Scheduler Overhaul,

with contributions from rbyers, sadrul, rjkroege, sievers, epenner, skyostil, brianderson, jdduke, sievers, nduca, kloba

Status: Deprecated. Ideas are valid but priorities and shape have changed.

Goal

On **all** platforms, with **one** architecture, we wish to:

- Minimize the latency of impl-side scrolling
- Maximize throughput and Minimize latency of javascript-driven scroll, e.g. for
- Maximize throughput of pure rAF-based applications, e.g. JSGameBench, MapsGL or Pong. Some tuning on this goal is deferred to phase 2.

Status

Deadline scheduler landed for Android. @brianderson working on making this work everywhere.

Two Phase Plan

Meeting all these goals is really hard, especially maximizing throughput of rAF and doing renderer-side input coalescing. Therefore:

- Phase 1, this doc: get better rAF throughput, by improving activation and draw time decisions. This gives back a lot toward rAF-based apps.
-
- y to get absolute-best rAF throughput and do input coalescing on the renderer-side.

Background reading

Poke at the docs on Compositor Thread Architecture and Impl side painting, and feel free to ask questions on graphics-dev.

Design Observations

We've learned a lot about tuning-for-latency over the last few quarters, especially via prototype work done by @skyostil to try to improve things and Brians' Zil`CH research.

A "normal" rendering system receives input, processes it, calls the invalidate function on the drawing system, and then lets that system draw based on a draw schedule, all in one thread. In Chrome, inputs get received in the UI thread but handled by the render process, then by the browser if they were not consumed by the renderer. Drawing, similarly, is done by the render process to form a tab's image, then done by the browser to composite that tab plus the browser chrome together. A naive implementation of Android's "traditional" scheduling model in Chrome would result in:

1. Get an input. Send it to render process. +1.5ms
2. Process the input, triggering an invalidate. Send the invalidate to the browser to trigger a view system redraw. +1.5ms
3. Browser begins redraw. Send redraw request to renderer process. +1.5ms
4. Renderer completes redraw and sends draw data to browser process. +1.5ms

That is ~6ms just to do the equivalent that Android would do in a combination of an input event, invalidate and a draw of the view tree. This sort of issue leads us to try to fold drawing with input whenever we are pretty sure that a draw would result from the input.

On top of this challenge, web page rendering suffers from unpredictable and sometimes extremely severe delays when processing input and doing drawing. Some sites do their drawing and input handling in milliseconds, having tuned touch handlers, low invalidation rate, and cheap painting tiles. But, others have huge tile painting costs, super heavy input handlers, crazy forced recalc-style bottlenecks that lead to 300ms+ hitches. In the long term, we want to fix these big stalls. This requires substantial work on Blink, however, and is not a practical solution for now.

To deal with this, we split a frame into a synchronous part and an asynchronous part. The asynchronous part is where we send input to WebKit for handling, and raster tasks to threads for rasterization. At any time while these asynchronous jobs are in flight, we can "draw" the current world -- showing the world as if those inputs had not been handled. This makes a page feel responsive when it is slow to respond, and is key to good-enough feeling performance. However, when one of those threads is sometimes responsive and sometimes unresponsive, we have a uniquely painful challenge. When we begin asynchronous work for the frame, we don't know if that work will come back on-time or not. At this point, we have a few options:

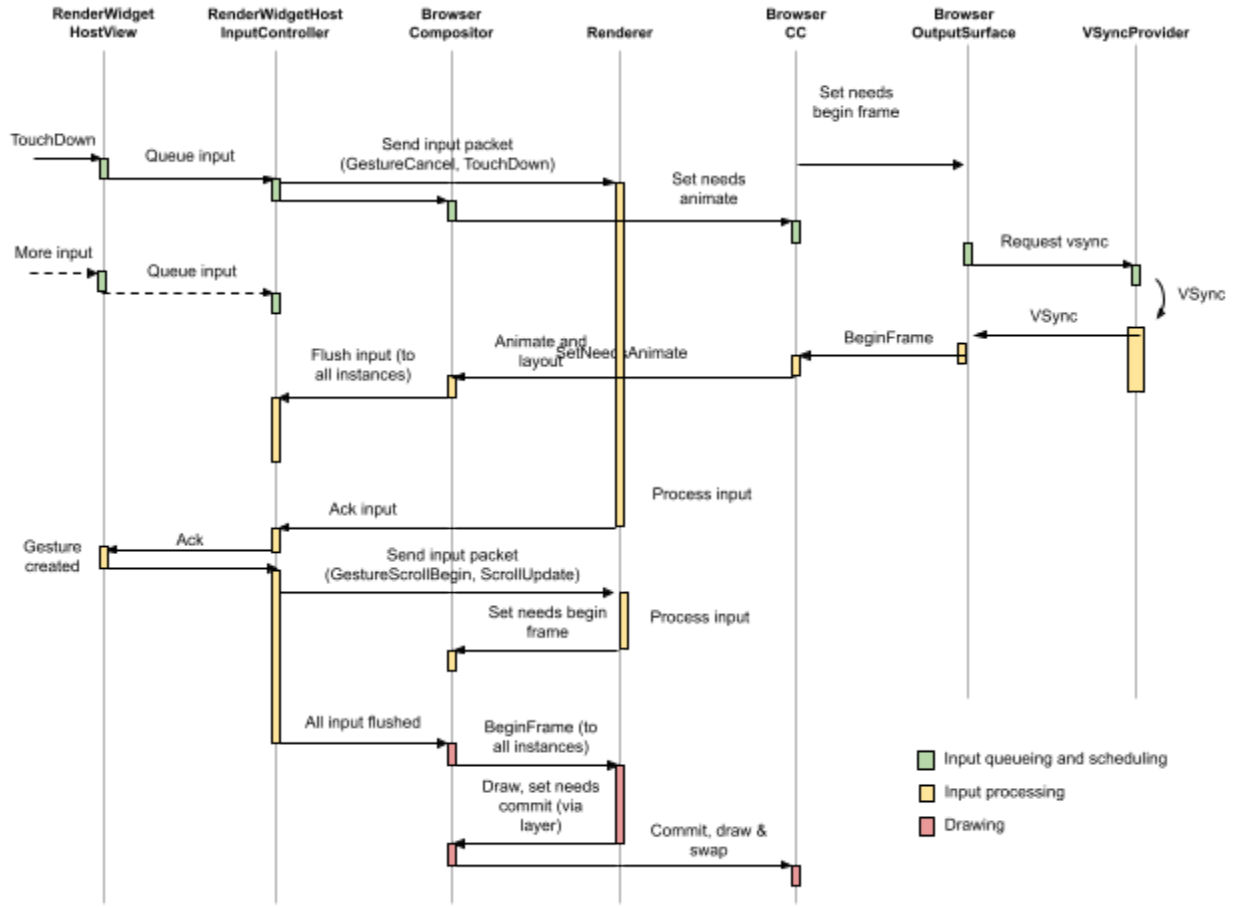
1. Assume it will run long, draw the world up to now, and use the results of the async work in the next frame. This adds 1 frame of latency to each pipeline stage that makes this

assumption.

2. Wait a bit hoping that the work will complete. If no response is received, say in 8ms, draw anyway. If the response is received promptly, congratulations, you are going to space. If no response is received, then draw the world as-is.

w

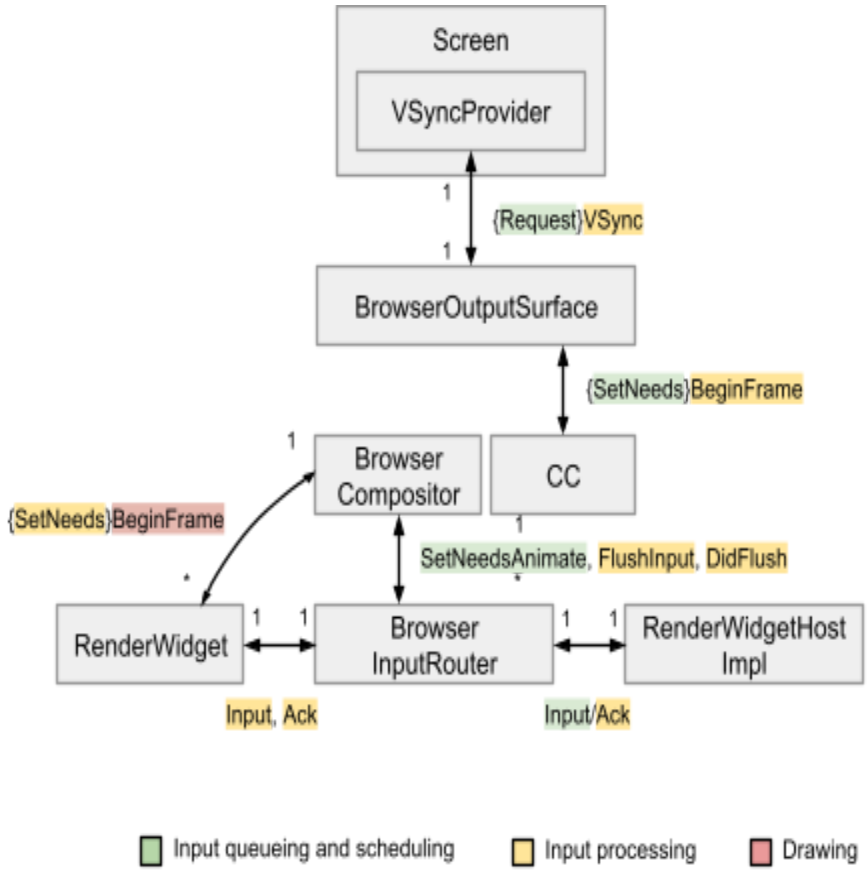
The current chrome does option #1, basically assuming that things will run long. This leads to lagginess but good throughput. We want to try out option 2 --- even with a fixed deadline of 8ms,



77when you have a well-behaved page that is responsive, this wins back a frame of latency. We want to give this a try.

Our Plan

We propose moving all scheduling logic for all compositors to the browser process. Renderer compositors will not self-draw with an independently running ticker. Rather, their draw will be controlled by a BeginRenderFrame message sent by the browser compositor. And, we will modify the browser side input system on all platforms to always send input events **before** the BeginRenderFrame message. This ensures a clean processing flow: process input, begin



rendering a frame, wait for any async work, then draw the frame. All compositors will track a draw deadline. If they send work to a child thread to do rendering, e.g. impl thread tells RendererMain to rAF, then they draw as soon as the response is received [and activated]. If no response arrives by the deadline, and setNeedsRedraw was true for other reasons, then drawing happens anyway. The deadline is propagated along with the BeginRenderFrame message. This gives a chance for the child process to insert its results into the current frame before it gets drawn.

With this work complete, this is the flow of the TouchMove in the browser that follows a TouchDown. When that is acked, we needs to send a GestureScrollBegin and GestureScrollUpdate packet. This is one of the hardest cases to get right:

The remainder of this document describes the various components we plan to build in order to implement this new architecture.

Browser Input & Compositor Architecture

Objective: Send input, then BeginFrame messages to the Render process

Browser InputRouter Design Doc:
here

To guarantee that input events are delivered predictably at the start of a frame instead of whenever they are generated by the platform, we introduce a new BufferedInputRouter class:

Each RenderWidgetHostImpl would own a BufferedInputRouter to which it delegates all input events and the associated acks from the renderer. Since we want to deliver input during animation phase of the browser compositor, BIR schedules animation upon receiving input. This causes the browser compositor to request a BeginFrame message from its output surface. The output surface uses another new class, the vsync provider, to satisfy this request:

```
class VSyncProvider {
   // draw deadline is the output time - (GPU process time prediction)
   typedef void (*Callback)(frame_time, draw_deadline, output_time);
   void SetVSyncCallback(Callback) // Just one callback, called on an arbitrary thread.
   void RequestVSync();
}
```

Once the vsync fires, the animation phase of the browser compositor starts, causing all input controller instances to flush their pending input packets to the renderers. Each renderer will process the given input and respond with an ack. The renderer will also send an additional BeginFrame request to the browser if the input prompted drawing. Once all acks have been received and there is no more pending input, the input controller lets the browser compositor send the BeginFrame message to the renderers to allow them to commence drawing.

Output Surfaces

The FrameRate controller is deleted. The compositor is now triggered by the OutputSurface:

```
class OutputSurface {
  void didHandleInputPacket(inputPacketId);
  void setNeedsBeginFrame(bool enable);

  void didBeginFrame(enum BeginFrameResult, inputPacketsThatWereHandled);
}
class OutputSurfaceClient {
```

```
  void beginFrame(TimeTicks frame_time, TimeTicks draw_by_deadline, TimeTicks
output_time);
}

// Used to switch between 0-latency and 1-frame-deep mode (1 or 2 BeginFrames in
flight)
enum BeginFrameResult {
  DidBeginEarly,
  DidBeginOnDeadline,
  DidBeginAndStillNeedAnother, // A hint that having 2 BeginFrames in flight might help
with throughput, for example if the main thread ran long but drew css.
  DidntBegin, // Main thread ran long. Parent compositor can use this information to
trigger early.
}
```

MaxFramesPending logic moves for now into CCScheduler. It may eventually go up to BrowserOutputSurfaceImpl.

Input System

We will deprecate RenderWidget::ForwardInputEvent, as well as the existing InputEvent IPC model and introduce a completely new input system. This input system will be designed from the ground up to handle input queueing for gestures, regular input, all in a way that is coordinated relative to VSync and BeginFrame messages.

Detailed design doc here:

CC Scheduler

Objective: Send BFAC to main thread on BF, but draw on deadline.

This is a simplified version of how CC scheduling will work for Phase 1. Optimization for particular use cases will be handled in Phase 2:

1. Main thread to impl thread relationship is unchanged.
2. Need to add frameOuptutTime and frameDrawDeadline to LayerTreeClient::animateAndLayout.
3. Scheduler forgets about:
    a. HasActivatedThisFrame, SA_TryToActivate -> due to bug 236648
    b. FrameRateController and MaxFramesPending.
    c. SetTimebaseAndInterval.

4. FramePhase { AtBeginFrame, BeforeDrawDeadline, AtDrawDeadline, NoBeginFrame }
   Set by ThreadProxy based on where we are relative to BeginFrame, and Deadline events
5. Scheduler logic
   a. Change logic for SA_BeginFrame to send when not in NoBeginFrame and one already in progress
   b. WillDraw == (CanDraw && FramePhase != NoBeginFrame).
   c. Sending SA_Draw to depend on WillDraw.
   d. To support Impl-side ResourceUpdateController, AnticipatedDrawTime must be reimplemented because we no longer have a timebase and interval.
6. When RenderWidget's Input handler rejects an input, preemptively SetNeedsCommit on ThreadProxy

Scheduler:
  setDidSendInputToMainThread() // cleared by BeginFrame.
  setDidConsumeInputOnImplThread() // cleared by BeginFrame.
  setInsideBeginFrame(bool)
  setDrawDeadline(TimeTicks deadline) // posts a task for deadline actions.
  setPendingTreeCanBeActivated();

SchedulerStateMachine:
  setFramePhase(FramePhase phase);

CC Activation

Objective: allow activation to happen as soon as the raster work for the frame is done.
Problem: Right now, activation and checkForCompleted is pinned to vsync. Oops.
Next steps: we think this is a small patch, done by the right person. Filed http://crbug.com/236648 for tracking purposes.

FAQ

- How does this work for browsers that don't have a browser compositor?
  Browser still has a BufferedInputRouter and a LegacyBeginFrameScheduler that internally manages and triggers RBF acks

- How does old-style software mode work? Single thread mode?
  InvaidateRect, ScrollRect, ScheduleComposite and ScheduleAnimate all trigger a SetBeginFrame. BeginFrame calls DoDeferrredUpdate.