Objects in OpenGL ES 2.0

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

    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 assumed that the vertex data is initialized once, and never needs to change during rendering. However, OpenGL ES allows you to change the data in a buffer at runtime. A key part of the design of vertex buffer objects is that the application can inform OpenGL ES how it uses the data stored in the buffer. The usage parameter allows OpenGL ES to choose different strategies for storing the buffer depending on how your application intends to use the data. In Listing 9-2 (page 71), 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.

OpenGL ES supports the following usage cases:

- `GL_STATIC_DRAW` should be used for vertex data that is specified once and never changes. Your application should create these vertex buffer objects during initialization and use them until they are no longer needed.
- `GL_DYNAMIC_DRAW` should be used when data stored in the buffer may change during the rendering loop. Your application should still initialize these buffers during initialization, and update the data as needed by calling the `glBufferSubData` function.
- `GL_STREAM_DRAW` is used when your application needs to create transient geometry that is rendered a small number of times and then discarded. This is most useful when your application must dynamically calculate new vertex data every frame and you perform the calculations inside the shader. Typically, if your application is using stream drawing, you want to create two different buffers; while OpenGL ES is drawing using the contents of one buffer, your application is filling the other. See "Use Double Buffering to Avoid Resource Conflicts" (page 51). Stream drawing is not available in OpenGL ES 1.1; use `GL_DYNAMIC_DRAW` instead.

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 9-4 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 application maintains reasonable performance.

Listing 9-4 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);
}
```

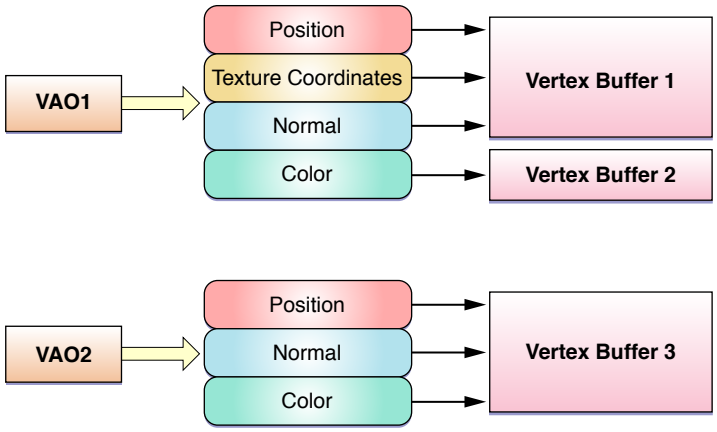
```
void DrawModelUsingMultipleVertexBuffers()
{
    glBindBuffer(GL_ARRAY_BUFFER, staticBuffer);
    glVertexAttribPointer(ATTRIB_POSITION, 2, GL_FLOAT, GL_FALSE,
sizeof(vertexStruct), (void*)offsetof(vertexStruct,position));
    glEnableVertexAttribArray(ATTRIB_POSITION);

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

    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 9-4 (page 73). 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 application renders a frame, there's a lot of unnecessary overhead reconfiguring the graphics pipeline. If the application draws many different kinds of models, reconfiguring the pipeline may become a real bottleneck. To avoid this, use a vertex array object to store a complete attribute configuration. The `OES_vertex_array_object` extension is available on all iOS devices starting in iOS 4.0.

Figure 9-6 shows an example configuration with two vertex array objects. Each configuration is independent of the other; each points to a different number of vertices and even into different vertex buffer objects.

Figure 9-6 Vertex array object configuration



Listing 9-5 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 9-5 Configuring a vertex array object

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

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

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

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

For best performance, your application 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 application 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.

Best Practices for Working with Texture Data

Texture data is often the largest portion of the data your application 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 application, you must manage your application's textures carefully. To summarize the guidelines:

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

Reduce Texture Memory Usage

Reducing the amount of memory your iOS application uses is always an important part of tuning your application. However, an OpenGL ES application is also constrained in the total amount of memory it can use to load textures. iOS devices that use the PowerVR MBX graphics hardware have a limit on the total amount of memory they can use for textures and renderbuffers (described in "PowerVR MBX" (page 94)). Where possible, your application 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 your application must balance any changes it makes to its textures with the quality level of the final rendered frame. For best results, try the different techniques described below, and choose the technique 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 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.

Important Future Apple hardware may not support the PVRTC texture format. You must test for the existence of the `GL_IMG_texture_compression_pvrtc` extension before loading a PVRTC compressed texture. For more information on how to check for extensions, see "Check for Extensions Before Using Them" (page 27). For maximum compatibility, your application may want to include uncompressed textures to use when this extension is not available.

For more information on compressing textures into PVRTC format, see "Using texturetool to Compress Textures" (page 100).

Use Lower-Precision Color Formats

If your application 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 application needs that level of quality.

Use Properly Sized Textures

The images that an iOS-based device displays are very small. Your application 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!

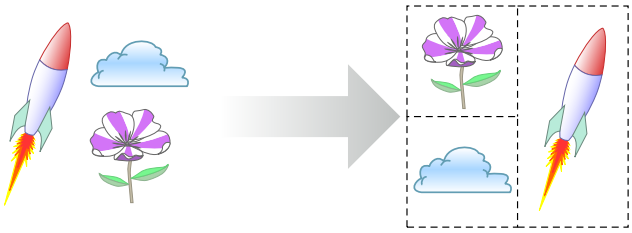
Load Textures During Initialization

Creating and loading textures is an expensive operation. For best results, avoid creating new textures while your application is running. Instead, create and load your texture data during initialization. Be sure to dispose of your original images once you've finished creating the texture.

Once your application creates a texture, avoid changing it except at the beginning or end of a frame. Currently, all iOS devices use a tile-based deferred renderer that makes calls to the `glTexSubImage` and `glCopyTexSubImage` functions particularly expensive. See "Tile-Based Deferred Rendering" (page 89) for more information.

Combine Textures into Texture Atlases

Binding to a texture takes time for OpenGL ES to process. Applications 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**. A texture atlas allows your application to bind a single texture and then make multiple drawing calls that use that texture, 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 limitations of the OpenGL ES implementation, in particular the maximum texture size allowed by the implementation.

Use Mipmapping to Reduce Memory Bandwidth

Your application 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 application 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 APPLE_texture_max_level extension to control how your textures are filtered.

Use Multi-texturing Instead of Multiple Passes

Many applications 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 up to eight. Your application 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 application by calling the `glGetIntegerv` function, passing in `GL_MAX_TEXTURE_UNITS` as the parameter.

If your application 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.

Configure Texture Parameters Before Loading Texture Image Data

Always set any texture parameters before loading texture data, as shown in Listing 10-1. By setting the parameters first, OpenGL ES can optimize the texture data it provides to the graphics hardware to match your settings.

Listing 10-1 Loading a new texture

```
glGenTextures(1, &spriteTexture);
glBindTexture(GL_TEXTURE_2D, spriteTexture);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE,
 spriteData);
```

Best Practices for Shaders

Shaders provide great flexibility to your application, but 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. Listing 11-1 presents a typical strategy to load, compile, and verify a shader program.

Listing 11-1 Loading a Shader

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

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

           // Define shader text
           glShaderSource(shaderText);

           // Compile shader
           glCompileShader(shader);

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

          // Link program
           glLinkProgram(prog);

           // Validate program
           glValidateProgram(prog);

           // Check the status of the compile/link
           glGetProgramiv(prog, GL_INFO_LOG_LENGTH, &logLen);
           if(logLen > 0)
           {
               // Show any errors as appropriate
               glGetProgramInfoLog(prog, logLen, &logLen, log);
               fprintf(stderr, "Prog Info Log: %s\n", log);
       }
```

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

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

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

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

Compile, link, and validate your programs when your application is initialized. Once you've created all your shaders, your application can efficiently switch between them by calling `glUseProgram`.

Respect the Hardware Limits on Shaders

OpenGL ES places limitations on the number of each variable type you can use in a vertex or fragment shader. OpenGL ES implementations are not required to implement a software fallback when these limits are exceeded; instead, the shader simply fails to compile or link. Your application must validate all shaders to ensure that no errors occurred during compilation, as shown above in Listing 11-1.

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 application. 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.

For iOS applications, 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 application to ensure that the results are what you expect.

Listing 11-2 defaults to high precision variables, but calculates the color output using low precision variables because higher precision is not necessary.

Listing 11-2 Low precision is acceptable for fragment color

```
default precision highp; // Default precision declaration is required in fragment
 shaders.
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);
}
```

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 11-3 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 11-3 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 11-4. 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 11-4 Proper use of vector operations

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

Similarly, your application 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 11-5 runs twice as fast on a scalar processor because it specifies that only two components are needed.

Listing 11-5 Specifying a write mask

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

Use Uniform or Constants Instead of Computation Within 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.

Avoid Branching

Branches are discouraged in shaders, as they can reduce the ability to execute operations in parallel on 3D graphics processors. 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: Branching on a value computed inside the shader.

Instead of creating a large shader with many knobs and levers, 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.

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:

```
// Loop
    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.

Note Starting in iOS 4.2, the shader compiler applies more aggressive loop optimizations.

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.

Avoid 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. Although the shader language supports this, 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 11-6 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, your application avoids 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 11-6 Dependent Texture Read

```
varying vec2 vTexCoord;
uniform sampler textureSampler;

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

Platform Notes

Apple provides different implementations of OpenGL ES for different hardware platforms. Each implementation has different capabilities, from the maximum size allowed for textures to the list of supported OpenGL ES extensions. This chapter spells out the details on each platform to assist you in tailoring your applications to get the highest performance and quality.

The information in this chapter is current as of iOS 4.2 but is subject to change in future hardware or software releases. To ensure that your application is compatible with future platforms, always explicitly test for these capabilities at runtime, as described in "Determining OpenGL ES Capabilities" (page 25).

Table 12-1 iOS Hardware Devices List

Device Compatibility	Graphics Platform	OpenGL ES 2.0	OpenGL ES 1.1
iPod Touch	PowerVR MBX	No	Yes
iPod Touch (Second Generation)	PowerVR MBX	No	Yes
iPod Touch (Third Generation)	PowerVR SGX	Yes	Yes
iPod Touch (Fourth Generation)	PowerVR SGX	Yes	Yes
iPhone	PowerVR MBX	No	Yes
Phone 3G	PowerVR MBX	No	Yes
iPhone 3GS	PowerVR SGX	Yes	Yes
iPhone 3GS (China)	PowerVR SGX	Yes	Yes
iPhone 4	PowerVR SGX	Yes	Yes
iPad Wi-FI	PowerVR SGX	Yes	Yes
iPad Wi-Fi+3G	PowerVR SGX	Yes	Yes

PowerVR SGX Platform

The PowerVR SGX is the graphics processor in the iPhone 3GS, iPhone 4, third-generation iPod touch, and the iPad, and is designed to support OpenGL ES 2.0. The graphics driver for the PowerVR SGX also implements OpenGL ES 1.1 by efficiently implementing the fixed-function pipeline using shaders. More information about PowerVR technologies can be found in the PowerVR Technology Overview. Detailed information about the PowerVR SGX can be found in the POWERVR SGX OpenGL ES 2.0 Application Development Recommendations.

Tile-Based Deferred Rendering

The PowerVR SGX uses a technique known as **tile based deferred rendering (TBDR)**. When you call OpenGL ES functions to submit rendering commands to the hardware, those commands are buffered until a large list of commands are accumulated. These commands are rendered by the hardware as a single operation. To render the image, the framebuffer is divided into tiles, and the commands are drawn once for each tile, with each tile rendering only the primitives that are visible within it. The key advantage to a deferred renderer is that it accesses memory very efficiently. Partitioning rendering into tiles allows the GPU to more effectively cache the pixel values from the framebuffer, making depth testing and blending more efficient.

Another advantage of deferred rendering is that it allows the GPU to 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. To gain the most benefit from this feature, draw as much of the frame with opaque content as possible and minimize use of blending, alpha testing, and the `discard` instruction in GLSL shaders. Because the hardware performs hidden surface removal, it is not necessary for your application to sort primitives from front to back.

Some operations under a deferred renderer are more expensive than they would be under a traditional stream renderer. The memory bandwidth and computational savings described above perform best when processing large scenes. 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 application renders a batches of triangles using a texture, and then modifies the texture, the OpenGL ES implementation must either flush those commands immediately or duplicate the texture. Neither option uses the hardware efficiently. Similarly, any attempt to read pixel data from the framebuffer requires that preceding commands be processed if they would alter that framebuffer.

Release Notes and Best Practices for the PowerVR SGX

These practices apply to both OpenGL ES 2.0 and OpenGL ES 1.1 applications.

- Avoid operations that alter OpenGL ES objects that are already in use by the renderer (because of previously submitted drawing commands). When you need to modify OpenGL ES resources, schedule those modifications at the beginning or end of a frame. These commands include `glBufferSubData`, `glBufferData`, `glMapBuffer`, `glTexSubImage`, `glCopyTexImage`, `glCopyTexSubImage`, `glReadPixels`, `glBindFramebuffer`, `glFlush`, and `glFinish`.
- To take advantage of the processor's hidden surface removal, follow the drawing guidelines found in "Do Not Sort Rendered Objects Except Where Necessary For Correct Rendering" (page 58).
- Vertex buffer objects (VBOs) provide a significant performance improvement on the PowerVR SGX. See "Use Vertex Buffer Objects to Manage Copying Vertex Data" (page 70).
- In iOS 4.0 and later, separate stencil buffers are not supported. Use a combined depth/stencil buffer.
- In iOS 4.0 and later, the performance of `glTexImage2D` and `glTexSubImage2D` have been significantly improved.
- In iOS 4.2 and later, the performance of Core Animation rotations of renderbuffers have been significantly improved, and are now the preferred way to rotate content between landscape and portrait mode. For best performance, ensure the renderbuffer's height and width are each a multiple of 32 pixels.

OpenGL ES 2.0 on the PowerVR SGX

Limits

- The maximum 2D or cube map texture size is 2048 x 2048. This is also the maximum renderbuffer size and viewport size.
- You can use up to 8 textures in a fragment shader. You cannot use texture lookups in a vertex shader.
- You can use up to 16 vertex attributes.
- You can use up to 8 varying vectors.
- You can use up to 128 uniform vectors in a vertex shader and up to 64 uniform vectors in a fragment shader.
- Points can range in size from 1.0 to 511.0 pixels.
- Lines can range in width from 1.0 to 16.0 pixels.

Supported Extensions

The following extensions are supported:

- APPLE_framebuffer_multisample
- APPLE_rgb_422

- APPLE_texture_format_BGRA8888
- APPLE_texture_max_level
- EXT_blend_minmax
- EXT_discard_framebuffer
- EXT_read_format_bgra
- EXT_shader_texture_lod
- EXT_texture_filter_anisotropic
- IMG_read_format
- IMG_texture_compression_pvrtc
- OES_depth24
- OES_depth_texture
- OES_fbo_render_mipmap
- OES_mapbuffer
- OES_packed_depth_stencil
- OES_rgb8_rgba8
- OES_standard_derivatives
- OES_texture_half_float
- OES_texture_float
- OES_vertex_array_object

iOS 3.1 and later also support rendering to cube map textures. Previous versions of iOS return `FRAMEBUFFER_UNSUPPORTED` for cube maps.

Known Limitations and Issues

- The PowerVR SGX does not support non–power of two cube mapped or mipmapped textures

Best Practices on OpenGL ES 2.0

In iOS 4.2 and later, the shader compiler performs more aggressive loop unrolling.

The PowerVR SGX processes high-precision floating-point calculations using a scalar processor, even when those values are declared in a vector. Proper use of write masks and careful definitions of your calculations can improve the performance of your shaders. See "Perform Vector Calculations Lazily" (page 84) for more information.

Medium- and low-precision floating-point values are processed in parallel. However, low-precision variables have a few specific performance limitations:

- Swizzling components of vectors declared with low precision is expensive and should be avoided.
- Many built-in functions use medium-precision inputs and outputs. If your application provides low-precision floating-point values as parameters or assigns the results to a low-precision floating-point variable, the shader may have to include additional instructions to convert the values. These additional instructions are also added when swizzling the vector results of a computation.

For best results, limit your use of low-precision variables to color values.

OpenGL ES 1.1 on the PowerVR SGX

OpenGL ES 1.1 is implemented on the PowerVR SGX hardware using customized shaders. Whenever your application changes OpenGL ES state variables, a new shader is implicitly generated as needed. Because of this, changing OpenGL ES state may be more expensive than it would be on a pure hardware implementation. You can improve the performance of your application by reducing the number of state changes it performs. For more information, see "Be Mindful of OpenGL ES State Variables" (page 53).

Limits

- The maximum 2D texture size is 2048 x 2048. This is also the maximum renderbuffer size and viewport size.
- There are 8 texture units available.
- Points can range in size from 1.0 to 511.0 pixels.
- Lines can range in width from 1.0 to 16.0 pixels.
- The maximum texture LOD bias is 4.0.
- For `GL_OES_matrix_palette`, the maximum number of palette matrices is 11 and the maximum vertex units is 4.
- The maximum number of user clip planes is 6.

Supported Extensions

- APPLE_framebuffer_multisample
- APPLE_texture_2D_limited_npot
- APPLE_texture_format_BGRA8888
- APPLE_texture_max_level

- EXT_blend_minmax
- EXT_discard_framebuffer
- EXT_read_format_bgra
- EXT_texture_filter_anisotropic
- EXT_texture_lod_bias
- IMG_read_format
- IMG_texture_compression_pvrtc
- OES_blend_equation_separate
- OES_blend_func_separate
- OES_blend_subtract
- OES_compressed_paletted_texture
- OES_depth24
- OES_draw_texture
- OES_fbo_render_mipmap
- OES_framebuffer_object
- OES_mapbuffer
- OES_matrix_palette
- OES_packed_depth_stencil
- OES_point_size_array
- OES_point_sprite
- OES_read_format
- OES_rgb8_rgba8
- OES_stencil8
- OES_stencil_wrap
- OES_texture_mirrored_repeat
- OES_vertex_array_object

PowerVR MBX

The PowerVR MBX implements the OpenGL ES 1.1 fixed-function pipeline. More information about PowerVR technologies can be found in the PowerVR Technology Overview. Detailed information about the PowerVR MBX can be found in the PowerVR MBX 3D Application Development Recommendations.

The PowerVR MBX is a tile-based deferred renderer. Although it does not support custom fragment shaders, as on OpenGL ES 2.0, the graphics hardware still avoids calculating color values for hidden fragments. See "Tile-Based Deferred Rendering" (page 89) for more information on how to tailor your application to perform well on a deferred renderer.

OpenGL ES applications targeting the PowerVR MBX must limit themselves to no more than 24 MB of memory for textures and renderbuffers. Overall, the PowerVR MBX is more sensitive to memory usage, and your application should minimize the size of textures and renderbuffers.

Release Notes and Best Practices for the PowerVR MBX

- For best performance, interleave the standard vertex attributes in the following order: `Position`, `Normal`, `Color`, `TexCoord0`, `TexCoord1`, `PointSize`, `Weight`, `MatrixIndex`.
- Starting in iOS 4, the performance of `glTexImage2D` and `glTexSubImage2D` have been significantly improved.
- Combining vertex array objects and static vertex buffer objects to submit your vertex data to OpenGL ES allows the driver to perform additional optimizations, reducing the overhead of getting your data to the graphics hardware.

OpenGL ES 1.1 on the PowerVR MBX

Limits

- The maximum 2D texture size is 1024 x 1024. This is also the maximum renderbuffer size and viewport size.
- There are 2 texture units available.
- Points can range in size from 1.0 to 64 pixels.
- Lines can range in width from 1.0 to 64 pixels.
- The maximum texture LOD bias is 2.0.
- For `GL_OES_matrix_palette`, the maximum number of palette matrices is 9 and the maximum vertex units is 3.
- The maximum number of user clip planes is 1.

Supported Extensions

The extensions supported by the OpenGL ES 1.1 implementation for iOS are:

- APPLE_framebuffer_multisample
- APPLE_texture_format_BGRA8888
- APPLE_texture_max_level
- EXT_discard_framebuffer
- EXT_texture_filter_anisotropic
- EXT_texture_lod_bias
- IMG_read_format
- IMG_texture_compression_pvrtc
- OES_blend_subtract
- OES_compressed_paletted_texture
- OES_depth24
- OES_draw_texture
- OES_framebuffer_object
- OES_mapbuffer
- OES_matrix_palette
- OES_point_size_array
- OES_point_sprite
- OES_read_format
- OES_rgb8_rgba8
- OES_texture_mirrored_repeat
- OES_vertex_array_object

Known Limitations and Issues

The PowerVR MBX implementation of OpenGL ES 1.1 does not completely conform to the OpenGL ES 1.1 specification. For compatibility reasons, your application should avoid these edge cases:

- The texture magnification and minification filters (within a texture level) must match:
    - Supported:

```
GL_TEXTURE_MAG_FILTER = GL_LINEAR,
GL_TEXTURE_MIN_FILTER = GL_LINEAR_MIPMAP_LINEAR
```

- Supported:

```
GL_TEXTURE_MAG_FILTER = GL_NEAREST,
GL_TEXTURE_MIN_FILTER = GL_NEAREST_MIPMAP_LINEAR
```

- Not Supported:

```
GL_TEXTURE_MAG_FILTER = GL_NEAREST,
GL_TEXTURE_MIN_FILTER = GL_LINEAR_MIPMAP_LINEAR
```

- There are a few, rarely used texture environment operations that aren't available:
  - If the value of `GL_COMBINE_RGB` is `GL_MODULATE`, only one of the two operands may read from a `GL_ALPHA` source.
  - If the value of `GL_COMBINE_RGB` is `GL_INTERPOLATE`, `GL_DOT3_RGB`, or `GL_DOT3_RGBA`, then several combinations of `GL_CONSTANT` and `GL_PRIMARY_COLOR` sources and `GL_ALPHA` operands do not work properly.
  - If the value of `GL_COMBINE_RGB` or `GL_COMBINE_ALPHA` is `GL_SUBTRACT`, then `GL_SCALE_RGB` or `GL_SCALE_ALPHA` must be `1.0`.
  - If the value of `GL_COMBINE_ALPHA` is `GL_INTERPOLATE` or `GL_MODULATE`, only one of the two sources can be `GL_CONSTANT`.
  - The value of `GL_TEXTURE_ENV_COLOR` must be the same for all texture units.
- Two-sided lighting (`GL_LIGHT_MODEL_TWO_SIDE`) is ignored.
- The `factor` argument to `glPolygonOffset` is ignored. Only the slope-independent `units` parameter is honored.
- Perspective-correct texturing is supported only for the S and T texture coordinates. The Q coordinate is not interpolated with perspective correction.

iOS Simulator

iOS Simulator includes complete and conformant implementations of both OpenGL ES 1.1 and OpenGL ES 2.0 that you can use for your application development. Simulator differs from the PowerVR MBX and PowerVR SGX in a number of ways:

- Simulator does not use a tile-based deferred renderer.
- Simulator does not enforce the memory limitations of the PowerVR MBX.
- Simulator does not support the same extensions as either the PowerVR MBX or the PowerVR SGX.
- Simulator does not provide a pixel-accurate match to either the PowerVR MBX or the PowerVR SGX.

Important Rendering performance of OpenGL ES in Simulator has no relation to the performance of OpenGL ES on an actual device. Simulator provides an optimized software rasterizer that takes advantage of the vector processing capabilities of your Macintosh computer. As a result, your OpenGL ES code may run faster or slower in iOS simulator (depending on your computer and what you are drawing) than on an actual device. Always profile and optimize your drawing code on a real device and never assume that Simulator reflects real-world performance.

OpenGL ES 2.0 on Simulator

Supported Extensions

Simulator supports the following extensions to OpenGL ES 2.0:

- APPLE_framebuffer_multisample
- APPLE_rgb_422
- APPLE_texture_format_BGRA8888
- APPLE_texture_max_level
- EXT_blend_minmax
- EXT_discard_framebuffer
- EXT_shader_texture_lod
- EXT_texture_filter_anisotropic
- IMG_read_format
- IMG_texture_compression_pvrtc
- OES_depth24
- OES_depth_texture
- OES_fbo_render_mipmap
- OES_mapbuffer
- OES_rgb8_rgba8
- OES_texture_half_float

- OES_texture_float
- OES_vertex_array_object

OpenGL ES 1.1 on Simulator

Supported Extensions

- APPLE_framebuffer_multisample
- APPLE_texture_2D_limited_npot
- APPLE_texture_format_BGRA8888
- APPLE_texture_max_level
- EXT_blend_minmax
- EXT_discard_framebuffer
- EXT_texture_filter_anisotropic
- EXT_texture_lod_bias
- EXT_texture_filter_anisotropic
- IMG_read_format
- IMG_texture_compression_pvrtc
- OES_blend_equation_separate
- OES_blend_func_separate
- OES_blend_subtract
- OES_compressed_paletted_texture
- OES_depth24
- OES_draw_texture
- OES_fbo_render_mipmap
- OES_framebuffer_object
- OES_mapbuffer
- OES_matrix_palette
- OES_point_size_array
- OES_point_sprite
- OES_read_format
- OES_rgb8_rgba8

- OES_stencil8
- OES_stencil_wrap
- OES_texture_mirrored_repeat
- OES_vertex_array_object

Using texturetool to Compress Textures

The iOS SDK includes a tool that allows you to compress your textures into the PVR texture compression format, aptly named `texturetool`. If you have Xcode installed with the iOS 4.3 SDK in the default location (`/Developer/Platforms/`), then `texturetool` is located at: `/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 3.0. Options not compatible with previous versions of iOS are noted.

texturetool Parameters

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

```
user$ texturetool -h


Usage: texturetool [-hlm] [-e <encoder>] [-p <preview_file>] -o <output> [-f
<format>] input_image


    -h            Display this help menu.

    -l            List available encoders, individual encoder options, and file
formats.

    -m            Generate a complete mipmap chain from the input 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 A-1 Encoding options

```
user$ texturetool –l
Encoders:

PVRTC
 ––channel–weighting–linear
 ––channel–weighting–perceptual
 ––bits–per–pixel–2
 ––bits–per–pixel–4

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 allows you to automatically generate 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 iOS 2.2 SDK added an additional format (`–f`) parameter that allows you to control the format of its output file. Although this parameter is not available with iOS 2.1 or earlier, the data files produced are compatible with those versions of iOS.

The default format is Raw, which is equivalent to the format that `texturetool` produced under iOS SDK 2.0 and 2.1. 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. Your application must parse the data header to obtain the actual texture data. See the *PVRTextureLoader* sample for an example of working with texture data in the PVR format.

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 Mac 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 two in length. If your application attempts to load a non-square or non-power-of-two texture in iOS, an error is returned.

Listing A-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 A-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.

Listing A-4 Example of uploading PVRTC data to the graphics chip

```
void texImage2DPVRTC(GLint level, GLsizei bpp, GLboolean hasAlpha, GLsizei width,
 GLsizei height, void *pvrtcData)
{
    GLenum format;
    GLsizei size = width * height * bpp / 8;
    if(hasAlpha) {
        format = (bpp == 4) ? GL_COMPRESSED_RGBA_PVRTC_4BPPV1_IMG :
GL_COMPRESSED_RGBA_PVRTC_2BPPV1_IMG;
    } else {
        format = (bpp == 4) ? GL_COMPRESSED_RGB_PVRTC_4BPPV1_IMG :
GL_COMPRESSED_RGB_PVRTC_2BPPV1_IMG;
    }
    if(size < 32) {
        size = 32;
    }
    glCompressedTexImage2D(GL_TEXTURE_2D, level, format, width, height, 0, size,
data);
}
```

For sample code, see the *PVRTextureLoader* sample.

Document Revision History

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

Date	Notes
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.
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 anti-aliasing operations.

anti-aliasing In graphics, a technique used to smooth and soften the jagged (or aliased) edges that are sometimes apparent when graphical objects such as text, line art, and images are drawn.

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 application.

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 allows user-defined matrices.

depth In OpenGL, refers to the *z* coordinate and specifies how far a pixel lies from the observer.

depth buffer A block of memory used to store a depth value for each pixel. The depth buffer is used to determine whether or not a pixel can be seen by the observer. 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 while 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, drawable objects implement the `EAGLDrawable` protocol. Currently, only `CAEAGLLayer` objects can be used as drawable objects and are used to integrate 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 rastering 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 to support integrating OpenGL ES into the graphics subsystem, such as a windowing system. iOS does not use system framebuffers; instead, it provides additional functions that allow renderbuffers to share their contents with Core Animation.

framebuffer attachable image The rendering destination for a framebuffer object.

framebuffer object An core feature in OpenGL ES 2.0 that allows an application to create an OpenGL ES-managed framebuffer destination. A framebuffer object (FBO) contains state information for an OpenGL ES framebuffer and its set of images, called renderbuffers. All iOS implementations of OpenGL ES 1.1 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 transforms 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 application.

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

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 The number of bits per pixel in a pixel image.

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

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. OpenGL ES implementations are not required to provide a software fallback when the capabilities of the graphics hardware are exceeded.

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; can be one-, two-, or three-dimensional or be a cube map.

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

texture matrix A 4 x 4 matrix that OpenGL 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 where the data for that vertex should be read from. Vertex array objects simplify the effort of configuring the graphics pipeline by allowing the configuration to be initialized, and then recovered quickly.

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