Concurrency and OpenGL ES

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

Each process in iOS consists of one or more threads. A *thread* is a stream of execution that runs code for the process. Apple offers both traditional threads and a feature called **Grand Central Dispatch (GCD)**. Using Grand Central Dispatch, you can decompose a task into subtasks without manually managing threads. GCD allocates threads based on the number of cores available on the device and automatically schedules tasks to those threads.

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

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

Deciding Whether You Can Benefit from Concurrency

Creating a multithreaded app requires significant effort in the design, implementation, and testing of your app. Threads also add complexity and overhead. Your app may need to copy data so that it can be handed to a worker thread, or multiple threads may need to synchronize access to the same resources. Before you attempt to implement concurrency in an OpenGL ES app, optimize your OpenGL ES code in a single-threaded environment using the techniques described in OpenGL ES Design Guidelines. Focus on achieving great CPU-GPU parallelism first and then assess whether concurrent programming can provide additional performance.

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

- The app performs many tasks on the CPU that are independent of OpenGL ES rendering. Games, for example, simulate the game world, calculate artificial intelligence from computer-controlled opponents, and play sound. You can exploit parallelism in this scenario because many of these tasks are not dependent on your OpenGL ES drawing code.
- Profiling your app has shown that your OpenGL ES rendering code spends a lot of time in the CPU. In this scenario, the GPU is idle because your app is incapable of feeding it commands fast enough. If your CPU-bound code has already been optimized, you may be able to improve its performance further by splitting the work into tasks that execute concurrently.

If your app is blocked waiting for the GPU, and has no work it can perform in parallel with its OpenGL ES drawing, then it is not a good candidate for concurrency. If the CPU and GPU are both idle, then your OpenGL ES needs are probably simple enough that no further tuning is needed.

OpenGL ES Restricts Each Context to a Single Thread

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

OpenGL ES is not reentrant. If you modify the same context from multiple threads simultaneously, the results are unpredictable. Your app might crash or it might render improperly. If for some reason you decide to set more than one thread to target the same context, then you must synchronize threads by placing a mutex around all OpenGL ES calls to the context. OpenGL ES commands that block—such as `glFinish`—do not synchronize threads.

GCD and `NSOperationQueue` objects can execute your tasks on a thread of their choosing. They may create a thread specifically for that task, or they may reuse an existing thread. But in either case, you cannot guarantee which thread executes the task. For an OpenGL ES app, that means:

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

Strategies for Implementing Concurrency in OpenGL ES Apps

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

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

Multithreaded OpenGL ES

Whenever your application calls an OpenGL ES function, OpenGL ES processes the parameters to put them in a format that the hardware understands. The time required to process these commands varies depending on whether the inputs are already in a hardware-friendly format, but there is always overhead in preparing commands for the hardware.

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

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

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

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

Perform OpenGL ES Computations in a Worker Task

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

The approach described in Figure 6-6 alternates between updating OpenGL ES objects and executing rendering commands that use those objects. OpenGL ES renders on the GPU in parallel with your app's updates running on the CPU. If the calculations performed on the CPU take more processing time than those on the GPU, then the GPU spends more time idle. In this situation, you may be able to take advantage of parallelism on systems with multiple CPUs. Split your OpenGL ES rendering code into separate calculation and processing tasks, and run them in parallel. One task produces data that is consumed by the second and submitted to OpenGL.

For best performance, avoid copying data between tasks. Rather than calculating the data in one task and copying it into a vertex buffer object in the other, map the vertex buffer object in the setup code and hand the pointer directly to the worker task.

If you can further decompose the modifications task into subtasks, you may see better benefits. For example, assume two or more vertex buffer objects, each of which needs to be updated before submitting drawing commands. Each can be recalculated independently of the others. In this scenario, the modifications to each buffer becomes an operation, using an `NSOperationQueue` object to manage the work:

1. Set the current context.
2. Map the first buffer.
3. Create an `NSOperation` object whose task is to fill that buffer.
4. Queue that operation on the operation queue.
5. Perform steps 2 through 4 for the other buffers.
6. Call `waitUntilAllOperationsAreFinished` on the operation queue.
7. Unmap the buffers.
8. Execute rendering commands.

Use Multiple OpenGL ES Contexts

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

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

Guidelines for Threading OpenGL ES Apps

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

- Use only one thread per context. OpenGL ES commands for a specific context are not thread safe. Never have more than one thread accessing a single context simultaneously.
- When using GCD, use a dedicated serial queue to dispatch commands to OpenGL ES; this can be used to replace the conventional mutex pattern.
- Keep track of the current context. When switching threads it is easy to switch contexts inadvertently, which causes unforeseen effects on the execution of graphic commands. You must set a current context when switching to a newly created thread and clear the current context before leaving the thread.