Multitasking, High Resolution, and Other iOS Features

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

Implementing a Multitasking-Aware OpenGL ES App

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

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

Background Apps May Not Execute Commands on the Graphics Hardware

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

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

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

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

To summarize, your app needs to call the `glFinish` function to ensure that all previously submitted commands are drained from the command buffer and are executed by OpenGL ES. After

it moves into the background, you must avoid all use of OpenGL ES until it moves back into the foreground.

Delete Easily Re-Created Resources Before Moving to the Background

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

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

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

Here's how you should handle the two scenarios:

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

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

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

Supporting High-Resolution Displays

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

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

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

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

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

- Optimize your fragment shader's performance using the performance-tuning guidelines found in this document.
- Implement a simpler algorithm in your fragment shader. By doing so, you are reducing the quality of individual pixels to render the overall image at a higher resolution.
- Use a fractional scale factor between 1.0 and and the screen's scale factor. A scale factor of 1.5 provides better quality than a scale factor of 1.0 but needs to fill fewer pixels than an image scaled to 2.0.
- Use lower-precision formats for your `GLKView` object's `drawableColorFormat` and `drawableDepthFormat` properties. By doing this, you reduce the memory bandwidth required to operate on the underlying renderbuffers.
- Use a lower scale factor and enable multisampling. An added advantage is that multisampling also provides higher quality on devices that do not support high-resolution displays.

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

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

Supporting Multiple Interface Orientations

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

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

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

Presenting OpenGL ES Content on External Displays

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

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

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