Drawing with OpenGL ES

The Open Graphics Library (OpenGL) is a cross-platform C-based interface used to create 2D and 3D content on desktop systems. It is typically used by games developers or anyone needing to perform drawing with high frame rates. You use OpenGL functions to specify primitive structures such as points, lines, and polygons and the textures and special effects to apply to those structures to enhance their appearance. The functions you call send graphics commands to the underlying hardware, where they are then rendered. Because rendering is done mostly in hardware, OpenGL drawing is usually very fast.

OpenGL for Embedded Systems is a pared-down version of OpenGL that is designed for mobile devices and takes advantage of modern graphics hardware. If you want to create OpenGL content for iPhone OS–based devices—that is, iPhone or iPod Touch—you'll use OpenGL ES. The OpenGL ES framework (OpenGLES.framework) provided with iPhone OS conforms to the OpenGL ES v1.1 specification. You can find out more about OpenGL ES by reading Polygons In Your Pocket: Introducing OpenGL ES.

This section is designed to get you started writing OpenGL ES applications for iPhone OS–based devices. "Setting Up a Rendering Surface" provides step-by-step instructions for creating a surface that you can draw to using OpenGL ES. But before you start writing the OpenGL ES portion of your application, you'll want to read "Implementation Details" to learn about the capabilities of iPhone OS–based devices and the specifics of the OpenGL ES implementation in iPhone OS. "Best Practices" provides coding guidelines that can help your application perform optimally.

Setting Up a Rendering Surface

The setup for drawing with OpenGL ES is straightforward and requires the same types of tasks you'd perform to draw to surfaces on a Macintosh computer. The primary difference is that you'll use the EAGL API to set up the window surface instead of an API such as CGL or AGL. The EAGL API provides the interface between the OpenGL ES renderer and the windows and views of an iPhone application. (See OpenGL ES Framework Reference.)

When you set up your Xcode project, make sure you link to OpenGLES.framework. Then, set up a surface for rendering by following these steps:

1. Subclass UIView and set up the view appropriately for your iPhone application.
2. Override the layerClass method of the UIView class so that it returns 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’s recommended that you mark the layer as opaque by setting the opaque property provided by the CALayer class. See “Best Practices.”

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.

   The EAGL framework lets you specify custom color formats and whether or not the native surface retains its contents after presenting them. You identify these properties in the dictionary using the kEAGLDrawablePropertyColorFormat and kEAGLDrawablePropertyRetainedBacking keys. For a list of values for these keys, see the EAGLDrawable protocol.

6. Create a new EAGLContext object to manage the drawing context. You typically create and initialize this object as follows:

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

   If you want to share objects (texture objects, vertex buffer objects, and so forth) between multiple contexts, use the initWithAPI:sharegroup: initialization method instead. For each context that should share a given set of objects, pass the same EAGLSharegroup object in the sharegroup parameter.

7. Make your EAGLContext object the context for the current thread using the setCurrentContext: class method.

   You can have one current context per thread.

8. Create and bind a new render buffer to the GL_RENDERBUFFER_OES target. (Typically you would do this in a two-step process using the glGenRenderbufferOES function to allocate an unused name and the glBindRenderbufferOES functions create and bind the render buffer to that name.)

9. Attach the newly created render buffer target to your view’s layer object using the renderBufferStorage:fromDrawable: method of your EAGLContext object. (The layer provides the underlying storage for the render buffer.) For example, given the context created previously, you would use the following code to bind the buffer to the view’s layer (obtained previously and stored in the myEAGLLayer variable):

```
[myContext renderbufferStorage:GL_RENDERBUFFER_OES
fromDrawable:myEAGLLayer];
```

The width, height, and format of the render buffer storage are derived from the bounds and properties of the CAEAGLLayer object at the moment you call the renderbufferStorage:fromDrawable: method. If you change the layer's bounds later, Core Animation scales the content by default. To avoid scaling, you must recreate the renderbuffer storage by calling renderbufferStorage:fromDrawable:.

10. Configure your frame buffer as usual and bind your render buffer to the attach points of your frame buffer.

Best Practices

To create great OpenGL ES applications that run optimally, whenever possible follow the guidelines discussed in this section. You'll also want to read "Implementation Details" to see how your code can take advantage of the specific features of the GPU.

General Guidelines

When developing OpenGL ES applications for iPhone OS–based devices:

* Use an EAGL surface that is the same size as size of the screen. Read the bounds property of the UIScreen class to get the screen size.
* Do not apply any Core Animation transforms to the CAEAGLLayer object that contains the EAGL window surface.
* Set the opaque property of the CALayer class to mark the CAEAGLLayer object as opaque.
* Do not place any other Core Animation layers or UIKit views above the CAEAGLLayer object that you're rendering to.
* If your application needs to present landscape content, avoid transforming the layer. Instead, set the OpenGL ES state to rotate everything by changing the Model/View/Projection transform, and swapping the width and height arguments to the glViewport and glScissor functions.
* Limit interactions between OpenGL ES and UIKit or Core Animation rendering. For example, avoid rendering with OpenGL ES while you render notifications, messages, or any other user interface controls provided by UIKit or Core Animation.
* Economize memory usage. iPhone OS–based devices use a shared memory system. Memory used by your application's graphics is not available for the system. For example, after loading GL textures, free your copy of the pixel data if you no longer need to use it. (See "Memory")
* To implement transparency, use alpha blending instead of alpha testing.

Alpha testing is considerably more expensive than alpha blending.

*

Disable unused or unnecessary features. For instance, do not enable lighting or blending if you do not need them.

*

Minimize scissor state changes. They are considerably more expensive on OpenGL ES for iPhone OS than they are on OpenGL for Mac OS X.

*

Minimize the number of times you call the EAGLContext class method setCurrentContext: during a rendering frame. Changing the current surface on OpenGL ES for iPhone OS is considerably more expensive than it is on OpenGL for Mac OS X. Rather than using multiple contexts, consider using multiple frame buffer objects instead.

*

Operations that depend on completing previous rendering commands (such as glTexSubImage, glCopyTexImage, glCopyTexSubImage, glReadPixels, glFlush, glFinish, or setCurrentContext:) can be very expensive if you perform them in the middle of a frame. If you need these operations, perform them at the beginning or end of a frame.

CPU Usage

Respecifying OpenGL ES state can cause the CPU to perform unnecessary work. Use techniques to reduce CPU usage. For example, use a texture atlas. This allows you to perform additional draw calls without changing the texture binding. (Or even better, collapse multiple draw calls into one.) If you create a texture atlas, you also need to sort state calls to avoid rebinding the texture. Otherwise, you won’t see a performance benefit.
Vertex Data

When creating geometry:

*

Reduce overall geometry by doing such things as using indexed triangle strips and providing only as much detail as the user can see.

*

To achieve fine detail in your drawing without increasing the vertex count, consider using DOT3 lighting or textures.

*

Memory bandwidth is limited, so use the smallest acceptable type for your data. 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, if you can.

Textures

To get the best performance with textures:

*

Use textures with the smallest size per pixel that you can afford. If possible, use 565 textures instead of 8888.

*

Use compressed textures stored in the PVRTC format. See the

specification for the extension GL_IMG_texture_compression_pvrtc.

* Use mipmapping with the LINEAR_MIPMAP_NEAREST option.

* Create and load all textures prior to rendering. Don’t upload or modify textures during a frame. Specifically, avoid calling the functions glTexSubImage or glCopyTexSubImage in the middle of a frame.

* Use multitexturing rather than applying textures over multiple passes.

Drawing Order

Drawing order is important for hardware that uses tile based deferred rendering. (See “Rendering Path.”)

* Don’t waste CPU time sorting objects front to back. The tile based deferred rendering model used by the GPU makes sorting unnecessary.

* Draw opaque objects first; draw alpha blended objects last.

Lighting

Simplify lighting as much as possible.

* 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.

* If you can, precompute lighting. Static lighting gives better performance than dynamic. You can compute the lighting ahead of time and store the result in textures or color arrays that you can look up later.

Debugging and Tuning

The Instruments application includes an OpenGL ES instrument that you can use to gather information about the runtime behavior of your OpenGL code. In addition, you can set a breakpoint on the opengl_error_break symbol in GDB to see when OpenGL errors are generated.

Implementation Details

Understanding the features of the hardware and the specifics of the implementation of OpenGL ES can help you tailor your code to get the best performance. Use the information in this section, along with “Best Practices,” as you design your OpenGL ES application.

OpenGL ES Implementation

The OpenGL ES implementation in iPhone OS differs from other implementations of OpenGL ES in the following ways:

* The maximum texture size is 1024 x 1024.

- 2D texture targets are supported; other texture targets are not.
- Stencil buffers aren’t available.

Hardware Capabilities

When developing any OpenGL application, it’s important to check for the functionality that your application uses and have a contingency in place if the hardware doesn’t support the particular feature or extension that you want to use. This is true for Macintosh computers and it is even more important when writing OpenGL ES applications for iPhone OS–based devices. It’s essential that you understand the capabilities of the specific hardware that you are writing for. Keep in mind that, unlike OpenGL on Macintosh computers, there is no software rendering fallback option for iPhone OS–based devices.

The graphics hardware for iPhone OS–based devices has the following limitations:

- The texture magnification and magnification 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_LINEAR_MIPMAP_NEAREST
  - 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 an 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.

Supported Extensions

These are the OpenGL ES extensions that you can use when developing OpenGL ES applications for iPhone OS–based devices:

* 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

Memory

OpenGL ES applications should use no more than 24 MB of memory for both textures and surfaces. This 24 MB is not dedicated graphics memory but comes from the main system memory. Because main memory is shared with other iPhone

applications and the system, your application should use as little of it as possible. “Best Practices” provides several guidelines for ways to use memory economically. In particular, see “General Guidelines” and “Vertex Data.”
Rendering Path

The GPU in the iPhone and iPod touch is a PowerVR MBX Lite. This GPU uses a technique known as Tile Based Deferred Rendering (TBDR). When you submit OpenGL ES commands for rendering, TBDR behaves very differently from a streaming renderer. A streaming renderer simply executes rendering commands in order, one after another. In contrast, a TBDR defers any rendering until it accumulates a large number of rendering commands, and then operates on this command list as a single scene. The framebuffer is divided up into a number of tiles, and the scene is drawn once for each tile, each time drawing only the content that is actually visible within that tile. The TBDR approach has several advantages and disadvantages compared to streaming renderers. Understanding these differences will help you write better performing software.

The most significant advantage of TBDR is that it can make much more efficient use of available bandwidth to memory. Constraining rendering to only one tile allows the GPU to more effectively cache the framebuffer, making depth testing blending much more efficient. Otherwise, the memory bandwidth consumed by these framebuffer operations often becomes a significant performance bottleneck.

When using deferred rendering, some operations become more expensive. For example, if you call the function glTexSubImage in the middle of a frame, the accumulated command list may include commands from both before and after the call to glTexSubImage. This command list needs to reference both the old and new version of the texture image at the same time, forcing the entire texture to be duplicated even if only a small portion of the texture is updated. Duplication can make functions such as glTexSubImage significantly more expensive on a deferred renderer than a streaming renderer.

The PowerVR GPU relies on more than just TBDR to optimize performance; it performs hidden surface removal before fragment processing. If the GPU determines that a pixel won’t be visible, it discards the pixel without performing texture sampling or fragment color calculations. Removing hidden pixels can significantly improve performance for scenes that have obscured content. 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 and alpha testing.

For more information on exactly how these features are implemented and how your application can best take advantage of them, see PowerVR Technology Overview and PowerVR MBX 3D Application Development Recommendations.
Simulator Capabilities

The iPhone simulator includes a complete and conformant implementation of OpenGL ES 1.1 that you can use for your application development. This implementation differs in a few ways from the implementation found in iPhone OS–based devices. In particular, the simulator does not have the same limitations regarding texture magnification filters or texture environment operations that are described in “Hardware Capabilities.” In addition, the simulator supports antialiased lines while iPhone OS–based devices do not.

Important: It is important to understand that the rendering performance of OpenGL ES in the simulator has no relation to the performance of OpenGL ES on an actual device. The 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 Mac OS X (depending on your computer and what you are drawing) than on an actual

device. Therefore, you should always profile and optimize your drawing code on a real device and not assume that the simulator reflects real-world performance.

The following sections provide additional details about the OpenGL ES support available in the iPhone simulator.

Supported Extensions

The iPhone simulator supports all of the OpenGL ES 1.1 core functionality and most of the extensions supported by iPhone OS–based devices. The following extensions are not supported by the simulator, however:

* GL_OES_draw_texture
* GL_OES_matrix_palette
* GL_EXT_texture_filter_anisotropic
* GL_IMG_texture_compression_pvrtc

For a list of the extensions supported by the hardware, see “Supported Extensions.”

Memory

On a device, OpenGL ES applications can use no more than 24 MB of memory for both textures and surfaces. The simulator does not enforce this limit. As a result, your code can allocate as much memory as your computer’s rendering hardware supports. Be sure to keep track of the size of your assets during development.

Rendering Path

In contrast to the Tile Based Deferred Rendering technique used in devices, the simulator’s software rasterizer uses a traditional streaming model for OpenGL ES commands. Objects are transformed and rendered immediately as you specify them. Consequently, the performance of some operations can differ significantly from that on actual devices.

As with any two different implementations of OpenGL ES, there may be small differences between the pixels rendered by the simulator and those rendered by the device. For example, OpenGL ES allows some calculations, such as color interpolation and texture mipmap filtering, to be approximated. In general, the two implementations will produce similar results, but do not rely on them to be bit-for-bit identical.

For More Information

You may want to consult these resources as you develop OpenGL ES applications for iPhone OS–based devices:

* OpenGL ES 1.X Specification is the official definition of this technology provided by the Khronos Group. You’ll also find other useful information on this website.
* PowerVR MBX OpenGL ES 1.x SDK page provides information about the specific OpenGL ES implementation supported by the PowerVR MBX graphics

hardware.

* OpenGL ES 1.1 Reference Pages provides a complete reference to OpenGL ES specification, indexed alphabetically as well as by theme.

* OpenGL ES Framework Reference describes the functions and constants that provide the interface between OpenGL ES and the iPhone user interface.

< Previous PageNext Page > Show TOC

Last updated: 2008-11-12