Gpu Off During Scroll (Public)

go/gpu-off-during-scroll-public
ccameron@, 2024-08-28

Gpu Off During Scroll is a project to enable Delegated Compositing on Android, Windows, and Wayland. When possible, user scrolls will be done by moving around already-rendered buffers of pixels using the fixed-function display controller hardware (aka DPU).

The primary benefits are (1) reduced power consumption and (2) reduced jank, caused by reducing overall CPU and GPU work of the scroll-input-to-pixels-on-screen pipeline, and by reducing contention on the GPU. The secondary benefit is that this simplifies the software stack substantially.

High level design: Early overlay and render pass decisions

Chrome currently does not set up the most OSes or hardware to succeed at delegated composting. This project aims to fix this problem in Chrome and **meet the existing OSes and hardware where they are**. Do not try to solve software problems in silicon!

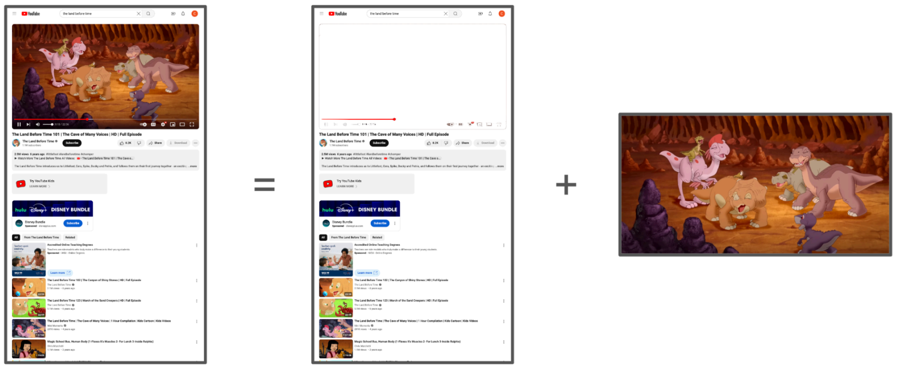
At present, Chrome mostly thinks of itself as having just one Surface (the “back buffer”). Using various overlay strategies, Chrome will sometimes “pluck out” individual quads (e.g, a video or canvas) to “promote” to an “overlay” (a Surface). This is done statelessley towards the end of the rendering pipeline, reflecting that this capability is a bolted-on afterthought.

In this project we move the overlay logic much earlier in the pipeline. The overlay decisions are made to be stateful (removing the enormous complexity of trying to make them be temporally coherent). The overlay decisions are also integrated into render pass allocation (allowing multiple quads, e.g, content that scrolls together, to be “fused” into a single overlay).

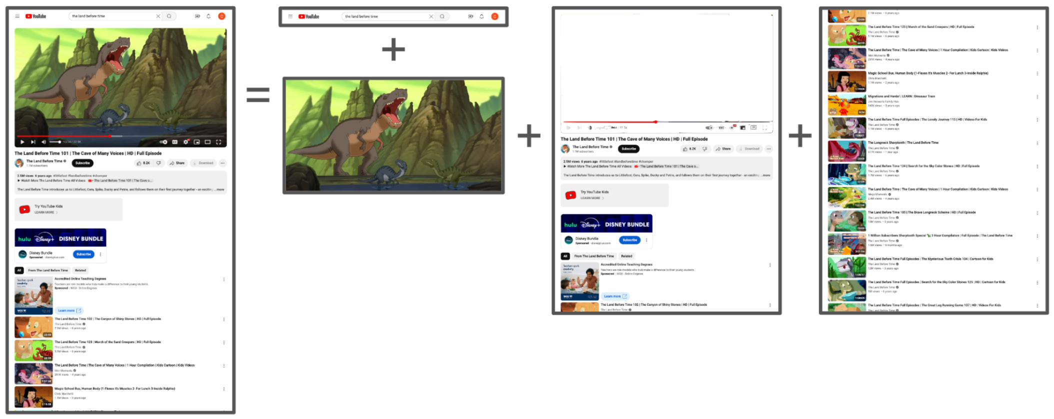
The goal is for this overlay logic to produce results that are *likely* to hit hardware composition (and not have to fall back to GPU composition). This will generally mean targeting 3 or 4 total surfaces, and having the main scrollable area be a surface treadmill with 2 surfaces being visible at any time.

Example: Scrolling YouTube

Consider the following example of watching a YouTube video. Our current overlay strategies “pluck out” the video, punch a transparent hole in the backbuffer, and place the video behind the backbuffer, as shown below.



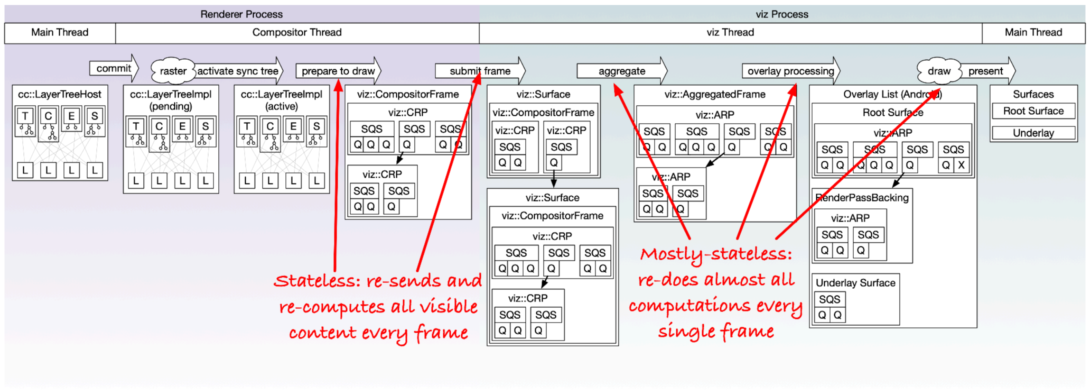
This is reasonable, but it will require that we re-draw the whole backbuffer every time the user scrolls. A better division into surfaces would be the one shown below. In such a scenario, a user scroll would not require any GPU work.



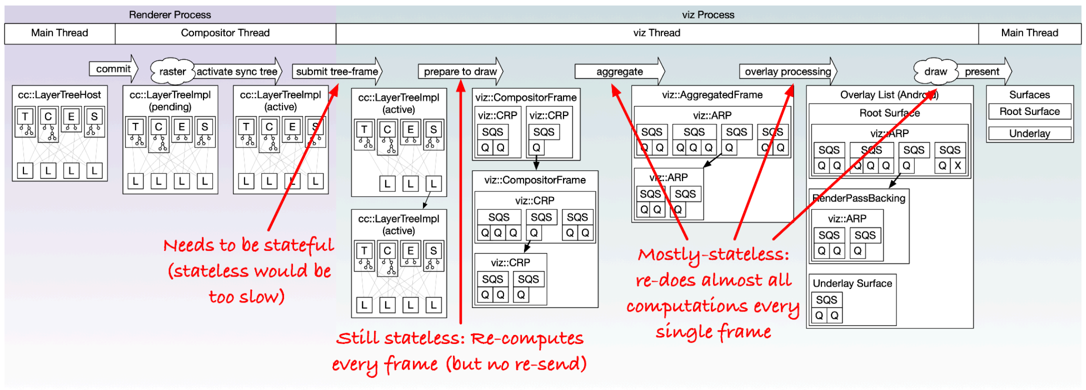
Chrome currently cannot produce this sort of division, or anything like it. It is not possible to guarantee that this representation would avoid GPU composition, but this at least gives SurfaceFlinger and the underlying Hardware Composer a fighting chance.

Lower level details (And by that I mean still very high level)

The current boundary between renderers and viz is the viz::CompositorFrame, which is populated by the function cc::LayerTreeHostImpl::PrepareToDraw. This boundary is stateless. Every frame sends a full viz::CompositorFrame. The viz::CompositorFrames from the tree-like structure of surfaces are smashed together into a viz::AggregatedFrame, also in a mostly-stateless way.



This project fits in after Phase Two of go/phased-jellymander. After that phase, the boundary between renderers and viz will be a cc::LayerTreeImpl, and the call to PrepareToDraw will be in the viz process. Importantly, the cc::LayerTreeImpls are already fully rasterized (tile management and rasterization happens in the renderer process). This pipeline would look more like this:

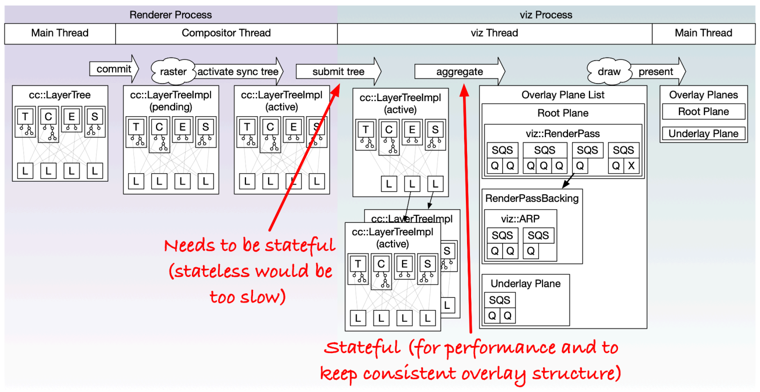


The sequence of changes to be made would be as follows:

- Push the current code in PrepareToDraw that populates the viz::CompositorFrame to instead be done the property tree forest in SurfaceAggregator::Aggregate.
    - There is then no longer a creation of CompositorFrames and aggregation into AggregatedFrame, but instead a direct generation of AggregatedFrame.
- Create persistent tree for render passes
    - The current implementation of PrepareToDraw creates a bunch of ephemeral render passes

    - Change this to be a persistent tree structure that that has updates pushed to it
    - This is approximately the RenderPass/Overlay tree from [this figure]
- Make the this PrepareToDraw/Aggregate function responsible for overlay decisions
    - Add a persistent overlay tree/list structure structure that lines up with the above render pass structure
    - This would wrap the OS-specific compositor (SurfaceControl, DirectComp, CoreAnimation, or Wayland)

At this point the the pipeline looks as follows



- The beginning of the end
    - With all of that in place, we can be strategic about chopping what is currently the root render pass into something that allows scrolling with the GPU turned off
    - If, given all of this, we still struggle to hit DPU composition, we may want to *at that point* ask for Android APIs to help us.

Additional benefits

The above motivation is mostly about delegating scrolling to the DPU, but this architecture makes implementing many other features much easier. Here is a short list.

Overlay planes based content (e.g, HDR separation)

The pixel format for the backbuffer is decided based on the contents of the RenderPasses that contribute to it. So if you have an HDR video on-screen, then the whole window gets promoted to a pixel format that supports HDR (float16).

If that HDR video is going to get promoted to an overlay, then there actually was no HDR content in your backbuffer, and so you promoted the whole thing to a more expensive pixel format for no good reason.

In the Gpu Off During Scroll architecture, because we make render pass backing and overlay decisions at the same place, we trivially avoid this behavior.

Overlay planes based source (e.g, top-Chrome separation)

There is almost never a good reason for the top-chrome of a window and the web contents of the window to be stored in the same backbuffer. They are updated independently, they have different pixel format requirements, and they aren't even physically overlapping.

Separating the top-chrome off into a separate buffer is trivial in the Gpu Off During Scroll architecture. We can just create a "put this surface into its own overlay" tag, and set that tag on platforms where this is desired.

RawDraw

Without RawDraw (as things are), we rasterize tiles' draw commands to textures-backed quads, which we then composite together to the backbuffer.

With RawDraw, we save the tiles' draw commands and rasterize them directly into the backbuffer. This saves memory (no tile memory) and can improve performance (we just draw once, not twice).

You can think of this as the "solid color draw quad" optimization, just much more generalized.

This effort has stalled out in the past because of bad interactions with overlays, and also because it can hit pathologically bad performance. The pathological performance can happen because we often have to re-rasterize the whole backbuffer. If we chop the backbuffer into smaller parts, this becomes less likely. If we also have a much more flexible RenderPass and overlay scheme, we can identify slow-to-raster content when we draw it (by observing that it took a long time), and force that content into a separate RenderPass in any subsequent frames that would need to draw it.

Note that in this scheme, what used to be a Tile now in effect becomes a RenderPass and, in effect, much of the rasterization logic from the renderer to viz. This transition should be taken very slowly and cautiously (which is an option with Gpu Off During Scroll).