Jason Ekstrand

Why Wayland on Android is a hard problem

This is a question several people have asked me and that I have, aparently, become something of an expert on. There are attempts (moste notablly, the one built into libhybris) to solve this problem. However, it's almost provably impossible to actually get it correct because of differences in the way Wayland and Android handle EGL and graphics contexts. Before I can really explain the rub, we need a little background.

Wayland EGL

First, let's begin with how it works when a client gets a GL context on a Wayland compositor. From the client's perspective it's fairly simple. It uses the eglGetDisplay() function in EGL and gives EGL it's wl_display pointer casted to a NativeDisplayType pointer. Then, when it wants to create a window, it calls wl_egl_window_create() with the surface that it wants to use along with the size to get a wl_egl_window pointer. It then casts the wl_egl_window pointer to a NativeWindowType pointer and passes it to eglCreateWindowSurface(). Destroying the wl_egl_window as well as changing its size are done through the functions in wayland-egl.h. Beyond that, the client can use regular EGL functions to handle the rendering context just like it would any other EGLSurface and can use Wayland protocol calls to manage input and any other Wayland things.

The server's perspective on the Wayland EGL surfaces is a little more complicated but it's still not bad. The server's end of the EGL surface is handled through the EGL_WL_bind_wayland_display extension. On startup, server calls eglBindWaylandDisplayWL() annd gives EGL its wl_display pointer. The EGL implementation sets up and aditional global or two and adds whatever hooks it needs at this point. When the server recieves a wl_buffer object from the client, it can use eglQueryWaylandBufferWL() to get the width, height, and color format of the buffer. To bind the buffer as a texture, it uses eglCreateImageKHR() function with a target of EGL_WAYLAND_BUFFER_WL and the wl_buffer resource recieved from the client casted to an EGLClientBuffer. The compositor can then bind the EGLImageKHR as a texture using glEGLImageTargetTexture2DOES(). It looks kind of complicated but it's really pretty simple: Get a wl_buffer, create an EGLImageKHR, bind as a texture.

But what does all this look like from the driver's perspective? That's where it starts to get complicated...

Inside of eglBindWaylandDisplayWL(), the driver adds at least one global to the wl_display that provides some way to create wl_buffer object and associate it with a physical GPU buffer. Inside the client implementation, it gets GPU buffers, allows the client to render to them, and then hands the buffer to the server with wl_surface.commit(). The general procedure for the eglSwapBuffers() call is the following:

1. Get the wl_buffer object corresponding to the buffer the client just finished rendering to.
2. Call wl_surface.attach() with the given buffer.
3. Call wl_surface.damage() to damage the entire buffer. (It's best if it uses wl_surface.damage(INT32_MIN, INT32_MIN, UINT32_MAX, UINT32_MAX).)
4. Call wl_surface.commit() to apply the client's changes

The reason why the EGL implementation calls wl_surface.commit() instead of leaving that to the client is that it allows the client to run in a simple rendering loop with no actual Wayland calls involved. Everything absolutely required for rendering is hidden inside of eglSwapBuffers().

So far, not so bad. Where things get hairy is in synchronization. The eglSwapBuffers() is required (as per spec) to perform an implicit glFlus() operation to guarantee that everything the client rendered prior to calling eglSwapBuffers() ends up on screen. From the server's perspective, it gets a wl_buffer from the client and assumes that it is free to bind it as a texture and render with it immediately. *The Wayland protocol says nothing about buffer and rendering synchronization*. This sounds terrible, but it works very well in reality because it allows the driver to use whatever synchronization primitives it wants without restricting to a Wayland-specific sync object that it now has to implement. In order to properly handle synchronization, EGL implementation provides the following two guarantees to Wayland clients/servers:

1. eglSwapBuffers (or eglSwapBuffersWithDamageEXT) must call wl_display.attach, wl_display.damage, and wl_display.commit before it returns. This is so that the client can synchronize various bits of surface state with the wl_display.commit request. It doesn't matter whether or not it happens in another thread, just that it happens by the time eglSwapBuffers reutnrs.

2. As soon as a compositor gets a wl_surface.attach, it is free to pass that buffer resource into eglCreateImageKHR with type EGL_WAYLAND_BUFFER_WL, convert it to a texture, and start drawing with it immediately.

Exactly how these two constraints are met is up to the EGL implementation. On some particular graphics stacks, glFinish() is called inside of eglSwapBuffers() to ensure that the graphics card has finished with the buffer before passing it to the compositor. However, this causes the GPU to stall and decreases performance. Most drivers solve this by having sort of a fence associated with each buffer. Instead of eglSwapBuffers() waiting for the buffer to be finished before calling wl_surface.attach/damage/commit, they set up a sync fence, call wl_surface.attach/damage/commit, and returns immediately. Then they use the fence to serialize rendering commands somewhere further down the stream. This can mean that eglCreateImageKHR() blocks waiting for the buffer to be finished or that you pass the fence further down the line and block in glBindTexture() or even just use it to serialize commands to the GPU. How that fence is created, passed around, and waited upon is an internal detail of the EGL implementation and is outside the scope of the core Wayland protocol.

A final note concerning the wl_surface.frame event. This event has nothing to do with your buffers or with something happening on the GPU as such. It exists for the sole purpose of telling the client that the compositor has started to render using the currently attached buffer and that it can go ahead and start rendering for it's next frame. In a lot of cases, drivers don't care bout this event, but it can be useful for implementing eglSwapInterval(1) or similar.

Android EGL

Ok, now that we've given a brief intro to Wayland EGL, let's talk about Android. Android implements its graphics stack by using what is sometimes called meta-EGL: an EGL implementation that wraps another EGL implementation. This allows the Android OS to track a few things and provide additional extensions on top of what the core driver EGL implementation provides. Most of what we care about, however, is actually inside of the driver EGL implementation.

For the client, Android provides a data structure called ANativeWindow that is implemented by the display server. (For pure android, this is SurfaceFlinger.) The display server uses some magic to pass this structure to the client and the client passes it to eglCreateWindowSurface(). The EGL implementation then uses the function pointers inside of the ANativeWindow structure to get buffers from the queue, render to them, and then pass them back to the display server for display. The ANativeWindow structure also contains a mechanism for syncing rendering using file descriptor based fences. The function pointers we care about are as follows:

- dequeueBuffer(): This is used by the driver to get a buffer from the window system's queue of buffers for this window. The implementer of the ANativeWindow interface is responsible for allocating some buffers via gralloc and handing them out whenever the driver calls dequeueBuffer().
- queueBuffer(): This is called by the driver to hand the buffer (previously retrieved from dequeueBuffer()) back to the window system for compositing. The only real requirements on this function are that the buffer must come from dequeueBuffer() on this ANativeWindow and that the buffers must be enqueued in the same order they are dequed.
- cancelBuffer(): This is called by the driver to tell it that it does not intend to use a buffer it previously dequeued.
- perform(): This function is used to perform a variety of operations such as setting the size of the surface or to change the color format. Calls to perform come from the client (not the driver) via a set of inline wrapper functions such as native_window_set_buffers_diemnsions().

From the server's perspective, it has to find some way to pass these ANativeWindow structures to the client over IPC and implement the function pointers on the client-side. How this happens doesn't matter. Everything in the structures is basic binary data and a few file descriptors. The display server is also responsible for allocating buffers and handing them to the EGL implementation when needed. Buffers are represented by

ANativeBuffer which is just a collection of integers and file descriptors. Buffers are allocated using Android's "gralloc" module which is outside the scope of this post. Whether the display server allocates the buffers in the server process and passes them to the client or allocates them in the client process and passes them to the server doesn't matter. The point is that the EGL implementation has access to a queue of buffers via ANativeWindow and that they are allocated using gralloc.

It's also worth noting that Android has very few restrictions on how these functions can be called. Already mentioned is that the buffers have to be enqueued or canceled in the same order as they are dequeued. However, Android has no requirements on synchronizing with input or window geometry changes. In fact, in most Android applications, the entire UI is destroyed and recreated from the ground up every time the geometry changes due the device being rotated or the app going full-screen. This is very different from Wayland's "every frame is perfect" mantra.

Wayland + Android

Ok, so what happens when we take these two and try to make Wayland work in an Android world. As it turns out, this works surprisingly well. The libhybris project has an implementation that, at least on some hardware, works well enough that Jolla (among others) is shipping devices that run on it. While it works OK on some hardware, it is far from perfect in the general case. Really, this isn't the libhybris developers' fault, but a core collision in the way that Wayland and Android think.

The way libhybris works (and any other implementation seeking to do the same thing) is by writing another meta-EGL layer. The meta-EGL provides an EGL implementation that supports the EGL_WL_bind_wayland_display extension and implements it using an ANativeWindow under the hood. The ANativeWindow structure is usually implemented more-or-less as follows:

- dequeueBuffer(): If enough surfaces already exist, grab one from the queue and hands it back to the driver. If it needs a new surface, it sends communicates with the server-side of the driver over the Wayland protocol to allocate a new buffer and get a wl_buffer object. Whether the client gets the buffer from gralloc and hands it to the server or the server allocates the buffer and hands it to the client doesn't matter. The libhybris library currently does the former, but the later could also be done and I've thought about doing it.

- queueBuffer(): In general, this calls attach (with the buffer given to it by the driver), damage and commit. This is (sort-of) called by the driver's SwapBuffers() function.

- cancelBuffer(): Yeah, this should be obvious.

- perform(): Not used. These operations are done either through the Wayland protocol directly or through functions that act on the wl_egl_window structure.

Now scroll back up to the Wayland section and look at the two requirements that every Wayland-enabled EGL implementation is supposed to provide. The second one is fairly easy since Android already provides a fence mechanism. The first, as it turns out, is almost impossible to actually satisfy.

The real problem comes from the fact that Android provides no real guarantees as to what a driver has to do inside of eglSwapBuffers. At some point, the driver will put the buffer the client was rendering to back into the queue with ANativeWindow.queueBuffer(), but it may do that at some point in the future or from another thread. To make matters worse, it may not submit a buffer at all in some cases. By setting the EGL_SWAP_BEHAVIOR attribute of the EGLSurface to EGL_BUFFER_PRESERVED, or by using the EGL_EXT_swap_buffers_with_damage extension, it is perfectly acceptable for the client to do no rendering at all and then call eglSwapBuffers(). In that case, it is expected that the same image will appear on screen as the previous one. However, the Wayland guarantee about calling wl_surface.attach, wl_surface.damage, and wl_surface.commit on every eglSwapBuffers() call becomes very difficult to satisfy.

I've made a couple of attempts inside of libhybris to solve this. One was to use the EGL_KHR_fence_sync extension to wait for the driver EGL implementation to actually submit the buffer to the ANativeWindow before doing the commit. If the driver EGL implementation doesn't give us a buffer by the time the sync returns, we go ahead and commit anyway. Unfortunately, this relies on the driver EGL implementation implementing EGL_KHR_fence_sync in the way that we expect, namely not returning from eglClientWaitSyncKHR() until the buffer has actually been pushed. Not all Android-compatible driver EGL implementations do this. Giulio Camuffo has tried to make libhybris' implementation more intelligent tracking what buffers the driver EGL implementation has pulled from the queue and trying to guess which buffer it will put in next. Unfortunately, none of these solutions actually work on all drivers.

That's about the long and the short of it. It's kind of an unfortunate situation. If you've got any ideas about how to make it better, I'd love to hear about them. Or even better yet, just start hacking on libhybris and see if you can improve it. Be warned, however, that what works on one driver may break another completely. Welcome to the world of hardware. I hope you had a good read!