Controlling BeginFrames through DevTools (headless)

PUBLIC
bit.ly/bfc-v1
Authors: eseckler@
November 7th, 2016

Motivation

In a headless environment, rendering results are only visible in screenshots. Thus, it is usually not necessary to render frames periodically (as it normally happens on vsync) to avoid computational overhead. For some pages, however, it is not enough to only render a frame when a screenshot is requested, e.g. because they rely on animations progressing (CSS animation notifications), requestAnimationFrame being called, or layout to happen periodically to function correctly.

To have better control over how often frames are rendered, we're proposing to add a (headless-only) mode for which the default vsync signal (`BeginFrameSource`) is disabled and `BeginFrames` are issued through DevTools commands instead. This will allow headless embedders to control when/how often frames are rendered.

Some headless embedders also care about the reproducibility of rendering results. For example, if we execute a page until t=10s and then issue a `BeginFrame` to grab a screenshot, we would expect that the resulting frame contains renderer updates from t=10s (and not an old CompositorFrame, a stale renderer main frame, or animation updates from t>10s). For this purpose, we propose a *full-pipeline `BeginFrame` mode* in the compositors, which makes each frame-producing pipeline stage waits for all its children's updates before drawing a frame (browser display waits for renderer surfaces, renderers wait for main frames).

Summary of required compositor changes

Primarily browser-side

- `cc::BeginFrameArgs`
  - Sequence numbers.
- `cc::Surface` / `cc::CompositorFrameSinkSupport`
  - Track ACK/NACK from sinks.
- `cc::Display(Scheduler)`
  - Swap its `BeginFrameSource` with an externally controlled one.
  - Track ACK/NACK from Surfaces.
  - New full-pipe mode that waits indefinitely for all surfaces and draws/swaps once all `Surfaces` have acknowledged the current BeginFrame.
- `cc::CompositorFrame`
  - Store BeginFrame sequence numbers for contained main and impl frame.

Primarily renderer-side

- `cc::Scheduler / cc::SchedulerStateMachine`
  - New full-pipe mode that waits indefinitely for all frame/raster pipeline stages that have updates and commits once all have completed. Also disables latency optimizations, i.e. does not skip `BeginImplFrame` or `BeginMainFrame` and never prioritizes the impl-thread.
- `cc::LayerTreeHostImpl`
  - Attach sequence numbers to `CompositorFrame`, forward NACKs to frame sink.
- `content::CompositorExternalBeginFrameSource / viz::ClientLayerTreeFrameSink`
  - Notify `RenderWidgetHost(View)` on NACK from `Scheduler/LayerTreeHostImpl`.

Approach

In principle, we need to be able to control rendering of different `RenderViews/WebContents` separately, since we may have multiple parallel (and independent) WebContents. However, `BeginFrames` are issued on the browser-level (normally from the device/display through `BeginFrameSources`, which "tick" on vsync) and propagated to the `RenderViews` (originally through `ViewMsg_BeginFrame`, now via the `LayerTreeFrameSink` mojo interfaces). The `RenderView` will then supply a frame to the browser if it has updates.

Normally, the RenderView's frame will only be displayed as a result of the original browser-side `BeginFrame` if it submitted the frame before the BeginFrame's deadline. Otherwise, it may be delayed until a later browser-side `BeginFrame`. Because this is problematic for headless (see section below), we will add a new compositing mode in which the browser ignores the BeginFrame deadline and waits for the frame from each RenderView. Similarly, this mode also ensures that the each RenderView incorporates all pending main thread updates into its frame.

Thus, we need commands/events to, per WebContents:

- Enable the DevTools-controlled `BeginFrame` mode.
  - This disables vsync-driven `BeginFrame` sources.
- Enable the full pipeline compositing mode.
  - This ensures that every BeginFrame waits for all compositing pipeline stages.
- Get notified about `needsBeginFrame` changes.
  - (We can only issue `BeginFrames` while `needsBeginFrame` is `true`.)
- Issue a `BeginFrame`.
  - Specifying the frame time, interval, and (optionally) deadline.
  - If not specified, deadline is calculated from frame time and interval.
- Get notified about commit / NACK of a `BeginFrame` in the browser.
  - This should be the frame committed as a result of the BeginFrame - and includes the `CompositorFrames` of all of the `WebContent`'s renderers.

Full-pipeline rendering mode (infinite deadlines)

For headless, we care about deterministic and complete rendering, but we don't care about missing a vsync deadline, since we don't produce frames for an actual display. Therefore, we intend to add a mode to the compositor schedulers (Scheduler and DisplayScheduler) in which each rendering stage waits for all its children to produce fresh frames before committing them as a response to the `BeginFrame`. In particular, this means that:

- The `cc::DisplayScheduler` in the browser waits until all surfaces (particularly surfaces of RenderWidgetHostViews) have submitted a new frame (or acknowledged that they don't have any updates).
- The `cc::Scheduler` in the renderer issues a BeginMainFrame and waits until the next main frame is submitted (or the main thread acknowledges that it does not have any updates). It also waits for any necessary tiles to be prepared.
- As soon as all children have submitted updates or acknowledged, the deadline is triggered.

To enable reuse of this functionality in other places in Chromium (e.g. during resizes), we intend to:

- Make `cc::DisplayScheduler` (browser) commit immediately when all currently active `BeginFrameObservers` of the `Display`'s `BeginFrameSource` finished the `BeginFrame`. This is similar to the `DisplayScheduler`'s existing early commits, but replaces the existing heuristic with a mechanism that can accurately track active and inactive `Surfaces`.
- Make `cc::Scheduler` (renderer) commit immediately when all stages have completed (main frame, impl frame, raster). This is different to current behavior (commit only on deadline). With buffered input on all platforms (soon), this should be OK as a default behavior.

To implement these changes, we also need to

- Track which `Surfaces` have received and responded to a BeginFrame in the `DisplayScheduler`.
- Detect no-update/abort acknowledgements via the reverse path of BeginFrames through Surfaces, CompositorFrameSinks, LayerTreeHostImpl, and Scheduler. For this purpose, we will add a `BeginFrameAck/Nack` mechanism to these entities and between the renderers and browser. ACK/NACKs will include a sequence number of the BeginFrame that is acknowledged. ACKs can be packaged together with the corresponding CompositorFrame.
- See this design doc for BeginFrame sequence numbers and acknowledgments.

Implementation notes

Proposed DevTools Commands

We can probably only enable the BeginFrameControl and the full-pipe compositing on target creation. Thus, we will add new (headless-only) parameters to `Target.createTarget`:

```
Target.createTarget(..., enable_begin_frame_control, wait_for_all_pipeline_stages)
```

We will further need commands to issue BeginFrames and be notified on acknowledgment:

```
<Domain>.onSetNeedsBeginFrame(bool beginFrameNeeded)
<Domain>.sendBeginFrame(frameTime, frameInterval [, deadline]) returns sequenceNumber
<Domain>.onFrameAcknowledged(sequenceNumber, didCommitNewFrame)
```

The domain for these commands is to be considered (e.g. `Emulation` or a new `Headless` domain).

Controlling BeginFrame propagation to individual WebContents

Headless chrome primarily supports Aura platforms. On Aura, each `WindowTreeHost` has its own `Display`, and each `Display` its own `BeginFrameSource`. If we separate each WebContents into its own

WindowTreeHost, we could thus replace an individual Display's BeginFrameSource to control propagation of BeginFrames to the window's WebContents. This should suffice for an initial version.

Alternative approaches that also support other platforms or multiple WebContents per WindowTreeHost need to consider Display-sharing between WebContents and WebContents that are placed on multiple Displays (e.g. on ChromeOS). Thus they need to multiplex BeginFrames from the same Display/BeginFrameSource to specific Surfaces/BeginFrameObservers, additionally complicating the necessary logic changes to the DisplayScheduler.

Alternatives considered

1. Separate commands to control browser's BeginFrames and view's BeginFrames.

We could disable "automatic propagation" of BeginFrames from the browser-side to RenderViews and control Begin(Main)Frames for browser and each RenderView separately through DevTools. This may be more flexible than issuing BeginFrames through the browser's compositor, but quite brittle. It would require custom solutions for e.g. OOPIFs and risks diverging from expected behavior of the default compositing mode accidentally (e.g. due to future changes to the compositor).

2. Forcing a renderer main frame update for every BeginFrame + utilizing LatencyInfos to track renderer main frame commits to the browser.

DevTools' screenshots rely on forcing a main frame update and tracking the resulting CompositorFrame to the browser using a LatencyInfo attached to the CompositorFrame. An early prototype of BeginFrameControl tried to use (abuse?) forced redraws and LatencyInfos instead of BeginFrameAcks to wait for the renderer's updates. For each BeginFrame, we also issued a ViewMsg_ForceRedraw and the browser's DisplayScheduler would wait for a CompositorFrame containing the corresponding LatencyInfo before drawing.

The disadvantages of this include:

- Forced main frame updates even if there are no changes (unnecessary overhead).
- Abuse of LatencyInfo mechanism for non-tracing logic.
- Timing issues between ViewMsg_ForceRedraw in renderer main thread and BeginFrame in compositor thread.
- Requires quite intrusive changes to cc::Scheduler and cc::DisplayScheduler.

3. Using surface synchronization to force main frame updates and ensure updates have propagated.

Surface synchronization is a new mechanism for synchronizing updates to multiple compositing surfaces, intended primarily to help avoid guttering during resizes. It's based on two principles:

1. A parent in the surface hierarchy assigns new surface IDs for its children (e.g. when a resize begins). The parent's CompositorFrame includes references to these new surface IDs.
2. The display compositor waits for submission of all referenced child surfaces before "activating" the parent's new CompositorFrame (up to a deadline).

We considered using surface synchronization instead of adding the full-pipeline mode. Using surface synchronization, renderer updates could be forced by force-assigning new surface IDs to all clients and waiting for activation of a new CompositorFrame. This would need to be combined with a mechanism to

force a main frame update in the renderer as well. Additionally, it doesn't ensure that all tiles were present for the first CompositorFrame submitted to the new surface IDs.

As some clients may require multiple `BeginFrames` to render results, this approach would have also required adding support for re-running `BeginFrames` to ensure determinism (issuing multiple `BeginFrames` with the same timestamp). Further, setting new surface IDs on each rendered frame would introduce additional overhead when no actual changes are present between frames, as new `CompositorFrames` need to submitted and drawn.

4. Tracking whether a renderer did include all updates into its latest CompositorFrame.

Instead of adding a full-pipeline mode for headless, we considered tracking how "stale" a `CompositorFrame` submitted by a renderer was, i.e. how old the updates from the main thread are that were included a display compositor's last frame. This staleness was tracked as a `BeginFrame` sequence number. We intended to then issue `BeginFrames` until the staleness sequence number matched the sequence number of a sufficiently recent `BeginFrame`.

However, tracking this "staleness" actually turned out to be more complicated than adding the full-pipeline mode described above, primarily because of all the plumbing involved. It would have also required additional changes to support "re-running" a `BeginFrame` with the same timestamp multiple times, until the required updates had propagated through all pipeline stages.

5. Android WebView's OnDraw

Android WebView synchronously draws a `RenderView`'s frames using `LayerTreeHostImpl::OnDraw`. We considered using this `OnDraw` method for headless, too. However, it wasn't a good fit architecturally for headless:

- `OnDraw` requires synchronous IPCs, which we want to avoid in headless.
- This approach would not support integrating updates drawn on different surfaces, such as OOPIFs or canvases/videos rendered onto other surfaces.

6. VisualStateCallbacks instead of BeginFrame sequence numbers / acknowledgments.

To avoid the need for implementing a separate acknowledgment path for BeginFrames, we considered inserting and tracking `VisualStateCallbacks` instead. These callbacks would have allowed us to notice when a main frame update was propagated into the browser. However, making `VisualStateCallbacks` support "no update" acknowledgments required a number of complicated changes as well, and were racy in a non-sync-IPC context.