OpenGL ES Programming Guide for iOS

Graphics & Animation: 3D Drawing

2010-07-09

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

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction

The **Open Graphics Library (OpenGL)** is a cross-platform C-based interface 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 specify models as points, lines, or polygons to which an infinite number of shading techniques can be applied to produce a desired rendering effect. OpenGL functions send graphics commands to the underlying hardware, where they are then rendered. Because this underlying hardware is dedicated to processing graphics commands, OpenGL drawing is typically very fast.

OpenGL for Embedded Systems (OpenGL ES) is a version of OpenGL designed for mobile devices and takes advantage of modern graphics hardware. OpenGL ES simplifies the OpenGL interface, making it both easier for hardware to implement and also easier to learn.

You should read this document if

- You are new to OpenGL ES and want to determine if it is the right programming interface to use in your application.
- You are familiar with OpenGL or OpenGL ES and want to learn how to use it on iOS.

Organization of This Document

This document is organized into the following sections:

- "OpenGL ES on iOS" (page 11) presents an overview of OpenGL ES and how it fits into the graphics subsystem on iOS.
- "Determining OpenGL ES Capabilities" (page 17) recommends how you can choose a version of OpenGL ES and test its capabilities at runtime in order to provide a robust experience inside your application on every iOS-based device.
- "Working with OpenGL ES Contexts and Framebuffers" (page 21) describes how to work with EAGL to create a drawing context and a framebuffer as the destination for your drawing commands.
- "Best Practices for Working with Vertex Data" (page 29) describes how to efficiently submit your geometry to the OpenGL ES rendering pipeline.
- "Best Practices for Working with Texture Data" (page 35) describes strategies for creating and using texture data.
- "Performance Guidelines" (page 39) provides general purpose guidelines on how to improve your application's performance on both OpenGL ES 1.1 and OpenGL ES 2.0.
- "Platform Notes" (page 49) provides detailed information on iPhone Simulator as well as the MBX and SGX graphics processors available on iOS-based devices.
- "Using texturetool to Compress Textures" (page 59) describes how you can use `texturetool` to reduce the memory usage of your textures.

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 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 on extensions to OpenGL ES.
- 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 functions and classes provided by Apple to allow OpenGL ES to be used on iOS.

OpenGL ES on iOS

OpenGL for Embedded Systems (OpenGL ES) is a simplified version of OpenGL that eliminates redundant functionality in order to present a programming interface that is easily implemented in mobile hardware. OpenGL ES allows your application to configure a traditional 3D graphics pipeline and submit vertex data to OpenGL, where they are transformed and lit, assembled into primitives, and rasterized to create a 2D image.

Currently, there are two distinct versions of OpenGL ES:

- OpenGL ES 1.1 implements the standard graphics pipeline with a well-defined **fixed-function pipeline**. The fixed-function pipeline implements a traditional lighting and rasterization model that allows various parts of the pipeline to be enabled and configured to perform specific tasks, or disabled to improve performance.
- OpenGL ES 2.0 shares many functions in common with OpenGL ES 1.1, but removes all functions that act on the fixed-function pipeline, replacing it with a general-purpose **shader**-based pipeline. Shaders allow you to create your own vertex attributes and execute custom vertex and fragment functions directly on the graphics hardware, allowing your application to completely customize the operations applied to each each vertex and fragment.

Apple offers hardware that supports both OpenGL ES 1.1 and OpenGL ES 2.0.

The remainder of this chapter gives an overview of the iOS graphics model and OpenGL ES, and explains how the two fit together.

iOS Graphics Overview

Core Animation is fundamental to the iOS graphics subsystem. Every `UIView` object in your application is backed by a Core Animation layer. As the various layers update their contents, they are animated and composited by Core Animation and presented to the display. This process is described in detail in "Tuning for Performance and Responsiveness" in the *iOS Application Programming Guide*.

OpenGL ES, like every other graphics system on iOS, is a client of Core Animation. To use OpenGL ES to draw to the screen, your application creates a `UIView` class backed by a special Core Animation layer, a `CAEAGLLayer` object. A `CAEAGLLayer` object is aware of OpenGL ES and can be used to create rendering targets that act as part of Core Animation. When your application finishes rendering a frame, you present the contents of the `CAEAGLLayer` object, where they are composited with the data from other views.

The complete discussion of how to create a `CAEAGLLayer` object and use it display your rendered images is in "Working with OpenGL ES Contexts and Framebuffers" (page 21).

Although your application can compose scenes using both OpenGL ES layers and non–OpenGL ES drawing, in practice, you can achieve higher performance by limiting yourself to OpenGL ES. Compositing non–OpenGL ES with OpenGL ES content is covered in more detail in "Displaying Your Results" (page 26).

Overview of OpenGL ES

OpenGL ES provides a procedural API for submitting geometry to a hardware accelerated rendering pipeline. OpenGL ES commands are submitted to a **rendering context**, where they are consumed to generate images that can be displayed to the user. Most commands in OpenGL ES perform one of the following actions:

- Reading the current state of an OpenGL ES context. This is most typically used to determine the capabilities of an OpenGL ES implementation. See "Determining OpenGL ES Capabilities" (page 17) for more information.
- Changing state variables in an OpenGL ES context. This is typically used to configure the pipeline for some future operations. In OpenGL ES 1.1, state variables are used extensively to configure lights, materials, and other values that affect the fixed-function pipeline.
- Creating, modifying or destroying OpenGL ES objects. Both OpenGL ES 1.1 and 2.0 provide a number of objects, defined below.
- Submitting geometry to be rendered. Vertex data is submitted to the pipeline, processed, assembled into primitives and then rasterized to a **framebuffer**.

The OpenGL ES specifications define the precise behavior for each function.

An OpenGL ES implementation is allowed to extend the OpenGL ES specification, either by offering limits higher than the minimum required (such as allowing larger textures) or by extending the API through **extensions**. Apple uses the extensions mechanism to provide a number of critical extensions that help provide great performance in iOS. For example, Apple offers a texture compression extension that makes it easier to fit your textures into the memory available on iOS. Note that the limits and extensions offered by iOS may vary depending on the hardware. Your application must test the capabilities at runtime and alter its behavior to match what is available. For more information on how to do this, see "Determining OpenGL ES Capabilities" (page 17).

OpenGL ES Objects

As described above, OpenGL ES offers a number of objects that can be created and configured to help create your scene. All of these objects are managed for you by OpenGL ES. Some of the most important 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 your geometry but can also be used to map other data onto the geometry (for example, normals or lighting information). "Best Practices for Working with Texture Data" (page 35) discusses critical topics for using textures on Apple's OpenGL ES implementation.
- A **buffer** is a set of memory owned by OpenGL ES that your application can read or write data into. The most common use for a buffer is to hold vertex data that your application wants to submit to the graphics hardware. Because this buffer is owned by the OpenGL ES implementation, it can optimize the placement and format of the data in this buffer in order to more efficiently process vertices, particularly when the data does not change from frame to frame. Using buffers to manage your vertex data can significantly boost the performance of your application.
- **Shaders** are also objects. An OpenGL ES 2.0 application creates a shader, compiles and links code into it, and then assigns it to process vertex and fragment data.

- A **renderbuffer** is a simple 2D graphics image in a specified format. This format may be defined as color data, but it could also be depth or stencil information. Renderbuffers are not usually used alone, but are instead collected and used as part of a framebuffer.
- **Framebuffers** 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 destination for rendering. Framebuffer objects are part of the OpenGL ES 2.0 standard, and Apple also implements them on OpenGL ES 1.1 with the `OES_framebuffer_object` extension. Framebuffers are used extensively on iOS and are described in more detail below. A later chapter, "Working with OpenGL ES Contexts and Framebuffers" (page 21), describes strategies for creating and using framebuffers in iOS applications.

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

1. Generate an object identifier.

   For each object that your application wants to create, you should generate an identifier. An identifier is analogous to a pointer. Whenever your application wants to operate on an object, you use the identifier to specify which object to work on.

   Note that 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.

   Each object type in OpenGL ES has a method to bind an object to the context. You can only work on one object of each type at a time, and you select that object by binding to it. The first time you bind to an object identifier, OpenGL ES allocates memory and initializes that object.

3. Modify the state of your object.

   Commands implicitly operate on the currently bound object. After binding the object, your application makes one or more OpenGL ES calls to configure the object. For example, after binding to a texture, your application makes an additional call to actually load the texture image.

4. Use your objects for rendering.

   Once you've created and configured your objects, you can start drawing your geometry. As you submit vertices, the currently bound objects are used to render your output. In the case of shaders, the current shader is used to calculate the final results. Other objects may be involved at various stages of the pipeline.

5. Delete your objects.

   Finally, when you are done with an object, your application deletes it. When an object is deleted, its contents and object identifier are recycled.

In iOS, OpenGL ES objects are managed by a **sharegroup** object. Two or more rendering contexts can be configured to use the same sharegroup. The two rendering contexts can then use the same data (for example, a texture) to actually share a single texture object. Sharegroups are covered in "EAGLSharegroup" (page 15).

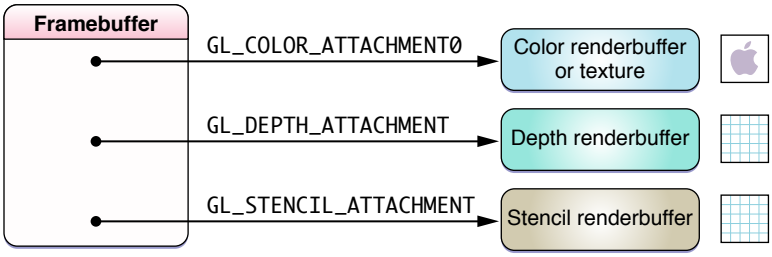
Framebuffers

Framebuffer objects are the target of all rendering commands. Traditionally in OpenGL ES, framebuffers were created using a platform-defined interface. Each platform would provide its own functions to create a framebuffer that can be drawn to the screen. The `OES_framebuffer_object` extended OpenGL ES to provide a standard mechanism to create and configure framebuffers that rendered to offscreen renderbuffers or to textures.

Apple does not provide a platform interface for creating framebuffer objects. Instead, all framebuffer objects are created using the `OES_framebuffer_object` extension. In OpenGL ES 2.0, these functions are part of the core specification.

Framebuffer objects provide storage for color, depth and/or stencil data by attaching images to the framebuffer, as shown in Figure 1-1. The most common image attachment is a renderbuffer. However, a texture can also be attached to the color attachment of a framebuffer, allowing an image to be drawn, and then later texture mapped onto other geometry.

Figure 1-1 Framebuffer with color, depth, and stencil buffers



The typical procedure for creating a framebuffer is as follows:

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

Apple extends framebuffer objects by allowing the color renderbuffer's storage to be allocated so that it is shared with a Core Animation layer. This data can be presented, where it is combined with other Core Animation data and presented to the screen. See "Working with OpenGL ES Contexts and Framebuffers" (page 21) for more information.

iOS Classes

All implementations of OpenGL ES require platform-specific code to create a rendering context and to use it to draw to the screen. iOS does this through EAGL, an Objective-C interface. This section highlights the classes and protocols of the EAGL API, which is covered in more detail in "Working with OpenGL ES Contexts and Framebuffers" (page 21).

EAGLContext

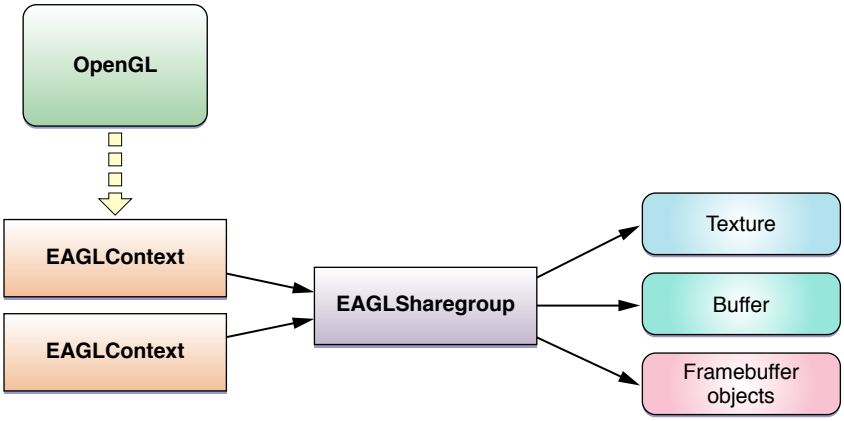
The `EAGLContext` class defines the rendering context that is the target of all OpenGL ES commands. Your application creates and initializes an `EAGLContext` object and makes it the current target of commands. When your application makes calls to OpenGL ES, those commands are typically stored in a queue maintained by the context and later executed to render the final image.

The `EAGLContext` class also provides a method to present images to Core Animation for display.

EAGLSharegroup

Every `EAGLContext` object contains a reference to an `EAGLSharegroup` object. Whenever an object is allocated by OpenGL ES for that context, the object is actually allocated and maintained by the sharegroup. This division of responsibility is useful because it is possible to create two or more contexts that use the same sharegroup. In this scenario, objects that are allocated by one rendering context can be used by another, as shown in Figure 1-2.

Figure 1-2 Two contexts sharing OpenGL objects



Using a sharegroup allows two or more contexts to share OpenGL resources without duplicating that data for each context. Resources on a mobile device are scarcer than those on desktop hardware. By sharing textures, shaders and other objects, your application makes better use of available resources.

An application can also implement resource-sharing mechanism by using a single context and creating one framebuffer object for each rendering destination. The application switches the current target for rendering commands as needed, without changing the current context.

EAGLDrawable Protocol

Your application does not directly implement the `EAGLDrawable` protocol on any objects. An `EAGLContext` object recognizes that objects that implement this protocol can allocate storage for a renderbuffer that can later be presented by the user. Only renderbuffers that are allocated using the drawable object can be presented in this way.

In iOS, this protocol is implemented only by the `CAEAGLLayer` class, to associate OpenGL ES renderbuffers with the Core Animation graphics system.

Determining OpenGL ES Capabilities

OpenGL ES in iOS supports both OpenGL ES 1.1, implementing a fixed-function graphics pipeline, and OpenGL ES 2.0, supporting a shader pipeline. OpenGL ES 2.0 is *not* a superset of OpenGL ES 1.1; the fixed-function capabilities of OpenGL ES 1.1 were removed to provide a streamlined interface to the graphics hardware.

Both OpenGL ES 1.1 and OpenGL ES 2.0 define minimum capabilities that every implementation must support. However, the OpenGL ES specification does not limit a hardware implementation to just those capabilities. An OpenGL ES implementation can extend the minimum capabilities (for example, increasing the maximum size of a texture) or extend the capabilities of OpenGL ES through the OpenGL ES extensions mechanism. Apple uses both mechanisms to offer a variety of capabilities on different iOS-based devices. A later chapter, "Platform Notes" (page 49), drills down into these specific capabilities. However, because Apple provides different versions of OpenGL ES, and different capabilities even within the same version, your application must test the capabilities of the device and adjust its behavior to match those capabilities. If your application fails to test the capabilities of OpenGL ES at runtime, it may crash or fail to run, providing a bad experience to the user.

Which Version Should I Target?

When designing your OpenGL ES application, the first question you must answer is whether your application supports OpenGL ES 1.1, OpenGL ES 2.0, or both.

The fixed-function pipeline of OpenGL ES 1.1 provides good baseline behavior for a 3D graphics pipeline, from transforming and lighting vertices to blending the final pixels with the framebuffer. If you choose to implement an OpenGL ES 2.0 application, you need to duplicate this functionality. OpenGL ES 2.0 is more flexible than OpenGL ES 1.1. Custom vertex and fragment operations that would be difficult or impossible to implement using OpenGL ES 1.1 can be trivially implemented with an OpenGL ES 2.0 shader. Implementing a custom operation in an OpenGL ES 1.1 application often requires multiple rendering passes and complex changes to OpenGL ES state that obscure the intent of the code. As your algorithms grow in complexity, shaders convey those operations more clearly and concisely and with better performance.

Your application should target OpenGL ES 1.1 if you want to support all iPhone and iPod touch devices. Your application should target OpenGL ES 2.0 when you want to take advantage of the rich and expressive power of OpenGL ES 2.0 shaders. You can also implement both in the same application, providing baseline behavior that works on OpenGL ES 1.1 devices, and an enriched experience on devices that support OpenGL ES 2.0.

Creating an OpenGL ES Rendering Context

In iOS, your application decides which version of OpenGL ES to use when it initializes an OpenGL ES rendering context, known as an `EAGLContext` object. Your application states which version of OpenGL ES that a particular context uses. For example, to create a context that uses OpenGL ES 1.1, your application would initialize it as shown:

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

If your application wants to use OpenGL ES 2.0, the code is almost identical:

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

If a particular implementation of OpenGL ES is not available, `initWithAPI:` returns `nil`. Your application must test whether the context was successfully initialized before using it.

To support both OpenGL ES 2.0 and OpenGL ES 1.1 in the same application, your application should first attempt to create an OpenGL ES 2.0 rendering context. If that fails, it tries to create an OpenGL ES 1.1 context instead, as shown in Listing 2-1.

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

```
EAGLContext* CreateBestEAGLContext()
{
   EAGLContext *context = [[EAGLContext alloc]
initWithAPI:kEAGLRenderingAPIOpenGLES2];
   if (context == nil)
   {
      context = [[EAGLContext alloc] initWithAPI:kEAGLRenderingAPIOpenGLES1];
   }
   return context;
}
```

Once your application has an initialized context, it can read the `API` property of the context to determine which version of OpenGL ES it supports. For example, your application could implement a fixed-function renderer and a shader renderer in classes, sharing a common base class. Your application would instantiate the appropriate renderer at runtime.

Important: OpenGL ES 2.0 removes many functions found in OpenGL ES 1.1 and adds functions that are not available in OpenGL ES 1.1. 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. While some sharing of code is possible between your two rendering paths, you should limit this sharing to calls that work identically between the two versions of OpenGL ES.

Restricting Your Application to a Particular Version of OpenGL ES

Starting in iOS 3.0, your application can include an entry in its information property list to prevent the application from launching if a particular version of OpenGL ES is not available. See Device Support in *iOS Application Programming Guide* for further instructions on how to add these keys.

If your application supports both OpenGL ES 2.0 and OpenGL ES 1.1, you should not restrict your application to run only on OpenGL ES 2.0 devices.

Checking Device Capabilities

Whether you've chosen to build an OpenGL ES 1.1 or OpenGL ES 2.0 application, the next thing your application needs to do is find out what the OpenGL ES implementation is capable of. Your renderer should make the context the current context and test these capabilities once, caching the results. It can then use this information to tailor the algorithms it uses. For example, depending on the number of texture units available to your application, you might be able to perform a complex algorithm in a single pass, you might need to perform the algorithm in multiple passes, or you might need to choose a simpler algorithm. Since the capabilities of the context do not change once it has been created, your application can test this once and determine which path it to use.

Implementation-Dependent Values

The OpenGL ES specification defines implementation-dependent values that define the 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. Each of these values has a minimum value that all conforming implementations are expected to support. If your application's usage exceeds these minimums, it must check the limit first, and fail gracefully if the implementation cannot provide the limit desired. Your application may need to load smaller textures, disable a rendering feature, or choose a different implementation.

Although the specification provides a comprehensive list of these limitations, a few stand out in most OpenGL applications. Table 2-1 lists values that both OpenGL ES 1.1 and OpenGL ES 2.0 applications should test if they require more than the minimum values in the specification.

Table 2-1 Common OpenGL ES hardware limitations

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

Further, OpenGL ES 1.1 applications should always check the number of texture units and the number of available clipping planes, as shown in Table 2-2.

Table 2-2 OpenGL ES 1.1 hardware limitations

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

In an OpenGL ES 2.0 application, the limits on your shaders are the primary area you need to test. All graphics hardware supports limited memory to pass attributes into the vertex and fragment shaders. An OpenGL ES 2.0 implementation is not required to provide a software fallback if your application exceeds this usage, so your application must either keep its usage below the minimums as defined in the specification, or it must check the shader limitations documented in Table 2-3 and use them to choose shaders that are within those limits.

Table 2-3 OpenGL ES 2.0 shader limitations

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

Check for Extensions Before Using Them

Any OpenGL ES implementation can extend the capabilities of the API by implementing OpenGL ES extensions. "Platform Notes" (page 49) details the capabilities supported on each implementation of OpenGL ES.

Your application must check for the existence of any OpenGL ES extension before relying on changes it makes to the OpenGL ES specification. The sole exception is the GL_OES_framebuffer_object extension. Framebuffer objects are available in all OpenGL ES 2.0 implementations; Apple extends all implementations of OpenGL ES 1.1 to include it. Apple relies on the framebuffer extension to provide all framebuffer objects in iOS.

Listing 2-2 provides code you can use to check for the existence of extensions.

Listing 2-2 Checking for OpenGL ES extensions.

```
BOOL CheckForExtension(NSString *searchName)
{
// For best results, extensionsNames should be stored in your renderer so that
 it does not
// need to be recreated on each invocation.
    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 call `glGetError` after every OpenGL ES command and flag any error that is returned. Typically, if an error is returned from `glGetError`, it means the application is using the API incorrectly.

Note that repeatedly calling `glGetError` can significantly degrade the performance of your application. You should not call it in the release version of your application.

Working with OpenGL ES Contexts and Framebuffers

Any OpenGL ES implementation provides a platform-specific library that includes functions to create and manipulate a rendering context. The rendering context maintains a copy of all OpenGL ES state variables and accepts and executes all OpenGL ES commands. In iOS, **EAGL** is the library that provides this functionality. An EAGL context (`EAGLContext`) is a rendering context, executing OpenGL ES commands and interacting with Core Animation to present the final images to the user. An EAGL sharegroup (`EAGLSharegroup`) extends the rendering context by allowing multiple rendering contexts to share OpenGL ES objects. In iOS, you might use a sharegroup to share objects to conserve memory by sharing textures and other expensive resources.

Creating an EAGL Context

Before your application can execute any OpenGL ES commands, it must first create and initialize an EAGL context and make it the current context.

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

When your application initializes the context, it chooses which version of OpenGL ES to use for that context. For more information on choosing a version of OpenGL ES to use, see "Which Version Should I Target?" (page 17).

Each thread in your application maintains a pointer to a current rendering context. When your application makes a context current, EAGL releases any previous context, retains the context object and sets it as the target for future OpenGL ES rendering commands. Note that in many cases, creating multiple rendering contexts may be unnecessary. You can often accomplish the same results using a single rendering context and one framebuffer object for each image you need to render.

Creating Framebuffer Objects

Although an EAGL context receives commands, it is not the ultimate target of those commands. Your application provides a destination to render the pixels into. In iOS, all images are rendered to framebuffer objects. Framebuffer objects are provided by all OpenGL ES 2.0 implementations, and Apple also provides them in all implementations of OpenGL ES 1.1 through the `GL_OES_framebuffer_object` extension. Framebuffer objects allow an application to precisely control the creation of color, depth, and stencil targets. Most of the time, these targets are known as renderbuffers, which is just a 2D image of pixels with a height, width, and format. Further, the color target may also be used to point at a texture.

The procedure to create a framebuffer is similar in both cases:

1. Create a framebuffer object.

2. Create one or more targets (renderbuffers or textures), allocate storage for them, and attach them to the framebuffer object.

3. Test the framebuffer for completeness.

The following sections explore these concepts in more detail.

Offscreen Framebuffer Objects

An offscreen framebuffer uses an OpenGL ES renderbuffer to hold the rendered image.

The following code allocates a complete offscreen framebuffer object on OpenGL ES 1.1. An OpenGL ES 2.0 application would omit the OES suffix.

1. Create the framebuffer and bind it so that future OpenGL ES framebuffer commands are directed to it.

```
GLuint framebuffer;
glGenFramebuffersOES(1, &framebuffer);
glBindFramebufferOES(GL_FRAMEBUFFER_OES, framebuffer);
```

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

```
GLuint colorRenderbuffer;
glGenRenderbuffersOES(1, &colorRenderbuffer);
glBindRenderbufferOES(GL_RENDERBUFFER_OES, colorRenderbuffer);
glRenderbufferStorageOES(GL_RENDERBUFFER_OES, GL_RGBA8_OES, width, height);
glFramebufferRenderbufferOES(GL_FRAMEBUFFER_OES, GL_COLOR_ATTACHMENT0_OES,
GL_RENDERBUFFER_OES, colorRenderbuffer);
```

3. Perform similar steps to create and attach a depth renderbuffer.

```
GLuint depthRenderbuffer;
glGenRenderbuffersOES(1, &depthRenderbuffer);
glBindRenderbufferOES(GL_RENDERBUFFER_OES, depthRenderbuffer);
glRenderbufferStorageOES(GL_RENDERBUFFER_OES, GL_DEPTH_COMPONENT16_OES, width,
 height);
glFramebufferRenderbufferOES(GL_FRAMEBUFFER_OES, GL_DEPTH_ATTACHMENT_OES,
GL_RENDERBUFFER_OES, depthRenderbuffer);
```

4. Test the framebuffer for completeness.

```
GLenum status = glCheckFramebufferStatusOES(GL_FRAMEBUFFER_OES) ;
if(status != GL_FRAMEBUFFER_COMPLETE_OES) {
    NSLog(@"failed to make complete framebuffer object %x", status);
}
```

Using Framebuffer Objects to Render to a Texture

Your application might want to render directly into a texture and use it as a source for later drawing. For example, you could use this to render a reflection in a mirror that could then be composited into your scene. The code to create this framebuffer is almost identical to the offscreen example, except that this time a texture is attached to the color attachment point.

1. Create the framebuffer object.

```
GLuint framebuffer;
glGenFramebuffersOES(1, &framebuffer);
glBindFramebufferOES(GL_FRAMEBUFFER_OES, framebuffer);
```

2. Create a texture to hold the color data.

```
// 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);
```

3. Attach the texture to the framebuffer.

```
glFramebufferTexture2DOES(GL_FRAMEBUFFER_OES, GL_COLOR_ATTACHMENT0_OES,
GL_TEXTURE_2D, texture, 0);
```

4. Allocate and attach a depth buffer.

```
GLuint depthRenderbuffer;
glGenRenderbuffersOES(1, &depthRenderbuffer);
glBindRenderbufferOES(GL_RENDERBUFFER_OES, depthRenderbuffer);
glRenderbufferStorageOES(GL_RENDERBUFFER_OES, GL_DEPTH_COMPONENT16_OES, width,
 height);
glFramebufferRenderbufferOES(GL_FRAMEBUFFER_OES, GL_DEPTH_ATTACHMENT_OES,
GL_RENDERBUFFER_OES, depthRenderbuffer);
```

5. Test the framebuffer

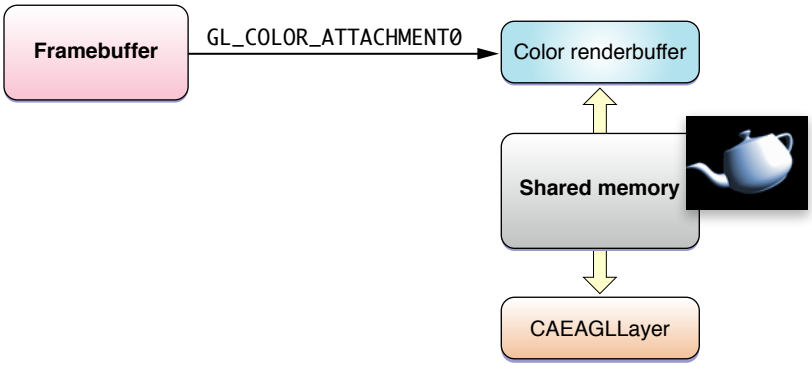
```
GLenum status = glCheckFramebufferStatusOES(GL_FRAMEBUFFER_OES) ;
if(status != GL_FRAMEBUFFER_COMPLETE_OES) {
    NSLog(@"failed to make complete framebuffer object %x", status);
}
```

Drawing to the Screen

Although both offscreen targets and textures are interesting, neither can display their pixels to the screen. To do that, your application needs to interact with Core Animation.

In iOS, all `UIView` objects are backed by Core Animation layers. In order for your application to present OpenGL ES content to the screen, your application needs a `UIView` object as the target. Further, that view must be backed by a special Core Animation layer, a `CAEAGLLayer` object. A `CAEAGLLayer` object is aware of OpenGL ES and references a renderbuffer, as shown in Figure 3-1 (page 24). When your application wants to display these results, the contents of this renderbuffer are animated and composited with other Core Animation layers and sent to the screen.

Figure 3-1 Core Animation shares the renderbuffer with OpenGL ES



The OpenGL ES template provided by Xcode does this work for you, but it is illustrative to walk through the steps used to create a framebuffer object that can be displayed to the screen.

1. Subclass `UIView` and set up a view for your iPhone application.

2. Override the `layerClass` method of the `UIView` class so that objects of your view class create and initialize a `CAEAGLLayer` object rather than a `CALayer` object.

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

3. Get the layer associated with the view by calling the `layer` method of `UIView`.

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

4. Set the layer properties.

   For optimal performance, it is recommended that you mark the layer as opaque by setting the `opaque` property provided by the `CALayer` class. See "Displaying Your Results" (page 26).

5. 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 format of rendered pixels and whether or not the renderbuffer retains its contents after they are presented to the screen. You identify these properties in the dictionary using the `kEAGLDrawablePropertyColorFormat` and `kEAGLDrawablePropertyRetainedBacking` keys. For a list of the keys you can set, see *EAGLDrawable Protocol Reference*.

6. Create the framebuffer, as before.

```
GLuint framebuffer;
glGenFramebuffersOES(1, &framebuffer);
glBindFramebufferOES(GL_FRAMEBUFFER_OES, framebuffer);
```

7. Create the color renderbuffer and call the rendering context to allocate the storage on our Core Animation layer. The width, height, and format of the renderbuffer storage are derived from the bounds and properties of the `CAEAGLLayer` object at the moment the `renderbufferStorage:fromDrawable:` method is called.

```
GLuint colorRenderbuffer;
glGenRenderbuffersOES(1, &colorRenderbuffer);
glBindRenderbufferOES(GL_RENDERBUFFER_OES, colorRenderbuffer);
[myContext renderbufferStorage:GL_RENDERBUFFER_OES fromDrawable:myEAGLLayer];
glFramebufferRenderbufferOES(GL_FRAMEBUFFER_OES, GL_COLOR_ATTACHMENT0_OES,
GL_RENDERBUFFER_OES, colorRenderbuffer);
```

If the Core Animation layer's properties changes, your application should reallocate the renderbuffer by calling `renderbufferStorage:fromDrawable:` again. Failure to do so can result in the rendered image being scaled or transformed for display, which may incur a significant performance cost. For example, in the template, the framebuffer and renderbuffer objects are destroyed and recreated whenever the bounds of the `CAEAGLLayer` object change.

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

```
GLint width;
GLint height;
glGetRenderbufferParameterivOES(GL_RENDERBUFFER_OES, GL_RENDERBUFFER_WIDTH_OES,
 &width);
glGetRenderbufferParameterivOES(GL_RENDERBUFFER_OES, GL_RENDERBUFFER_HEIGHT_OES,
 &height);
```

9. Allocate and attach the depth buffer.

```
GLuint depthRenderbuffer;
glGenRenderbuffersOES(1, &depthRenderbuffer);
glBindRenderbufferOES(GL_RENDERBUFFER_OES, depthRenderbuffer);
glRenderbufferStorageOES(GL_RENDERBUFFER_OES, GL_DEPTH_COMPONENT16_OES, width,
 height);
glFramebufferRenderbufferOES(GL_FRAMEBUFFER_OES, GL_DEPTH_ATTACHMENT_OES,
GL_RENDERBUFFER_OES, depthRenderbuffer);
```

10. Test the framebuffer object.

```
GLenum status = glCheckFramebufferStatusOES(GL_FRAMEBUFFER_OES) ;
if(status != GL_FRAMEBUFFER_COMPLETE_OES) {
    NSLog(@"failed to make complete framebuffer object %x", status);
}
```

To reiterate, the procedure to create a framebuffer object is similar in all three cases, differing only in how you allocate the object attached to the color attachment point of the framebuffer object.

Table 3-1 Mechanisms for allocating the color attachment of the framebuffer

Offscreen renderbuffer	`glRenderbufferStorageOES`
Drawable renderbuffer	`renderbufferStorage: fromDrawable:`
Texture	`glFramebufferTexture2DOES`

Drawing to a Framebuffer Object

Once you've allocated a framebuffer object, you can render to it. All rendering is targeted at the currently bound framebuffer.

```
glBindFramebufferOES(GL_FRAMEBUFFER_OES, framebuffer);
```

Displaying Your Results

Assuming you allocated a color renderbuffer to point at a Core Animation layer, you present its contents by making it the current renderbuffer and calling the `presentRenderbuffer:` method on your rendering context.

```
glBindRenderbufferOES(GL_RENDERBUFFER_OES, colorRenderbuffer);
[context presentRenderbuffer:GL_RENDERBUFFER_OES];
```

By default, the contents of the renderbuffer are invalidated after it is presented to the screen. Your application must *completely* recreate the contents of the renderbuffer every time you draw a frame. If your application needs to preserve the contents between frames, it should add the `kEAGLDrawablePropertyRetainedBacking` key to the dictionary stored in the `drawableProperties` property of the `CAEAGLLayer` object. Retaining the contents of the layer may require additional memory to be allocated, which can reduce your application's performance.

When the renderbuffer is presented to the screen, it is animated and composited with any other Core Animation layers visible on the screen, regardless of whether those layers were drawn with OpenGL ES, Quartz, or another graphics library. Mixing OpenGL ES content with other content comes at a performance penalty. For best performance, your application should rely solely on OpenGL ES to render your content. To do this, create a screen-sized `CAEAGLLayer` object, set the `opaque` property to `YES`, and ensure that no other Core Animation layers or views are visible.

If you must composite OpenGL ES content with other layers, making your `CAEAGLLayer` object opaque reduces, but doesn't eliminate, the performance penalty.

If your `CAEAGLLayer` object is blended with other layers, Core Animation incurs a significant performance penalty. You can reduce this penalty by playing your layer behind other UIKit layers.

Note: If you must blend transparent OpenGL ES content, your renderbuffer must provide a buffer with a premultiplied alpha to be composited correctly by Core Animation.

Finally, it is almost never necessary to apply Core Animation transforms to the `CAEAGLLayer` object and doing so adds to the processing Core Animation must do before displaying your content. Your application can often perform the same tasks by changing the modelview and projection matrices (or the equivalents in your vertex shader), and swapping the width and height arguments to the `glViewport` and `glScissor` functions.

Sharegroups

An `EAGLSharegroup` object manages OpenGL ES resources associated with one or more `EAGLContext` objects. A sharegroup is usually created when an `EAGLContext` object is initialized and disposed of when the last `EAGLContext` object that references it is released. As an opaque object, there is no developer-accessible API.

To create multiple contexts using a single sharegroup, your application first creates a context as before, then creates one or more additional contexts using the `initWithAPI:sharegroup:` initializer.

```
EAGLContext* firstContext = [[EAGLContext alloc]
initWithAPI:kEAGLRenderingAPIOpenGLES1];
EAGLContext* secondContext = [[EAGLContext alloc]
initWithAPI:kEAGLRenderingAPIOpenGLES1 sharegroup: [firstContext sharegroup]];
```

Sharegroups manage textures, buffers, framebuffers, and renderbuffers. It is your application's responsibility to manage state changes to shared objects when those objects are accessed from multiple contexts in the sharegroup. The results of changing the state of a shared object while it is being used for rendering in another context are undefined. To obtain deterministic results, your application must take explicit steps to ensure that the shared object is not currently being used for rendering when your application modifies it. Further, state changes are not guaranteed to be noticed by another context in the sharegroup until that context rebinds the shared object.

To ensure defined results of state changes to shared objects across contexts in the sharegroup, your application must perform the following tasks, in this order:

1. Change the state of an object.
2. Call `glFlush` on the rendering context that issues the state-modifying routines.
3. Each context must rebind the object to see the changes.

The original object is deleted after all contexts in the sharegroup have bound the new object.

Best Practices for Working with Vertex Data

Any OpenGL ES application must submit geometry to the OpenGL ES API to render the final scene. Whenever your data changes, you redraw the scene by submitting the changed vertex information. In all cases, OpenGL must efficiently process these commands to render your images quickly. Every time your application submits geometry, it must be processed and transferred to the hardware to be rendered. This chapter covers many common techniques for managing your vertex data so that it can be processed efficiently by the hardware of iOS-based devices.

If you are familiar with traditional OpenGL on Mac OS X and other platforms, you know that there are many different functions used to submit geometry to OpenGL, each with widely different uses and performance characteristics. As OpenGL matured, so did the techniques for working with vertex data. OpenGL ES dispenses with the older mechanisms in order to provide a simple interface that provides great performance.

Simplify Your Geometry

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 needed to draw an object can result in a direct reduction in the time it takes to transmit and process vertex information.

You can often reduce the complexity of your geometry by using some of the following techniques:

- Provide multiple versions of your geometry at different levels of detail, and choose an appropriate model based on the distance of the object from the camera.
- If you are using OpenGL ES 1.1 and added additional vertices to improve lighting details, you can simplify the geometry by using DOT3 lighting. To do this, you'll create a bump map texture for your object to hold normal information, and use the texture unit to apply lighting using a texture combine operation with the `GL_DOT3_RGB` mode.
- If your application is written to use OpenGL ES 2.0 shaders, you can easily move lighting and other per-vertex calculations into the fragment shader. Although processing these calculations for every fragment can be more expensive, it can result in a higher quality image. Moving calculations into the fragment shader should be used when the data changes frequently.

Avoid Storing Constants in Arrays

If your geometry uses data that remains constant across an object, you should not duplicate that data inside your vertex format. Instead, you should disable that array entirely and use a per-vertex attribute state call such as `glColor4ub` or `glTexCoord2F` instead. OpenGL ES 2.0 applications can either set a constant vertex attribute or use a uniform shader value to hold information that is constant across an object and changes infrequently.

Use Interleaved Vertex Data

In OpenGL ES, you enable each attribute your application needs and provide a pointer to an array of that data type. OpenGL ES allows you to specify a stride, which offers your application the flexibility of providing multiple arrays (also known as a *struct of arrays*) or a single array with a single vertex format (an *array of structs*).

Your application should use an array of structs with a single interleaved vertex format. Interleaved data provides better memory locality than using a separate array for each attribute.

You may want to separate out attribute data that is updated at a frequency different from the rest of your vertex data (as described in "Vertex Buffer Usage" (page 32)). Similarly, if attribute data can be shared between two or more pieces of geometry, separating it out may reduce your memory usage.

Use the Smallest Acceptable Types for Attributes.

Memory bandwidth is limited on iOS-based devices, so reducing the size of your vertex data can provide a significant boost to performance. When specifying the size of each of your components, you should use the smallest type that gives you acceptable results. Specify vertex colors using 4 unsigned byte values. Specify texture coordinates with 2 or 4 unsigned byte or short values, instead of floating-point values.

Avoid the use of the OpenGL ES `GL_FIXED` data type. It uses the same amount of bandwidth as `GL_FLOAT`, but provides a smaller range of values and requires additional processing.

If you specify smaller components, be sure you reorder your vertex format to avoid any misalignment penalties. See "Align Vertex Structures" (page 34).

Use Triangle Strips to Batch Geometry

Using triangle strips significantly reduces the number of vertex calculations that OpenGL ES must perform on your geometry. Your performance is best if an object (or even a group of objects) can be submitted in a single unindexed triangle strip using `glDrawArrays`. This may involve adding degenerate triangles to merge multiple smaller triangle strips into a single large strip.

If your geometry duplicates a large number of vertices (because vertices are shared by many triangles or because a large number of duplicate vertices were added to merge multiple triangle strips), you may obtain better performance by submitting an indexed triangle strip using `glDrawElements`. For performance, your application should sort the drawing order so that triangles that share the same vertex are drawn reasonably close to one another in the strip. Graphics hardware often caches recent vertex calculations, so submitting all the triangles that use the same vertex can allow the hardware to used the cached calculations, rather than repeating vertex calculations.

For best results, you should test your geometry using both indexed and unindexed triangle strips, and use the one that performs the fastest.

Use Vertex Buffers

By default, an application maintains its own vertex data and submits it to the hardware to be rendered. When you submit geometry, it is copied to the hardware to be rendered. Listing 4-1 shows the code a simple application would use to provide position and color information to OpenGL ES 1.1.

Listing 4-1 Submitting vertex data to OpenGL ES 1.1.

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

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

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

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

This code works, but is inefficient. Each time `glDrawElements` is called, the data is retransmitted to the graphics hardware to be rendered. If the data did not change, those additional copies are unnecessary. To avoid this, your application should store its geometry in a **vertex buffer object** (VBO). Data stored in a vertex buffer object is owned by OpenGL ES and may be cached by the hardware or driver to improve performance.

Listing 4-2 modifies the previous example to create vertex buffer objects to store the vertices and indices. In this example, the buffers are created when the application is initialized. Listing 4-3 shows how to use the vertex buffers to submit the geometry for rendering.

Listing 4-2 Creating vertex buffers in OpenGL ES 1.1

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

const vertexStruct vertices[] = {...};
const GLubyte indices[] = {...};

GLuint    vertexBuffer;
GLuint    indexBuffer;

void CreateVertexBuffers()
{

    glGenBuffers(1, &vertexBuffer);
```

```
    glGenBuffers(1, &indexBuffer);

    glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

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

}
```

Listing 4-3 Drawing using Vertex Buffers in OpenGL ES 1.1

```
void DrawUsingVertexBuffers()
{
    glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, sizeof(vertexStruct),
(void*)offsetof(vertexStruct,position));
    glEnableClientState(GL_COLOR_ARRAY);
    glColorPointer(4, GL_UNSIGNED_BYTE, sizeof(vertexStruct),
(void*)offsetof(vertexStruct,color));
    glDrawElements(GL_TRIANGLE_STRIP, sizeof(indices)/sizeof(GLubyte),
GL_UNSIGNED_BYTE, (void*)0);
}
```

The key difference in this code is that `glVertexPointer` and `glColorPointer` no longer directly point to the vertex data. Instead, the code creates a vertex buffer object and copies the vertex data into it by calling `glBufferData`. The functions `glVertexPointer` and `glColorPointer` are called with offsets into the vertex buffer object instead. This code also creates a buffer to hold the index information. Because this buffer is bound when you call `glDrawElements`, it is used as the source for indices.

Vertex Buffer Usage

Another big advantage of vertex buffers is that the application can hint how it uses the data in each vertex buffer. For example, in Listing 4-2 (page 31), the code informed OpenGL ES that the contents of both buffers were not expected to change (`GL_STATIC_DRAW`). The usage parameter allows OpenGL ES to alter its strategy for processing different types of vertex data to improve performance.

OpenGL ES supports the following usage cases:

- `GL_STATIC_DRAW` should be used for vertex data that is specified once and never changed. Your application should create these vertex buffers during initialization and use them repeatedly until your application shuts down.
- `GL_DYNAMIC_DRAW` should be used when data are expected to change after they are created. Your application should still allocate these buffers during initialization and periodically update them by calling `glBufferSubData`.

OpenGL ES 2.0 offers an additional option:

- `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 change vertex data every frame in a way that cannot be performed in a vertex shader. To use a stream vertex buffer, your application initially fills the buffer using `glBufferData`, then alternates between drawing geometry from the buffer and altering the content by calling `glBufferSubData`.

If different data in your vertex format has different usage characteristics, you may want to split the vertex data into one structure for each usage case, and allocate a vertex buffer for each. Listing 4-4 (page 33) modifies the previous example to hint that the color data may change.

Listing 4-4 Geometry with various usage patterns

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

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

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

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

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

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

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

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

void DrawUsingVertexBuffers()
{
    glBindBuffer(GL_ARRAY_BUFFER, staticBuffer);
```

```
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, sizeof(vertexStatic),
(void*)offsetof(vertexStatic,position));
    glBindBuffer(GL_ARRAY_BUFFER, dynamicBuffer);
    glEnableClientState(GL_COLOR_ARRAY);
    glColorPointer(4, GL_UNSIGNED_BYTE, sizeof(vertexDynamic),
(void*)offsetof(vertexDynamic,color));
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);
    glDrawElements(GL_TRIANGLE_STRIP, sizeof(indices)/sizeof(GLubyte),
GL_UNSIGNED_BYTE, (void*)0);
}
```

Consolidate Vertex Array State Changes Using Vertex Array Objects

iOS 4 adds the `OES_vertex_array_object` extension on all platforms. Vertex array objects provide two key optimizations. First, the call overhead to set series of attribute pointers can be reduced to a single function call inside the rendering loop. Second, because the attributes associated with a vertex array object are in a known and well-defined state, OpenGL ES can cache the layout of the vertex data and process the vertex data more quickly.

Align Vertex Structures

When you design your vertex formats, you should make sure that all components are properly aligned to their native alignment. 32-bit values such as `GL_FLOAT` should be aligned on a 32-bit boundary. 16-bit values should be aligned on a 16-bit boundary. Unaligned data requires significantly more processing, particularly when your application uses vertex buffers.

Best Practices for Working with Texture Data

In the previous chapter, you saw how a few simple rules could improve your application's ability to process vertex data. This chapter provides similar guidelines for textures.

Reduce Texture Memory Usage

The amount of memory your iOS application uses is critical. However, texture memory in OpenGL ES is even more constrained. iPhone or iPod touch devices that use the PowerVR MBX hardware have a limit on the total amount of memory they can use for textures and renderbuffers (described in "PowerVR MBX" (page 53)). Where possible, your application should minimize the amount of memory it uses for textures.

Broadly, your application must balance the size of your texture data with the quality of the final image.

Compress Textures

Texture compression is often the single largest source of memory savings. 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 20). 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 59).

Use Lower-Precision Color Formats

If your application cannot use compressed textures, you should consider using a smaller 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, you should first attempt to compress the texture or use a lower-precision color format. A texture compressed with the PVRTC format usually provides higher-quality output 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 iPhone and iPod touch devices uses a tile-based deferred renderer that makes calls to `glTexSubImage` and `glCopyTexSubImage` particularly expensive. See "Tile-Based Deferred Rendering" (page 49) for more information.

Combine Textures into Texture Atlases

Binding to a texture changes OpenGL ES state, which takes CPU time to process. Applications that reduce the number of changes they make to OpenGL ES state perform better. Reducing the number of state changes is discussed in detail in "Avoid Changing OpenGL ES State Unnecessarily" (page 42).

One way to avoid changing the texture is to combine multiple smaller textures into a single large texture, known as a **texture atlas**. Each model uses modified texture coordinates to select the desired texture from within the atlas. A texture atlas allows multiple models to be drawn without changing the bound texture and may also allow you to collapse multiple `glDrawElements` calls into a single call.

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.

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 smaller mipmaps are sampled, fewer texels are fetched from texture memory, which reduces the memory bandwidth used by the hardware and significantly improves 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.

Use Multitexturing Instead of Multiple Passes

Many applications perform multiple passes on their geometry, altering the configuration of OpenGL as they need to in order to generate the final result. This is not only expensive due to the number of state changes required, 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 support at least two texture units and your application can use these texture units to perform multiple steps of your algorithm in each pass. You can retrieve the number of texture units available to your application by calling `glGetIntegerv` with `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.
- Use the `GL_EQUAL` depth function to ensure that only pixels in your geometry are modified.

Performance Guidelines

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 not only can result in a poor frame rate, but can also dramatically reduce the battery life of an iOS-based device.

Although other chapters have already touched on many performance bottlenecks, this chapter takes a more comprehensive look at how to optimize your applications for iOS.

General Performance Recommendations

Redraw Scenes Only When Necessary

An iOS-based device continues to display a frame until your application presents a new frame to display. If the data used to render your image has not changed, your application should not reprocess the image. Your application should wait until something in the scene has changed before rendering a new frame.

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.

Use 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 boil down to a collection of mathematical functions that need to be evaluated every frame. This imposes a serious amount of arithmetic on the CPU.

The ARM processor in iPhone and iPod touch processes floating-point instructions natively. Your application should use floating-point instead of fixed point math whenever possible. If you are porting an application from another platform that does not support native floating-point math, you should rewrite the code to use floating-point types.

Note: iPhone supports 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.

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 is performing 2D drawing, it should disable fog and depth testing.

If your application is written for OpenGL ES 2.0, do not create a single shader that can be configured to perform every task your application needs to render the scene. Instead, compile multiple shader programs that perform specific, focused tasks. As with OpenGL ES 1.1, if your shader can remove unnecessary calculations, it provides faster overall performance.

This guideline must be balanced with other recommendations, such as "Avoid Changing OpenGL ES State Unnecessarily" (page 42).

Minimize the Number of Draw Calls

Every time your application submits geometry to be processed by OpenGL ES, the CPU spends time preparing the commands for the graphics hardware. To reduce this overhead, you should batch your geometry into fewer calls.

If your application draws geometry using triangle strips, you can reduce the number of submissions by merging two or more triangle strips into a single triangle strip. To do this, you add degenerate triangles formed either by two or three collinear points. For example, instead of using one call to draw a strip ABCD and another to draw a strip EFGH, you can add in the degenerate triangles CDD, DDE, DEE, and EEF to create a new strip ABCDDEEFGH. This strip can be drawn with a single submission.

In order to gather separate triangle strips into a single strip, all of the strips must share the same rendering requirements. This means:

- An OpenGL ES 1.1 based renderer must be able to render all of the triangle strips without changing any OpenGL state.
- An OpenGL ES 2.0 based renderer must share the same shader to render all of the triangle strips.
- The triangle strips must share the same vertex format, with the same vertex arrays or attributes enabled.

Consolidating geometry to use a single set of OpenGL state has other advantages in that it reduces the overhead of changing your OpenGL ES state, as documented in "Avoid Reading or Writing OpenGL ES State" (page 41).

For best results, consolidate geometry that is in close spacial proximity. Large, sprawling geometry is more difficult for your application to efficiently cull when it is not visible in the scene.

Contexts

The way your application interacts with the rest of the iOS graphics system is critical to the performance of your application. These interactions are documented in detail in "Displaying Your Results" (page 26).

Memory

Memory is a scarce resource on iOS-based 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, you should allocate only objects that you need and deallocate them when your application no longer needs them. For example, any of these scenarios can save memory:

- After loading an image into your OpenGL ES texture using `glTexImage`, you can 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 total resources. For example, a game could be divided into levels, each with resources that fit inside the resource limits.

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 Application Programming Guide*.

The PowerVR MBX processor found in some iPhone and iPod touch models has additional memory requirements. See "PowerVR MBX" (page 53) for more information.

Avoid Reading or Writing OpenGL ES State

Every time you read or write OpenGL ES state, the CPU spends time processing the command before sending it to the hardware. Occasionally, accessing OpenGL ES state may force previous operations to be completed before the state can be accessed. For best performance, your application should try to touch OpenGL ES state as infrequently as possible.

Avoid Querying OpenGL ES State

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

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

Although it is critical to call `glGetError` in a debug build of your application, calling `glGetError` in the release version of your application degrades performance.

Avoid Changing OpenGL ES State Unnecessarily

Changing OpenGL ES state requires the hardware to be updated with new information, which may cause cause hardware to stall or force it to execute previously submitted commands. Your application can reduce the number of state changes it requires by following these guidelines:

- Use common drawing strategies to render objects in your application.
- Sort the objects in your scene by the OpenGL ES state you need to set to draw those objects, so that you can set that state once and draw multiple objects at once. This may require that objects share common state. You should avoid save-change-restore sequences.
- Do not change OpenGL ES state redundantly. For instance, if lighting is already enabled, do not call `glEnable(GL_LIGHTING)` again.

Drawing Order

- Do not waste CPU time sorting objects front to back. OpenGL ES for iPhone and iPod touch implement a tile-based deferred rendering model that makes this unnecessary. See "Tile-Based Deferred Rendering" (page 49) 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 47).
  3. Finally, draw alpha-blended objects.

Lighting

Simplify lighting as much as possible. 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. For example, consider using directional lights instead of spot lighting, which incurs a higher performance cost. Consider simpler lighting equations in your shaders over more complex ones.
- Pre-compute your lighting and store the color values in a texture that can be sampled by fragment processing.

OpenGL ES 2.0 Shaders

Shaders present additional areas where you can improve your application's performance.

Compile and Link Shaders During Initialization

Creating a shader program is an expensive operation compared to other OpenGL ES state changes. Listing 6-1 presents a typical strategy to load, compile, and verify a shader program.

Listing 6-1 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);
```

You should 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. Further, 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 should validate all shaders to ensure that no errors occurred, as shown above in Listing 6-1 (page 43).

Your application must query the limits of the OpenGL ES implementation and not use shaders that exceed these limitations. Your application should call `glGetIntegerv()` for each value at startup and choose shaders that match the capabilities of the OpenGL ES implementation.

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

For all types, the query returns the number of 4-component floating-point vectors available. Your variables are packed into these vectors as described in the OpenGL ES shading language specification.

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 shaders should specify a default precision, and individual shader variables should override this to provide hints to the compiler on how to efficiently compile your shader. Although these hints may be disregarded by an OpenGL ES implementation, they can also be used by the compiler to generate more efficient shaders.

Which precision hint to use for your shading variables depends on each variable's requirements for range and precision. High-precision variables are interpreted as single precision floating-point values. Medium-precision variables are interpreted as half-precision floating-point values. Finally, low-precision qualified variables are interpreted with 8 bits of precision and a range of -2 to +2.

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

Although high precision is recommended for vertex data, other variables do not need this level of precision. For example, the fragment color assigned to the framebuffer can often be implemented in low precision without a significant loss in image quality, as demonstrated in Listing 6-2.

Listing 6-2 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);
}
```

Start with high-precision variables and then reduce the precision on variables that do not need this range and precision, testing your application along the way to ensure that your program still runs correctly.

Be Cautious of Vector Operations

Not all operations are performed in parallel by the graphics processor. Although vector operations are useful, you should not overuse the vector processor.

For example, the code in Listing 6-3 takes two operations to complete on a SIMD vector processor, because of the parentheses, on a scalar processor, this would require eight separate operations.

Listing 6-3 Poor use of vector operators

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

The same operation can be performed more efficiently by shifting the parentheses as shown in Listing 6-4:

Listing 6-4 Proper use of vector operations

```
highp float f0, f1;
highp vec4 v0, v1;
// On a scalar processor, this requires only 5 operations.
v0 = v1 * (f0 * f1);
```

Similarly, if your application can specify a write mask for a vector operation, it should do so. A scalar processor can ignore unused components.

Listing 6-5 Specifying a write mask

```
highp vec4 v0;
highp vec4 v1;
highp vec4 v2;
// On a scalar processor, this may be twice as fast when the write mask is
specified.
v2.xz = v0 * v1;
```

Use Uniform or Constants Instead of Computation Within a Shader

Whenever a value can be calculated outside the shader, you should pass it into the shader as a uniform or a constant. Calculating and using dynamic values can potentially be very expensive in a few circumstances.

Avoid Branching

Branches are discouraged within shaders, as they can reduce the ability to execute operations in parallel on 3D graphics processors. If your shaders must branch, it is more efficient to branch on a GLSL uniform variable or a constant known when the shader is compiled. Branching on a value computed in the shader can potentially be expensive. A better solution may be to create shaders specialized for specific rendering tasks. There's a tradeoff between reducing the number of branches in your shaders and increasing the number of shaders you create. You should test different scenarios and choose the fastest solutions.

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.

Array Access

Using indices computed in the shader is more expensive than a constant or uniform array index.

Dynamic Texture Lookups

Also known as *dependent texture reads*, a dynamic texture lookup occurs when the shader computes or modifies texture coordinates used to sample a texture. Although GLSL supports this, it can incur a substantial performance penalty to do so. If the shader has no dependent texture read, the texture sampling hardware can fetch texels sooner and hide the latency of accessing memory.

Avoid Alpha Test and Discard

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 calculated completely before being discarded.

An alternative to using alpha test or discard to kill pixels is to use alpha blending with alpha forced to zero. This can be implemented by looking up an alpha value in a texture. 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.

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

Platform Notes

Apple provides different implementations of OpenGL ES for different hardware platforms. Each of these implementations has different limitations, from the maximum size allowed for textures to the list of OpenGL ES extensions that implementation supports. 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 4.0 but is subject to change in future hardware or software. For best results, your application must test for these capabilities at runtime, as described in "Determining OpenGL ES Capabilities" (page 17).

PowerVR SGX Platform

The PowerVR SGX is the graphics processor in the iPhone 3GS and is designed for 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 submit OpenGL ES commands for rendering, the PowerVR SGX defers rendering until it accumulates a large list of rendering commands and then operates on this list as a single action. The framebuffer is then divided into tiles, and the scene is drawn once for each tile, with each tile drawing only the content that is visible within it. The key advantage to a deferred renderer is that it accesses memory more 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 scene. To gain the most benefit from this feature, you should try to draw as much of the scene 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 its geometry 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 (or duplicate resources to avoid flushing the scene), the renderer loses much of its efficiency.

For example, if your application updates a texture in the middle of a frame by calling `glTexSubImage`, the renderer may need to keep both the modified and previous versions of the texture at the same time, increasing memory usage in your application. Similarly, any attempt to read pixel data from the framebuffer requires that preceding commands be processed if they would alter that framebuffer.

Best Practices on the PowerVR SGX

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

- Avoid operations that depend on previous commands. If you need to execute these commands, schedule these operations at the beginning or end of a frame. These commands include `glTexSubImage`, `glCopyTexImage`, `glCopyTexSubImage`, `glReadPixels`, `glBindFramebufferOES`, `glFlush`, and `glFinish`.
- To take advantage of the processor's hidden surface removal, follow the drawing guidelines found in "Drawing Order" (page 42).
- Vertex buffer objects (VBOs) provide a performance improvement on the PowerVR SGX. See "Use Vertex Buffers" (page 31) for information on how to use these in your application.
- Starting in iOS 3.1, setting texture filter settings (`glTexParameter`) before loading texture images (`glTexImage2D`) allows OpenGL ES to optimize its memory usage and texture load times.
- Starting in iOS 4, the performance of `glTexImage2D` and `glTexSubImage2D` have been significantly improved.

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 your fragment shaders. You cannot use texture lookups in your vertex shaders.
- 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 your vertex shaders and up to 64 in your fragment shaders.
- 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:

- GL_OES_depth24
- GL_OES_fbo_render_mipmap

- GL_OES_mapbuffer
- GL_OES_packed_depth_stencil
- GL_OES_rgb8_rgba8
- GL_OES_standard_derivatives
- GL_EXT_blend_minmax
- GL_IMG_read_format
- GL_IMG_texture_compression_pvrtc
- GL_IMG_texture_format_BGRA8888
- APPLE_framebuffer_multisample
- EXT_discard_framebuffer
- OES_vertex_array_object
- APPLE_texture_max_level
- EXT_shader_texture_lod
- OES_depth_texture
- OES_texture_float
- OES_texture_half_float
- APPLE_rgb_422

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 following are known limitations as of iOS 4.0:

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

Best Practices on OpenGL ES 2.0

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 "Be Cautious of Vector Operations" (page 45) 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. Using a low-precision float as a parameter or assigning the result to a low-precision float adds conversion overhead.

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 efficiently implemented using shaders that are customized as your application changes OpenGL ES state. Because of this, OpenGL ES state changes may be more expensive than they would be on a traditional hardware implementation. You can improve the performance of your application by reducing the number of state changes it performs. For more information, see "Avoid Changing OpenGL ES State Unnecessarily" (page 42).

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

- GL_OES_blend_equation_separate
- GL_OES_blend_func_separate
- GL_OES_blend_subtract
- GL_OES_compressed_paletted_texture
- GL_OES_depth24
- GL_OES_draw_texture
- GL_OES_fbo_render_mipmap
- GL_OES_framebuffer_object
- GL_OES_mapbuffer
- GL_OES_matrix_palette
- GL_OES_packed_depth_stencil
- GL_OES_point_size_array
- GL_OES_point_sprite
- GL_OES_read_format
- GL_OES_rgb8_rgba8
- GL_OES_stencil8
- GL_OES_texture_mirrored_repeat

- GL_EXT_blend_minmax
- GL_EXT_texture_lod_bias
- GL_IMG_texture_compression_pvrtc
- GL_IMG_read_format
- GL_IMG_texture_format_BGRA8888
- GL_APPLE_texture_2D_limited_npot
- GL_IMG_texture_format_BGRA8888
- APPLE_framebuffer_multisample
- EXT_discard_framebuffer
- OES_vertex_array_object
- APPLE_texture_max_level

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 traditional pipeline is still accelerated by avoiding unnecessary fragment processing. See "Tile-Based Deferred Rendering" (page 49) 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.

Best Practices on the PowerVR MBX

- For best performance, you should 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.
- Using the `OES_vertex_array_object` extension to consolidate vertex array state changes significantly improves performance on the PowerVR MBX.

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 iPhone and iPod touch are:

- GL_OES_blend_subtract
- GL_OES_compressed_paletted_texture
- GL_OES_depth24
- GL_OES_draw_texture
- GL_OES_framebuffer_object
- GL_OES_mapbuffer
- GL_OES_matrix_palette
- GL_OES_point_size_array
- GL_OES_point_sprite
- GL_OES_read_format
- GL_OES_rgb8_rgba8
- GL_OES_texture_mirrored_repeat
- GL_EXT_texture_filter_anisotropic
- GL_EXT_texture_lod_bias
- GL_IMG_read_format
- GL_IMG_texture_compression_pvrtc
- GL_IMG_texture_format_BGRA8888
- APPLE_framebuffer_multisample
- EXT_discard_framebuffer
- OES_vertex_array_object

- APPLE_texture_max_level

Known Limitations and Issues

The PowerVR MBX implementation of OpenGL ES 1.1 has a number of limitations that are not shared by iPhone Simulator or the PowerVR SGX. :

- The texture magnification and minification filters (within a texture level) must match. For example:
  - 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.

iPhone Simulator

iPhone 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 on either the PowerVR MBX or the PowerVR SGX.
- Simulator does not attempt to simulate either the PowerVR MBX or the PowerVR SGX. Images generated by the simulator are not a pixel-for-pixel match with those generated on the device.

Important: It is important to understand that 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 OS simulator (depending on your computer and what you are drawing) than on an actual device. You should 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:

- GL_OES_depth24
- GL_OES_fbo_render_mipmap
- GL_OES_mapbuffer
- GL_OES_rgb8_rgba8
- GL_EXT_blend_minmax
- GL_IMG_read_format
- GL_IMG_texture_compression_pvrtc
- GL_IMG_texture_format_BGRA8888
- APPLE_framebuffer_multisample
- EXT_discard_framebuffer
- OES_vertex_array_object
- APPLE_texture_max_level
- EXT_shader_texture_lod
- OES_depth_texture
- OES_texture_float

- OES_texture_half_float
- APPLE_rgb_422

OpenGL ES 1.1 on Simulator

Supported Extensions

- GL_OES_blend_equation_separate
- GL_OES_blend_func_separate
- GL_OES_blend_subtract
- GL_OES_compressed_paletted_texture
- GL_OES_depth24
- GL_OES_draw_texture
- GL_OES_fbo_render_mipmap
- GL_OES_framebuffer_object
- GL_OES_mapbuffer
- GL_OES_matrix_palette
- GL_OES_point_size_array
- GL_OES_point_sprite
- GL_OES_read_format
- GL_OES_rgb8_rgba8
- GL_OES_stencil8
- GL_OES_texture_mirrored_repeat
- GL_EXT_blend_minmax
- GL_EXT_texture_filter_anisotropic
- GL_EXT_texture_lod_bias
- GL_IMG_texture_compression_pvrtc
- GL_IMG_read_format
- GL_IMG_texture_format_BGRA8888
- GL_APPLE_texture_2D_limited_npot
- APPLE_framebuffer_multisample
- EXT_discard_framebuffer
- OES_vertex_array_object
- APPLE_texture_max_level

Using texturetool to Compress Textures

The iPhone 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 3.0 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 iPhone 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
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.

Document Revision History