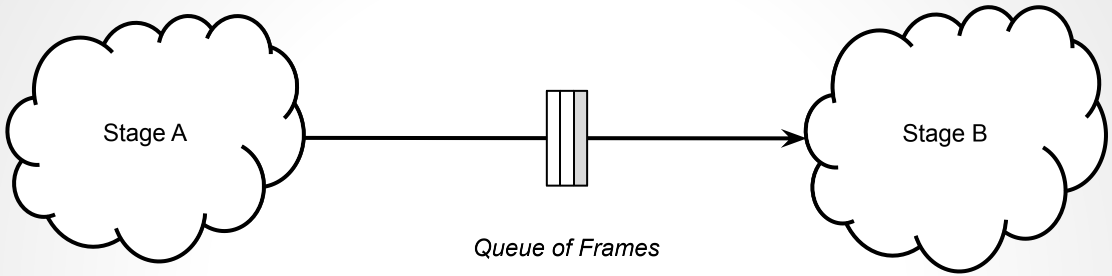
Chrome Scheduling

London Perf Summit
March 17, 2014
@brianderson

Overview

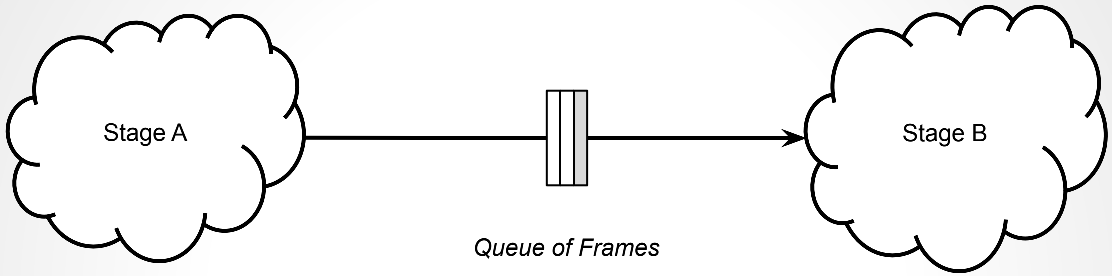
- Scheduling Basics
- Renderer
- Coordinating: Renderer + Input
- Coordinating: Renderers + UI
- Coordinating: Renderer + UI + GPU
- PPAPI Plugins
- Testing Framework
- Next Steps
- Questions

Low vs. High Latency Modes

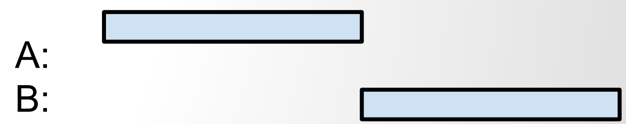


- Stage A is in a **low latency mode** if stage B consumes A's output within the same frame interval.
- Stage A is in a **high latency mode** if stage B consumes A's output in a subsequent frame interval.

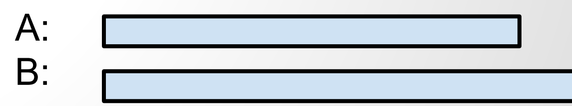
Latency vs. Throughput



- In a **low latency mode,** stage A and B must run serially.

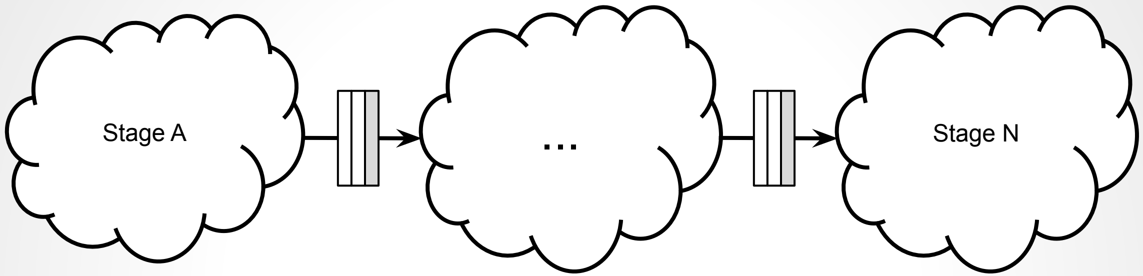


- In a **high latency mode,** stage A and B can run concurrently.



Caveat: Concurrency doesn't necessarily help throughput if we are already CPU bound.

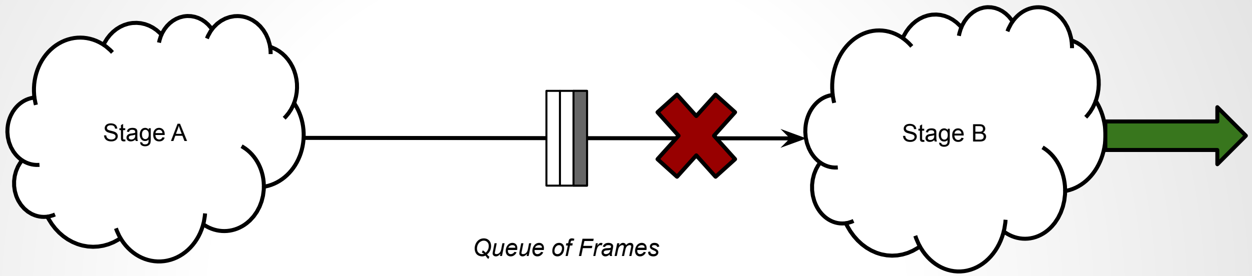
Latency and Memory



- Let H be the number of stages in a high latency mode between A and N.
- If Stage A writes to a framebuffer that is read by Stage N, then there must be at least H+1 copies/versions of the framebuffer.

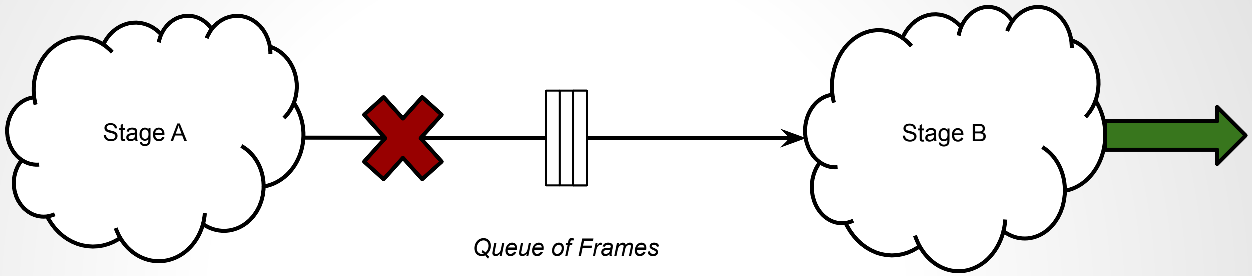
There are ways to reduce the number of buffers needed in some cases, but it gets tricky.

High Latency Mode



- Stage A will enter a high latency mode if B produces a frame without consuming a new frame from A.
- This usually occurs when Stage A takes too long, missing B's **deadline**.
- Examples: CSS animation, fast scrolling, UI updates, multiple Renderers.

Latency Recovery



- **Problem:** Once stage A enters a high latency mode, it will usually stay there.
- **Solution:** If stages A and B can run serially before B’s deadline, then skip a frame of production in stage A.
- **Caveats:** Latency jank!

Latency Jank

Is it possible to have “jank” even if we are hitting 60fps?

Yes!

Examples:

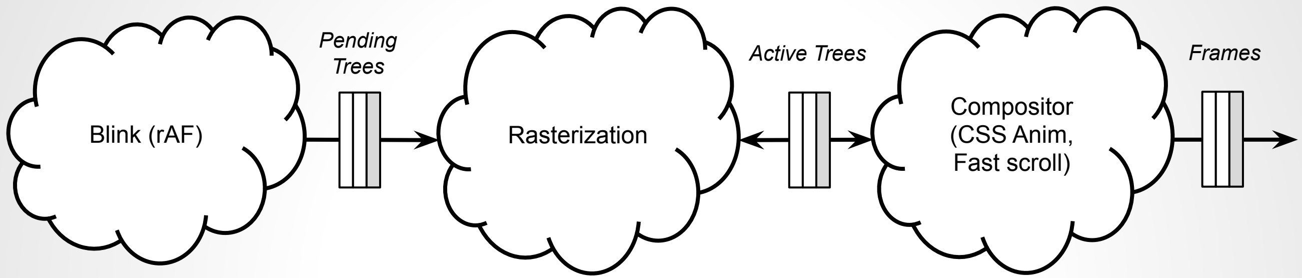
- Transition from high to low latency mode.
- Transition from low to high latency mode.
- Inconsistent animation timestamp intervals.
- Inconsistent intervals of user input.

Balancing It All

Let’s look at how Chrome’s scheduling works (and sometimes doesn’t work) to balance:

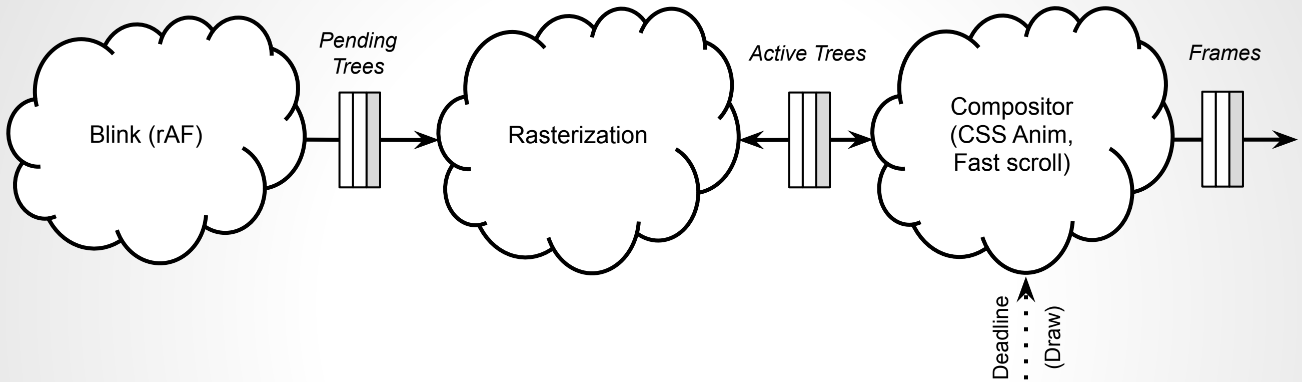
- Latency
- Throughput
- Jank
- Memory

The Renderer's Display Pipeline



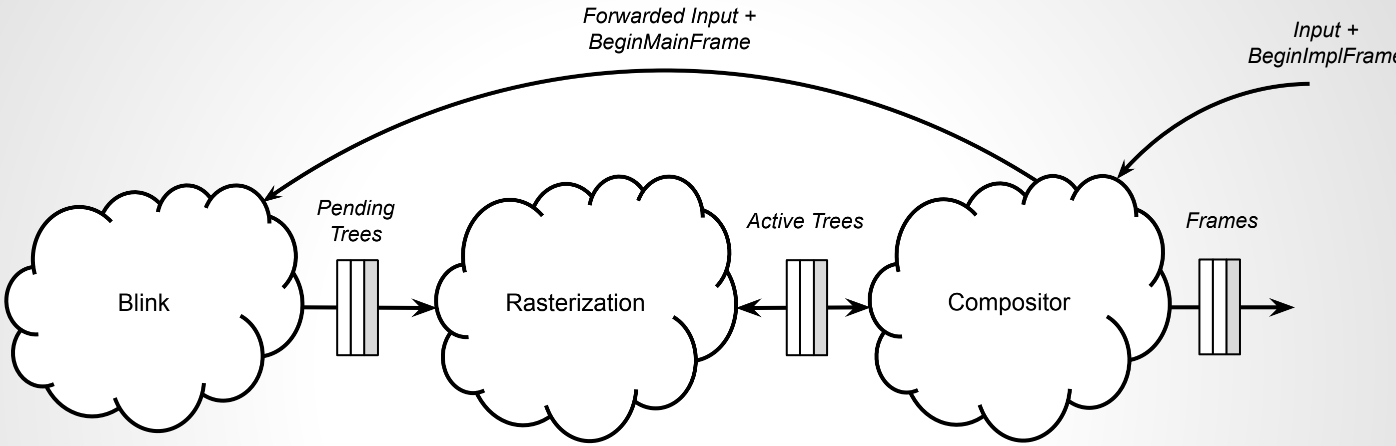
- Limit the size of every queue to 1 frame.
- Blink is forced into a low latency mode relative to Rasterization.
- Blink may run in a high latency mode relative to the Compositor.
- Recently enabled latency recovery between Blink and Compositor. (@dominikg, @skyostil)

The Renderer's Display Pipeline: Deadline



- The Compositor has a **deadline** to decide when to give up on Blink.
  - Is immediate if we are in "prefer smoothness"/accelerated scrolling mode.
  - Is a whole frame away if the Compositor is idle.
  - Is a fractional frame away if there are CSS animations.
- The deadline can be **triggered early** once Rasterization completes or if Blink and Rasterization are idle.

Coordinating: Renderer + Input



- The BeginFrame messages split user input into vsync intervals.
- Ideally, input arrives at the display rate and is followed by a BeginFrame.
- Compositor gets first dibs on input for accelerated scrolling.
- High latency modes introduce input synchronization issues.

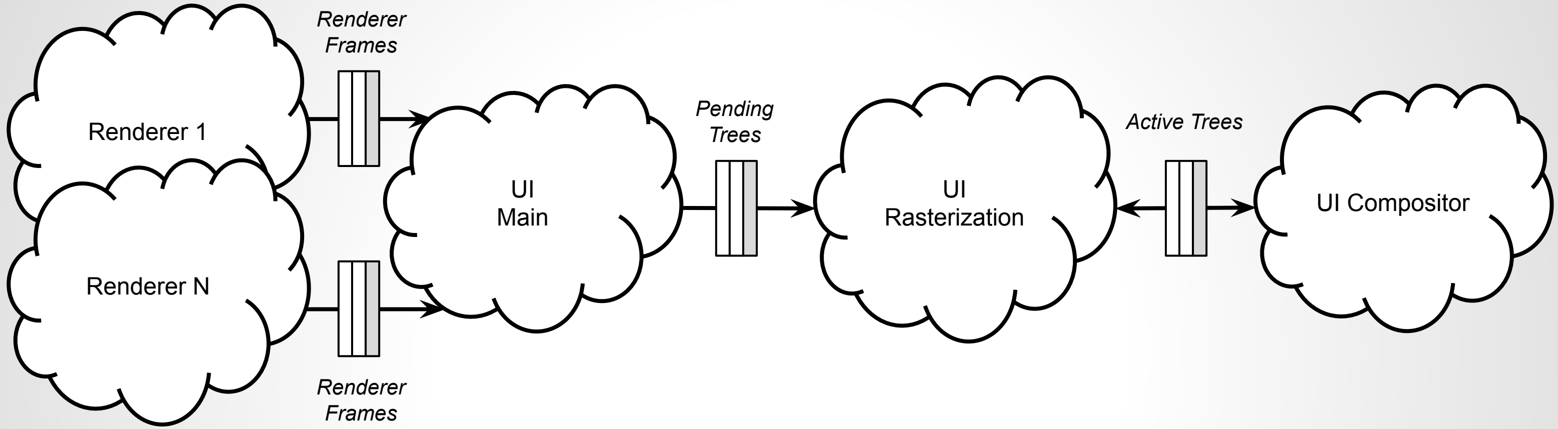
First Frame Latency is Very Important

The latency of all subsequent frames will only be as good as the first frame.

Some tricks we use to reduce first frame latency:

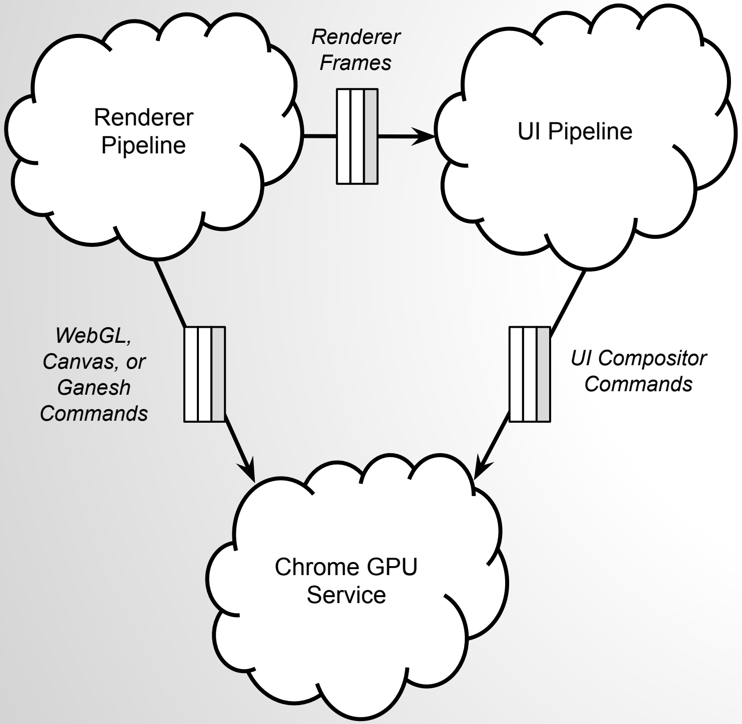
- Synthesize the first BeginFrame message in some cases.
- Proactive SetNeedsBeginFrame hides positive-edge latency.
- Retroactive BeginFrame attempts to catch up on frame production immediately if the deadline hasn't passed.

Coordinating: Renderers + UI



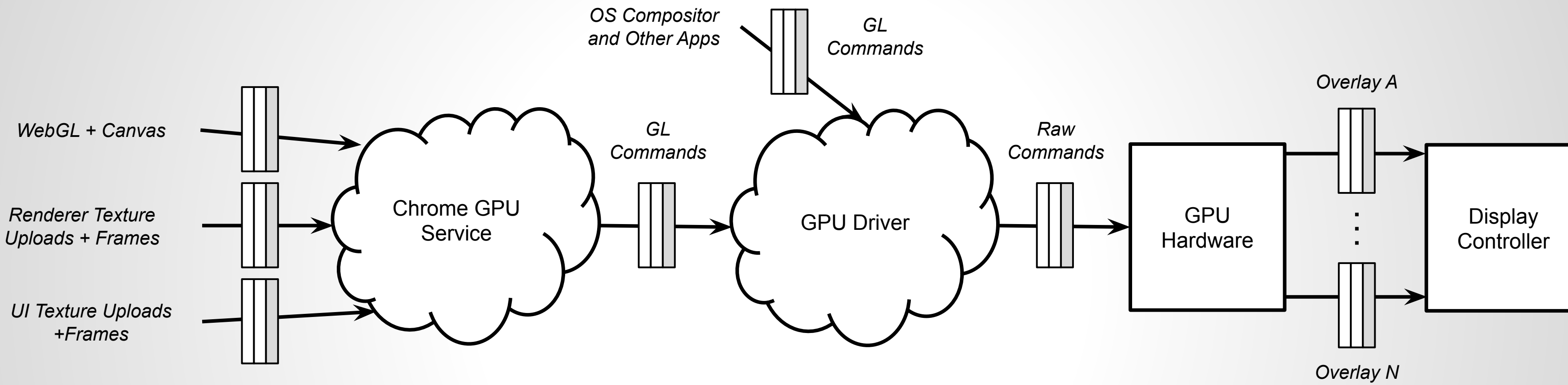
- UI stages operate under (almost) the same latency rules as the Renderer.
- Single Renderer case is handled very well and triggers UI immediately.
- Multiple Renderer case isn't handled well at the moment.
  - Renderers should be able to submit directly to the UI Compositor.
  - Renderer<->UI latency recovery does not yet exist.

Coordinating: Renderer + UI + GPU



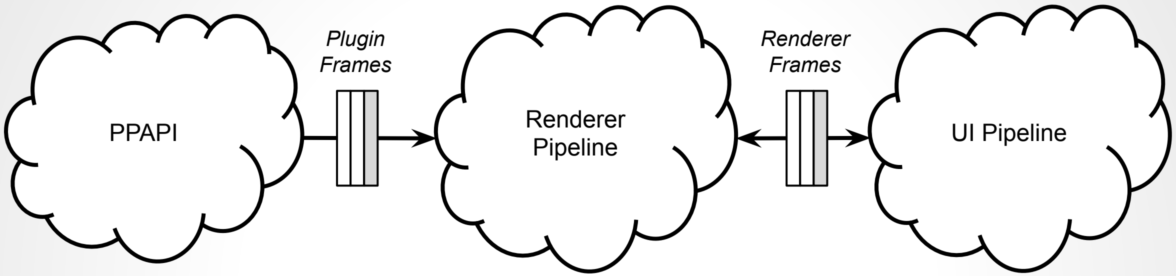
- The GPU is a shared serial resource.
- If the Renderer is in a high latency mode, its GPU commands could delay more urgent UI commands.
  - GPU Service should be smarter.
  - Use **weak future sync points**.
- To reduce number of buffers needed, use **strong future sync points**.
- To reduce GPU Service latency, aggressively stream commands?

Hidden GPU Latency



- We don't monitor feedback regarding latency after the GPU Service.
- On many systems that feedback doesn't even exist!

Pepper Plugin Display Pipeline



Issues:

- BeginFrame + deadline is not yet exposed to PPAPI.
- PPAPI should be able to submit frames **directly** to the UI Compositor like a Renderer does. See proposal for Surfaces.
- PPAPI is also a GPU Service client.

Synthetic Scheduler Tests

Goal: To test various scheduling corner cases in a way that is mostly immune to the variance introduced by code performance improvements.

A combination of features:

- Synthetic delay points
- Synthetic delay point modes
- Synthetic content types

Tracking bug | Design doc

Synthetic Tests: Delay Points

Each test can control a small, but key, set of synthetic delay points:

- **blink.HandleInputEvent** - Expensive JavaScript input event handling.
- **cc.RasterRequiredForActivation** - Long running raster tasks.
- **cc.BeginMainFrame** - Complex main thread picture recording, layout or RAF.
- **gpu.AsyncTexImage** - Slow texture uploads.
- **gpu.SwapBuffers** - GPU-bound rendering.

Synthetic Tests: Delay Point Configs

Each synthetic delay point can be configured in one of the following ways:

- **No-op** (default)
- **Delay to T ms** – Uses a busy loop to ensure that the execution of the scope where the delay point is defined takes at least T ms.
- **One shot delay to T ms** – Same as above but resets the delay to zero after one iteration.
- **Alternating between zero and T ms** – Every second iteration uses the configured delay, every other one adds no delay.

Synthetic Tests: Content

The following are content use cases the scheduler must address well.

- **simple_text_page.html** - Represents scrolling a simple page with no touch handlers.
- **touch_handler_scrolling.html** - Represents the case where a JavaScript handler intercepts all touch events and implements custom scrolling interaction by manipulating the DOM.
- **raf.html** - A constantly running requestAnimationFrame handler that modifies the DOM and the page is also being scrolled.
- **raf_touch_animation.html** - This has three frame producers: touch handler DOM modifications, requestAnimationFrame DOM modifications and a CSS animation.

Synthetic Tests: Putting It All Together

Recap: Delay points, Delay point modes, Content types

Running all possible combinations of the options above is not feasible, so we select a handful of use cases that we care about and/or provide good coverage. See design doc for details.

We are already tracking throughput in test results. Combined with @miletus' latency benchmarking infrastructure, we are able to track latency improvements of these tests as well.

Next Steps: Coordinate With Input

- Address synchronization issues in high latency modes.
- Resurrect the BufferedInputRouter for better vsync alignment.
- Unify BeginFrame message for all platforms.
- AnimationProxy to stay responsive during a long raster.

Next Steps: Improve Latency

- Allow clients to submit frames directly to the UI compositor.
- Latency recovery between Renderer<->UI and UI<->Display.
- Improve deadline prediction.
- Implement long deadlines for immunity to small single-frame jank.
- Improve GPU Service scheduling.
- Improve PPAPI scheduling.

Next Steps: Smart Concurrency

- Increase availabe concurrency between RAF | Rasterization | Compositor.
- Reduce concurrency if CPU bound.
- Explicitly drop frame rate to 30fps.
- Move scheduling logic off the Compositor thread.

Next Steps: Dirty Work

- SingleThreadProxy : public SchedulerClient
- Windows still doesn’t have a HighRes timestamp.
- Estimate GPU Service -> GPU -> Display latency.
- Various code cleanups.
  - Remove CompositeAndReadback and texture locking code.
- Flaky tests and heisenbugs.
  - Scheduling changes are guaranteed to break test assumptions.

Next Steps: Wishful Thinking

- True GPU context priorities and preemption.
  - Will help destroy assumptions about the GPU being a serial resource.
- Run on a 120Hz Pixel or Nexus type device.
  - Chrome’s deep pipeline could be a key feature here.
- Stream GPU context output directly to display.
  - i.e. Have the GPU race the display’s scan out.
  - Saves a full-screen write and read for every frame of dynamic content.

Questions?