Lightweight GPU Sync Points

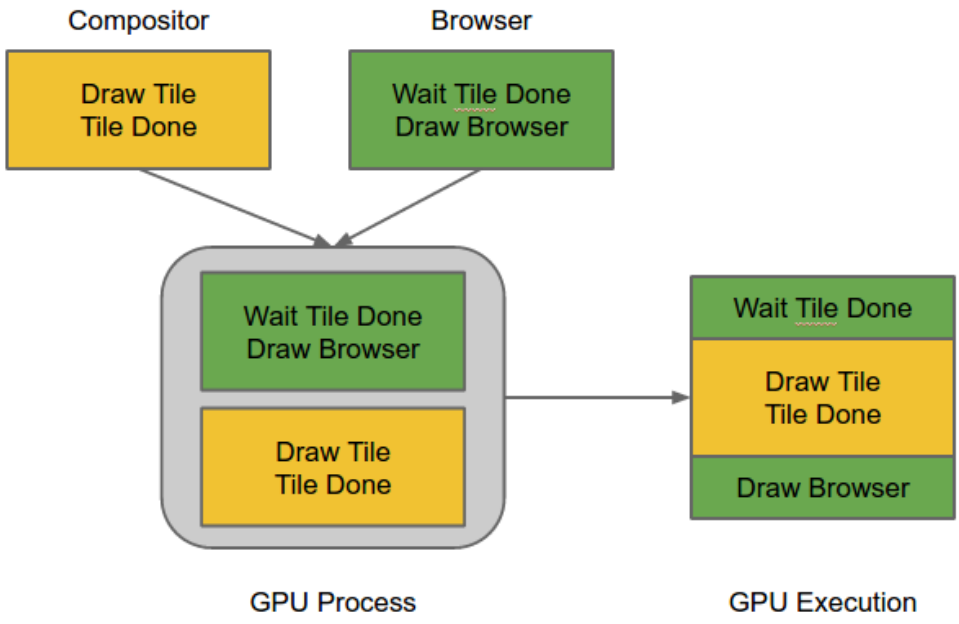
Contents

Objective

The ability to synchronize GPU commands between renderers, compositors, and the browser in a way that is both lightweight and guaranteed to finish.

Background

As different GPU clients are sending commands to the GPU process to render things asynchronously, there are bound to be times when we need to synchronize the draw calls. A simple example is when a compositor is rasterizing a tile, ideally you would want to be able to have the browser renderer to send the draw call for the rasterized tile simultaneously even though it depends on the compositor. The GPU could get both draw calls in any order, and draw the things in the right order. If the GPU received the browser draws first, it would need to know to wait and *yield* to the compositor draw calls. In this case it would look something like this:



Terminology

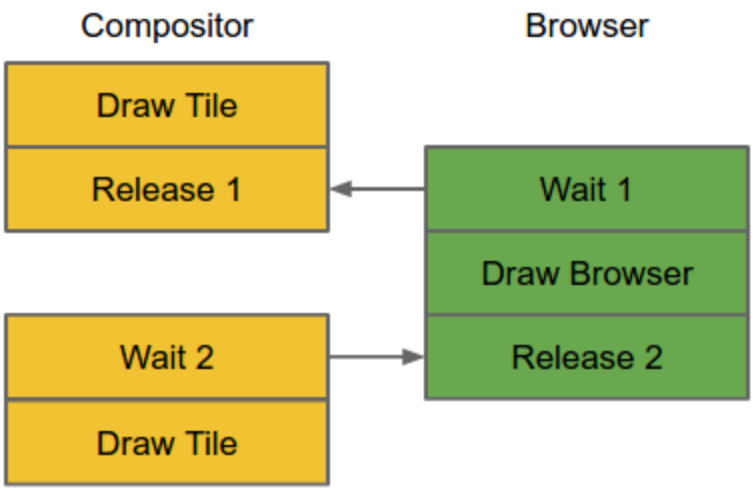
Sync Point: Object which we use to synchronize draw calls. This is what this design document is about.

Sync Point Wait: Terminology used for a command to yield and wait until a sync point has been passed. In the above example the “Wait Tile Done” yield would be the **Sync Point Wait**.

Sync Point Release: Terminology used to indicate that a sync point has been passed. In the above example, the “Tile Done” notification would be the **Sync Point Release**.

Issues

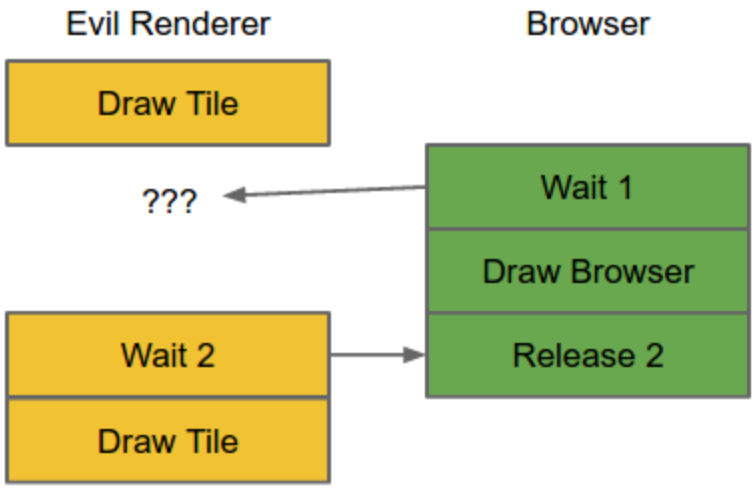
Under normal circumstances the usual pattern would look something like this:



As long as each wait is waiting on a release that will eventually happen, no deadlock can occur. However, because clients are not trustworthy, there are a few scenarios we must be aware of. Because draw calls are able to wait on one another, care must be taken to be sure that clients cannot cause the GPU execution to deadlock by waiting indefinitely. Deadlocks can occur either through accidental errors or malicious intent.

Invalid Releases

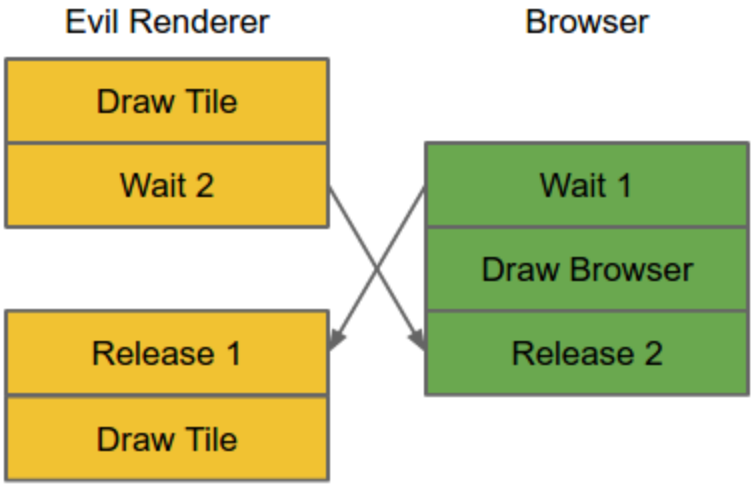
The most basic error that could happen is an invalid release. If there are any releases which do not exist, a wait on an invalid release could cause the browser to wait indefinitely. An indefinite wait could look something like this:



For this reason releases must be validated somehow before a wait is executed.

Dependent Releases

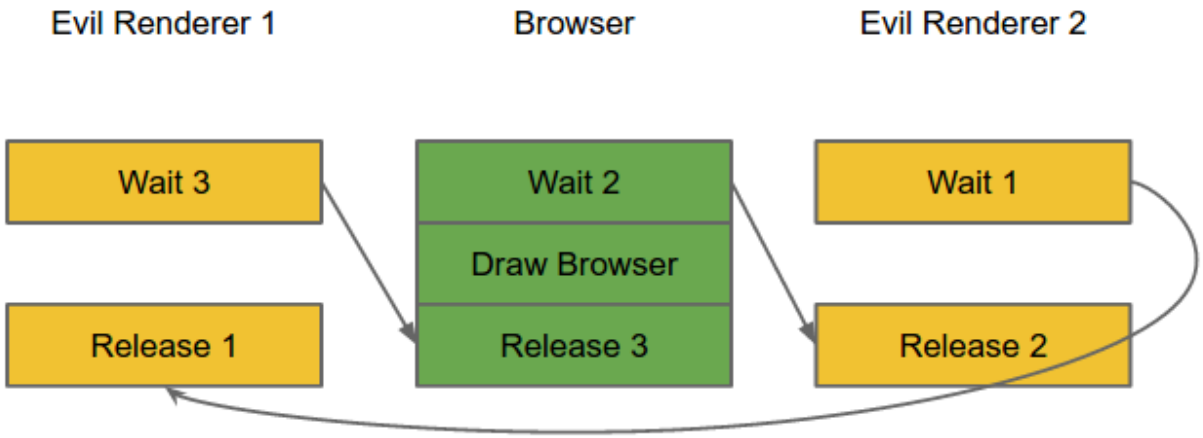
Another way a malicious client could cause the browser to wait indefinitely is if a malicious client caused a valid release to wait behind it's own wait. That way, both clients would be waiting on each other and none would ever continue executing:



For this reason we must somehow validate that wait's must only wait on releases that are not behind a wait. One way to guarantee this is to guarantee that corresponding releases always occur before a wait.

Complex Cycles

Another more subtle issue is the fact that we cannot just coordinate validations between 2 clients. It may be tempting to just have the browser keep track of who it is waiting on and somehow correlate waits/releases on the client side. Another thing that could happen is that 2 or more malicious clients could work together and cause the browser to sit in the middle of a cycle without knowing it. Such a configuration would look something like this:

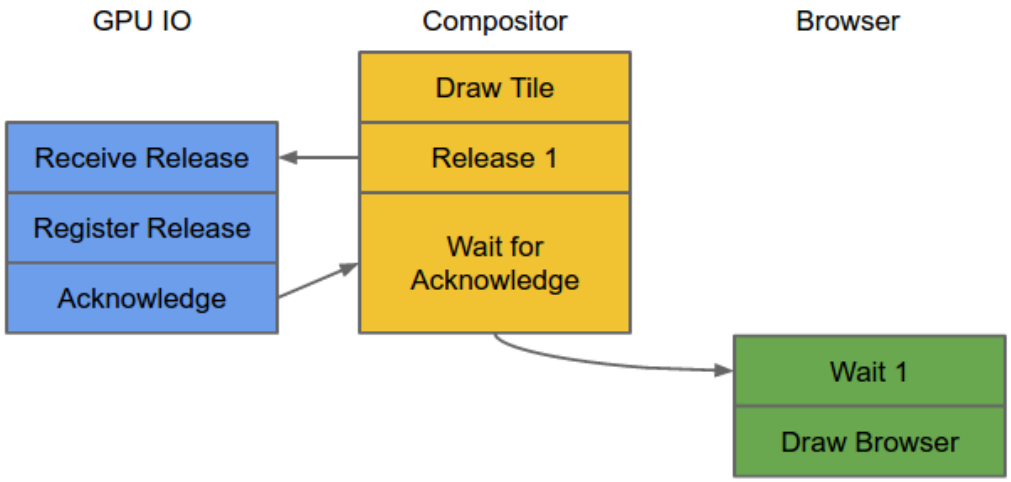


Here from the browser's point of view nothing looks wrong, but 2 clients can work together to cause a deadlock.

Current Implementation

Because all of these commands on each client ultimately get flushed to the GPU process and are executed linearly, one easy way to solve this issue is to simply guarantee that releases are flushed and exist on the GPU process before anything attempts to wait on it. When we want to wait on a sync point, it is required that the sync point was released through an IPC to the GPU process. The GPU process receives the release and notifies the client that the client may now allow that sync point to be waited upon.

This round-trip guarantees that no deadlocks could ever occur, but it is costly (each release currently costs 0.33ms on a z620). It is currently implemented as a synchronous IPC that the client issues to the GPU IO process, so it waits for the response to be sure everything happened properly. This stalls the client until the GPU process has acknowledged it has properly handled all the sync points. A simplified version look at the events look something like this:



In a case such as a compositor rendering a group of tiles, each tile would have it's own sync point. Each release then has a synchronous IPC call to verify the sync point was inserted properly. The overhead of this quickly adds up.

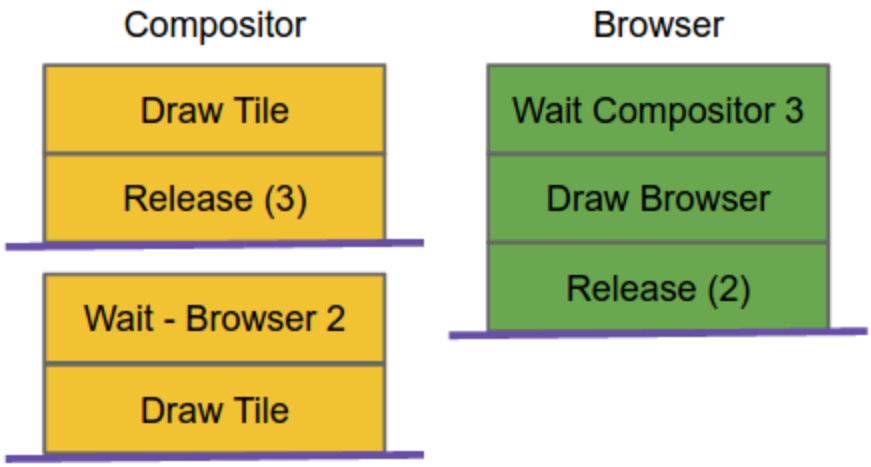
New Proposal

The current implementation is not only costly because of the round trip for every release, but it also makes the client have dependencies on the GPU IO thread which is most likely processing other IPC calls. Ideally we would be able to simply produce commands independently of the GPU process and have the GPU process processes all the commands and somehow figure out when releases are invalid on the fly (while solving the issues brought up above from malicious/accidental clients). If we could minimize the synchronous IPCs to the GPU IO process for many sync point releases to just one (because we no longer have to flush and have a synchronous IPC for every single release), it would be a huge win in terms of overall overhead. For example, if we were to

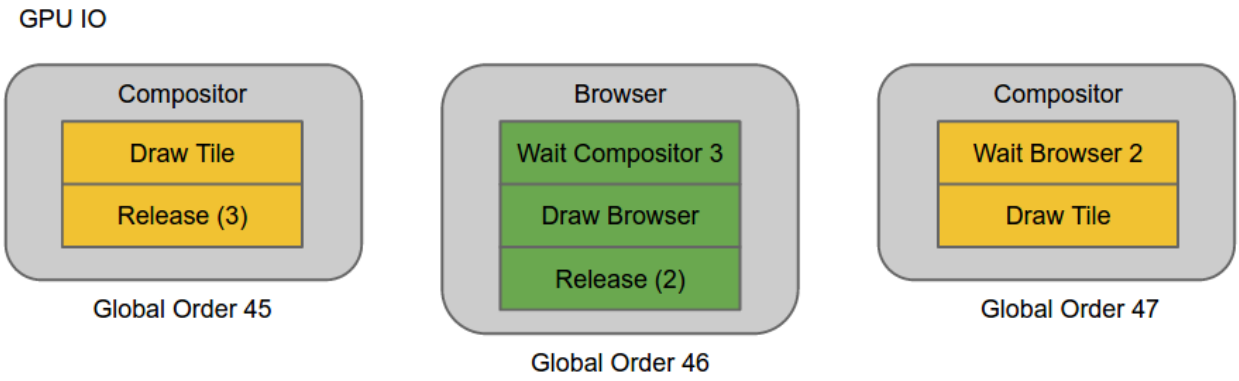
render 10 tiles and each synchronous IPC took 0.3 ms, that would add an overhead of at least 3 ms! If we could enqueue the work for all 10 tiles and flush all the releases at once, we would amortize this cost to just be 0.3ms.

The new proposal's sync point release will simply be an incrementing number, it can be seen as the number of releases for a particular route. The new method relies on the fact that the GPU IO thread multiplexes all of the requests in order of flushes within the same process. For cross process synchronization we would need to add a NOP round trip to the GPU IO thread to ensure the commands are ordered on the GPU IO thread properly. Even the NOP round trip is much better than a the current implementation where we have a round trip every release though, we only need to do it once when synchronizing between processes. If we require that the command buffer with the release is flushed before the corresponding command buffer with the wait, the GPU IO thread can label each command buffer flush with a "global order" which allows the GPU process to perform validation. A valid wait/release pair is simply a release which has a lower global order than the corresponding wait.

A wait can quickly decipher whether it needs to yield to another command buffer or not by checking if the route's highest release count. We can represent a sync object by simply having it be a combination of the Unique Route ID and the Release Count it is waiting on. For example in the original example it would look something like this, the purple lines here represent flush points:



Here the only requirement is that the flushes are done in order so that the GPU IO thread may order everything properly. Here is an example of how the global ordering may look:

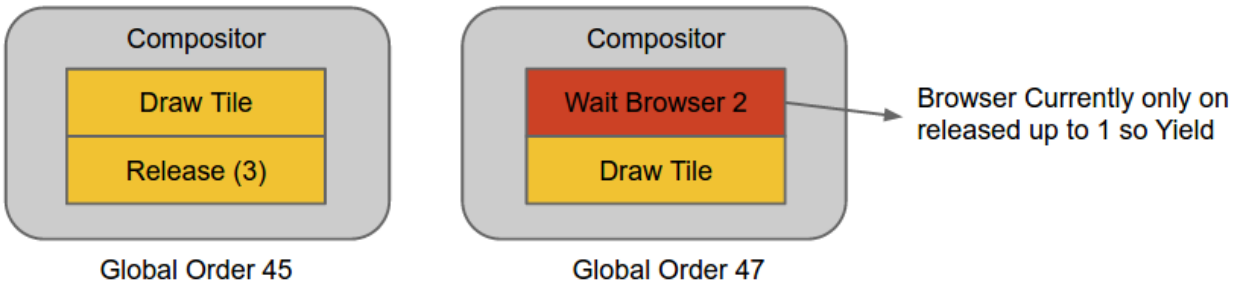


As the commands get multiplexed to the GPU process, it may be processed out of order. However, since we required that command buffers with releases are flushed first, they will have a lower global order assigned from the GPU IO thread.

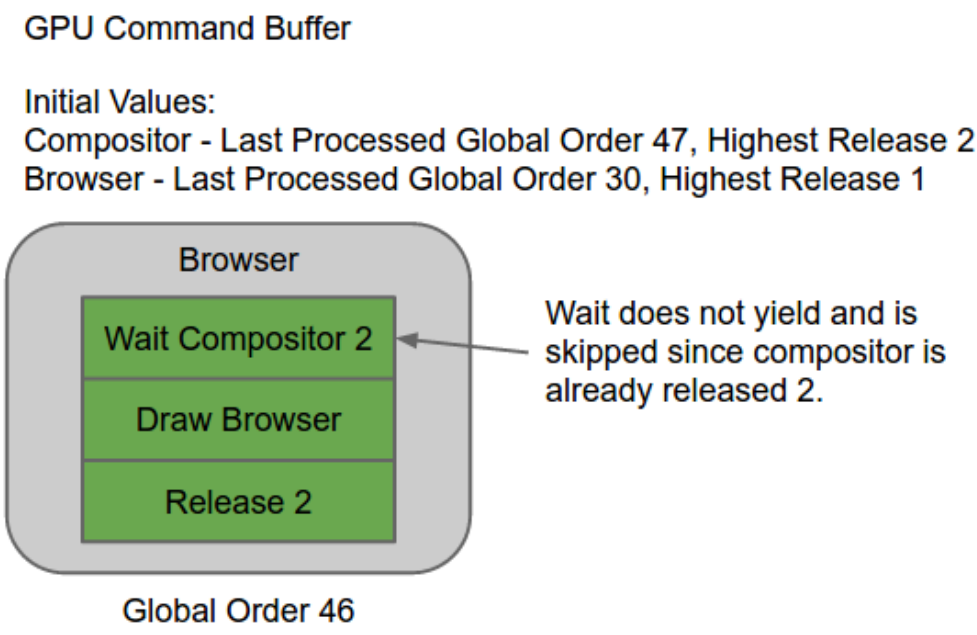
Each wait command (which contains the Route ID and the Release Count to wait on) will check the corresponding Route ID's current global order and release count released. This will be done from within the command buffer command execution itself. By comparing the last executed global order number and the highest released release count, it can instantly figure out whether it should yield or not. Here is an example of a possible processing sequence for the GPU command buffer thread:

GPU Command Buffer

Initial Values:
Compositor - Last Processed Global Order 44, Highest Release 2
Browser - Last Processed Global Order 30, Highest Release 1



If we happen to have processed the 2 compositor command buffers first, in this case it will yield after checking that the browser thread has not released 2 yet. It will then continue to process and eventually process the browser channel:



Here the browser command buffer will execute completely since it did not need to wait on anything, and once the browser has released 2 the compositor's wait will know that it can run and continue.

It is easy to see now why invalid releases found in the issues section are automatically handled now. Since wait's can compare their current global order to the corresponding route's global order, it can validate whether the wait is valid or not. If the release has a higher global order than the wait we can safely ignore it as it is invalid.

As a note, this scheme works for synchronizing any IPC with a routing ID even if it is not part of a command buffer. The IPC must be modified to be able to increment its own "release counter" and have a global order ID assigned to each flush. Command buffer commands can then wait on these route ID's and treat them as any other sync point release.