**[Rendering With Two Threads](http://handsomeliuyang.iteye.com/blog/1296771)**

**博客分类：**

* [Replica Island](http://handsomeliuyang.iteye.com/category/194253)

<http://handsomeliuyang.iteye.com/blog/1296771>

The Replica Island renderer is based heavily on the GLSurfaceView class that ships with the Android SDK. I've made a couple of modifications but the code is pretty similar to the regular version: a derivation of GLSurfaceView.Renderer that draws the frame gets called every frame, followed by a call to eglSwapBuffers() to actually display the rendered frame.   
  
GLSurfaceView provides a way to run user code in the same thread as the renderer. This makes writing games pretty easy; you can just implement a Runnable, implement a Renderer, stick them both into a GLSurfaceView and get stuff moving around on the screen. Indeed, it's more than sufficient for many applications; my [SpriteMethodTest](http://code.google.com/p/apps-for-android/source/browse/trunk/SpriteMethodTest) demo works this way just fine.  
  
But for Replica Island I took a different approach. The problem with the single GLSurfaceView thread is that eglSwapBuffers() must block on the hardware until the previous frame finishes drawing. That means that even if you have nothing to draw, a call to eglSwapBuffers() takes 16.67ms to complete. (And of course, if you have a lot to draw, it could take a lot longer).  
  
Now, just in case you are not used to thinking in terms of milliseconds, here's a quick primer. To achieve the magical "60 frames per second" that many games strive for, you need to have a new frame displayed to the user every 16.67 ms. If you go for 30 fps, you have ~32 ms to complete a frame. All your game code, plus all your OpenGL code, plus the actual time it takes to draw the frame must fit within 16.67 ms to achieve 60fps.   
  
In Replica Island, the game code is fairly heavy-weight. I have all that collision to run, plus updates of all the active entities on the screen, plus sound playback and all that jazz. Turns out that it's usually more work to calculate a single simulation step than it is to actually draw the frame. Since this code takes time to execute, the 16 ms block that eglSwapBuffers() incurs makes it really hard to hit 60 fps. What I really want to be able to do is run game code while eglSwapBuffers() is blocking; that way I can pipeline the game updates while the hardware is busy drawing the frame.  
  
So I split the game code off into a separate thread. This makes three threads, by the way: the main UI thread that all Activities have by default, the GLSurfaceView render thread, and this new game thread (actually, there are a few more that are generated by the system for things like orientation sensor updates, but they don't affect the equation much). Now my game code and my renderer can run asynchronously, and I win back some of that time spent in eglSwapBuffers().  
  
Now comes the tricky part. I have two threads running in parallel that need to sync up once a frame so that the game thread can tell the render thread what to do. There's a lot of ways to go about synchronizing these two threads, but I went with a double buffer solution. The game thread fills up a buffer of commands to draw the next frame, and when it is ready it waits for the render thread to begin the next frame. At that point, the buffer is passed to to the render, which can then go off and draw the next frame asynchronously. The buffer that was used to draw the last frame is passed back to the game thread, which fills it up again the next frame. So drawing is the process of swapping these two buffers back and forth during a (hopefully short) choke point at which both threads stop and communicate.  
  
This solution was attractive to me because it was simple, and so far it seems to be plenty fast. However, another solution might be to have a queue that is shared by both threads, with the game thread pushing commands in one end and the renderer executing commands out of the other. In theory such a solution wouldn't need both threads to ever perfectly align--blocking would only occur when one thread or the other was starved. But I haven't done this yet because it is going to be significantly more complex than the double buffer.   
  
My render commands are objects that are allocated out of pools that the game thread owns, and must be returned to those pools when they have been drawn. In the double buffer system, the queue that is returned from the render thread contains commands that can be safely returned to their pools, but in the shared queue system there's no obvious way for the game thread to know how much has been drawn. I suppose there could be two shared queues, one in each direction, but that would still be a lot more complicated than what I have now. Right now almost no code outside of the buffer swap system knows about other threads; the pool objects and the objects they contain are not thread safe and, as it stands, don't need to be.   
  
Is my solution the best for Android apps? I don't know. It seems to work pretty well and it is uncomplicated, which are two points in its favor. Still, I'd like to give this shared queue idea a shot at some point; my gut tells me that it will be slightly faster than the double buffer (less blocking in the average case) but a lot more complex, which might make it not worth the effort. Programmer guts are, however, extremely unreliable, so I will probably give this method a shot after Replica Island ships.