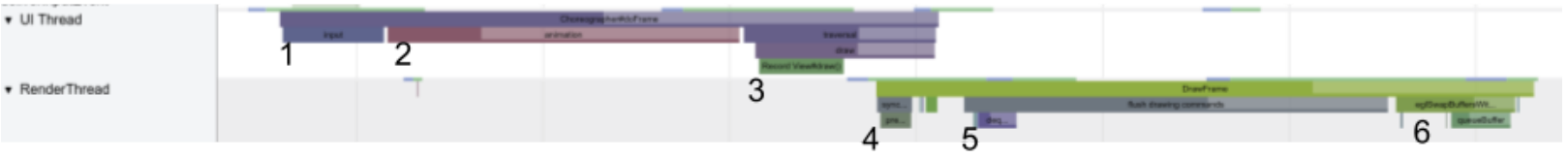
boliu@ android-webview-dev@
2018/02

Geared towards people with chromium rendering background
Renderer compositor, browser compositor, display compositor, compositor frame. High level understanding of how renderer compositor works. GPU command buffer, GPU thread.

Android view background knowledge

This section provides some background information that pertains to (almost) all Android Views.



Android frames are (as of Android 5.0) split between two threads, the UI thread, and RenderThread (RT, no relation to chrome renderer process). These are grossly simplified description of a frame of a typical Android View.

1. Input: Buffered input for that particular frame is delivered, ie this is when View#onTouchEvent and related events are called. Views generally interpret the touch events into gestures, and directly update any necessary view state on the UI thread. If state changes requires redraw, then can call View#invalidate here.
2. Animate: This is when **some** (newer) View animations for a frame are updated, eg those using ViewPropertyAnimator. The animation can change View state, which intern can cause View#invalidate as well.
3. Traversal: This is the heavy lifting step on the UI thread. There are two view tree traversals
    a. First traversal is layout, calling View#onMeasure, and may result in views being resized / moved around.
    b. Second one calls View#draw in pre-order depth first traversal.
        i. draw first tickets legacy animations, including fling in View#computeScroll
        ii. Call onDraw which records drawing commands into a Canvas object.
4. syncFrameState: This blocks on the UI on RT and copies over the recording to render thread. UI is unblocked after. This is very similar to chromium compositor's commit.
5. Flush draw commands: The surface is readied , and recorded commands are played back using GL.
6. eglSwapBuffers: Finally eglSwapBuffers is issued, finishing the frame in the app, and sends a frame to SurfaceFlinger.

Choreographer

A frame contains only a single top level task on the UI thread. Individual tasks of the frame are scheduled by Choreographer. Choreographer internally maintains three task queues (input, animate, traversal), and flushes them in order during a frame. And Choreographer is responsible for flushing its tasks at the “right” time to maintain 60 fps without tearing.

Not all of the queues are directly accessible to apps through the SDK. The calls like postFrameCallback and postOnAnimation allows the animate queue to be accessible. And invalidate implicitly causes tasks to be inserted in the traversal queue. Note only tasks that are already in the queue at the start of that stage are flushed. For example, a postOnAnimation call inside an animate task will cause the task to be inserted for the next frame; however it is still possible to insert tasks in the traversal queue while in animate queue, eg by calling invalidate.

Render thread

RT is not directly accessible to apps through the SDK. It generally responsible for rendering frames only, and runs native code only which means it is not backed by a java looper.

There is a class of render thread only animation (eg the “ripple” effect) which do not require the UI thread. The results of these animations are not immediately observable on the UI thread, and are only updated once the animation is complete. For this reason, this type of animation used rarely to maintain backward compatibility.

Implied requirements and notes

- Aside from RT animations, all operations happen on the UI thread, so the effect is observable immediately. For example, the effect of input events such scroll or zoom are visible after the return of onTouchEvent.
- Scroll offset for a scrollable view is directly accessible to the app through methods such as getScrollX/Y and setScrollX/Y.
- onDraw records the entire view, not only the part that’s visible. The recording format is in vector form, so that an additional 3D matrix transformation can be applied. This is how RT animations work.
- A frame can stop at almost any stage if it is not needed. If there is no invalidate (or layout), then the traversal queue is empty. A traversal may not lead to flushing draw commands, if for example everything invalidated is offscreen.
- Animate is the last point to start a frame with invalidate. Calling invalidate after draw will delay that invalidate until the next frame.
- Ignoring RT animations, “input” to a frame (things that can affect the render output of a frame) can happen right up until the onDraw of a view. For example, onComputeScroll, which ticks fling animations, is called right before onDraw.

Similarity to chromium compositor

Android View	Threaded chromium compositor
View / tree of Views	Layer / LayerTree
UI thread	Main thread
Render thread	Compositor thread
syncFrameState	commit
Render thread animation	Compositor (accelerated) animations
onDraw	Main frame update

Notable things in chromium compositor with no equivalent:

- Pending tree and raster
- Scheduler on the compositor thread

Both systems has high and low latency modes as well, though due to difference in where scheduling decision is made, they are slightly different.

Available hooks

To WebView, the Android View library works as the framework; WebView works by implementing hooks from the framework. So it is important to understand the available hooks from this framework.

Hooks available from the sdk

- onTouchEvent (and other similar methods) from the input queue
- postFrameCallback to post to the animate queue. This also provides the frame time that's should be used for updating animations.
- onComputeScroll from draw queue
- onDraw from draw queue

Draw functor

This is a hidden API used explicitly for implementing webview. The API is not documented publicly and the exact contract changes between Android versions. So app authors reading this should not take this as a license to use the draw functor.

The draw functor allows inserting a native (c++) function pointer into the recording stream on the UI thread. The functor is then called

- Inside syncFrameState, when the record stream is copied to RT
- During playback, inside flush draw commands

Draw functor allows webview to bypass the record mechanism, and get hooks directly on the render thread instead.

Integration with chromium compositing for chrome

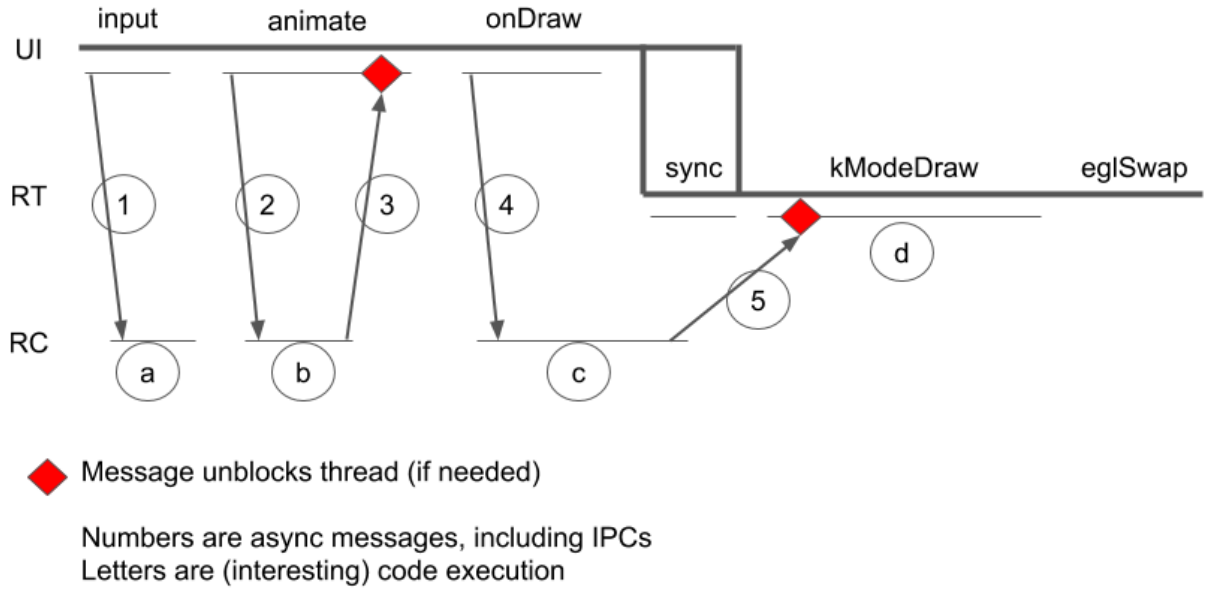
Chrome creates a separate SurfaceFlinger surface with SurfaceView, and composites into this separate surface instead of using the same surface as other views. This gives chrome greater control over scheduling among other things, which means chrome does not rely on as many hooks from Android.

Chrome at a high level implements onTouchEvent by sending the events to the compositor thread. And chrome uses postFrameCallback as the implementation of the BeginFrameSource, which drives begin frame signals. Both calls sends asynchronous messages to the compositor thread instead of being handled directly on the UI thread.

Integration with chromium compositing for webview

Webview is a “regular” Android View, ie it composites into the same Surface as other Android Views. Thus chromium compositing in WebView is integrated more deeply with Android View compositing. WebView, in theory, has to meet all implied synchronous behavior of Android Views as well, which dictates the design here.

The key component is synchronous IPC from UI thread to the renderer compositor thread. The rationale is explained in more detail in this doc. In the diagram below, synchronous IPCs are broken into constituent parts. RT is the render thread; RC is the renderer compositor thread.



- onTouchEvent: Implementation is same as chromium, ie async IPC (Message 1) to the compositor thread which handles or dispatches event (code a).
- postFrameCallback:
    - Tick root layer fling, if computeScroll has been called before.
    - The BeginFrameSource part is the same as chromium. The begin frame will tick all cc animations except root fling. Renderer compositor will also invalidate the frame sink in response to a begin frame (code b) instead of scheduling a deadline) if it wants to produce a frame.
    - Send a synchronous IPC (message 2, 3) to effectively “flush” all previous async IPCs, and send the resulting state (new scroll offset, invalidate state, etc) back to the browser. This allows the browser to invalidate the WebView inside postFrameCallback (ie Choreographer animate) so it does not introduce a frame of delay. New renderer state is available to subsequent Android View layout.
- computeScroll: Set state that computeScroll is called, ever.
- onDraw:
    - Send async IPC (message 4) to renderer compositor thread to draw (code c, ie LayerTreeHostImpl::OnDraw). Frame production is not fully asynchronous however. Each async IPC is also accompanied with a FrameFuture, which can be waited to get the delegated frame from the renderer. The frame is fulfilled on the browser IO thread when renderer sends the reply to the draw IPC (message 5).
    - Insert functor into Android’s recording stream, in order to get functor callbacks.
- (on render thread) functor callback in syncFrameState: Copy frame future from UI thread to render thread.
- (on render thread) functor callback in playback (code d):
    - Wait on frame future copied in last syncFrameState
    - Submit frame to viz::Display
    - viz::Display::DrawAndSwap to produce draw commands into the command buffer. Note the OutputSurface does not issue swap; it only flushes the command buffer.
    - The command buffer in viz::Display is specialized to run commands on the render thread as well. Flush the command again to ensure all GL commands from viz::Display run immediately.

Deviate from requirements

The astute reader might notice this does not fulfill all synchronous requirements. These tradeoffs are made for performance reasons, essentially to reduce synchronous IPCs and thus waiting.

- Input: Input is an asynchronous IPC, which means the result is not immediately observable after onTouchEvent returns. Note though it is still observable visually in the

frame, so there is no additional latency introduced. No bug reports was received from this change, so probably very few apps are actually relying on this behavior.

- SetScroll: <double check this> There is a copy of scroll offset on the UI thread, which WebView client can get and set immediately on the UI thread. Scroll update from compositor can also update this copy of scroll offset; there is no complex logic for reconciling “concurrent” updates. This is generally not a big problem because it was never possible to reconcile SetScroll and js scroll updates.
- Zoom: WebView also has APIs to control zoom. These are implemented as synchronous IPCs, because there are apps that just call zoomIn in a loop until a certain zoom level is reached.
- computeScroll: computeScroll was initially made an async IPC without any practical compatibility issues. Following that, ticking the root fling is moved out from computeScroll to postFrameCallback, and computeScroll only updates a state that it’s ever called; this is to maintain compatibility with apps that override computeScroll to no-op to prevent fling animation in webview.
- Scroll from compositor after postFrameCallback. Draw is async (on the UI thread), so any scroll in the compositor thread after begin frame but before draw will not be observable immediately on the UI thread. Most sources of scroll update for a frame has been moved to before or during begin frame, so are synchronized back to UI thread correctly; the only exception is scroll from js from a commit. This does cause synchronization issues, but is rare enough to not be a big problem.

Command Buffers: Texture sharing EGLImage and Mailbox abstraction

WebView runs the GPU service thread for the viz::Display on the Android Render Thread. The Render Thread is only accessible from the functor, so it’s not appropriate for running the general chromium GPU service. Thus WebView actually has two GPU service threads, the Render Thread which runs GL viz::Display, and the “gpu thread”, which runs GL from everything else in chromium.

Chromium shares textures between viz::Display context and other systems through the Mailbox abstraction, which is a global name for a piece of memory buffer that can be converted to something “renderable” in a specific command buffer context. Chromium uses a single GPU service thread, so Mailbox for texture is simply implemented as context <share??> group.

Context group across threads on Android drivers have never been extensively tested, so WebView opted to use EGLImage instead for sharing textures across EGLContexts on two different threads. The idea is every time the command buffer signals a sync point, convert all textures into EGLImages and save it in a global structure. Then every time a sync point is waited on, create textures from the EGLImages in the current context. This is implemented in MailboxManagerSync and TextureDefinition.

This works well most of the time, with some caveats.

- Some textures from Android are already backed internally by EGLImages, and the spec do not allow more EGLImages to be created from these. An example of this is video textures from Android. WebView gets around this making a GPU copy of the texture before sharing <link>
- Some drivers incur a GPU copy when creating EGLImages.
- EGLImage sharing is also the reason WebView cannot and probably will never run the GPU thread out of process.
- This is one concrete reason why WebView rendering is potentially slower than Chrome.

<TODO: Add note about in process command buffer and render thread scheduling>

Resourceless Software Draw

On Android 2.x, Android Composition were entirely software based. View.onDraw is passed a Canvas that is backed by a bitmap, and any draw commands is directly rastered into the bitmap. This mode can still be used by apps that

- have not been updated since hardware composition is the default on Android, ie apps that target older versions of Android
- call View.setLayerType(LAYER_TYPE_SOFTWARE)
- Create a Canvas and call into View.onDraw directly. This is the most intentional use case by apps, since it is allowed by the sdk, and is the only way (that works across all Android versions) to capture WebView content into an app manipulatable format.

Historically, WebView has also exposed methods such as capturePicture or PictureListener which was supposed to return the Picture, ie the SkPicture, of the entire webpage.

WebView implements these legacy features with the “resourceless software draw”. The idea is that the renderer compositor (today, renderer layer compositor and a viz::Display both on the renderer side) directly rasters layers into the bitmap during draw, without using resources (ie tiles, which are in GPU memory). All of this is synchronous, ie in a single function call on the UI thread, meaning this work cannot involve the blink main thread. There are a number of caveats as well:

- Only PictureLayer or SolidColorLayer works. This means features like video, webgl, accelerated canvas etc do not show up in the software frame
- Note this is generally not a “software mode”. To WebView, software and hardware can change on a frame by frame basis.
- Resourceless draw has the ability to capture any part of the document, including parts that are outside of the viewport. This is the reason for external viewport and matrix in LayerTreeHostImpl::OnDraw.
- To avoid regressing performance (too much), WebView uses private APIs to access the memory backing of the SkCanvas directly. There is skia API added specifically for

webview to do this across skia versions. In the best case scenario, renderer side directly rasters into the final canvas, incurring no additional copies.

- In the worse case, there can be two additional copies
    - There are cases when the skia API does not work, eg trying to capture complex clip data. In these cases, WebView allocates an auxiliary bitmap for renderer side to raster into, and then draws the bitmap into the canvas using Java code
    - In multiprocess (enabled starting on O), need to allocate shared memory, which similar to above incurs an additional copy.

In general, resourceless software draw need to work in its weird/limited way probably indefinitely. However there is no need for it to be particularly fast, or even 100% correct.

Other quirks

- Record whole document. By default, compositor records a rectangle around the current viewport rather than the whole document. However to support resourceless draw's ability to capture the entire document, webview has an option (disabled by default) to capture the entire document. On large documents, this can be incredibly slow.
- There is no guarantee that an invalidate call to Android will lead to a draw call for WebView. However there are features on the renderer side that requires draws to continue, and the classic (non-chromium) WebView behavior was to continue this work even if WebView is never drawn. So WebView implements a "fallback tick", ie a fake draw in case a real draw does not come within a timeout, that just moves things forward.
- Chromium compositing has a goal to reduce checkerboarding; the general strategy is to limit when "inputs" into a frame can happen, so to maximize the amount of time to produce the frame, in particular to raster tiles. Inputs include actual user input, but for webview, they also include things like app manipulation of scroll, zoom, transforms, etc. Thus this effort is essentially a lost cause.
- Chromium compositor's overlay system relies on the ability in the viz::Display to update overlays instead of pushing a new frame to the surface. This strategy works very poorly on WebView because by the time functor is called, a frame will be pushed to the app's surface regardless. So the decision that a frame is an overlay-only update needs to happen significantly earlier for WebView to benefit from overlays. This is the main reason WebView does not use any overlays.

History

- Kitkat: Chromium-based WebView first shipped in Kitkat. The first release was based on M30. In this time, Android has not split off render thread from the UI thread, so everything was happening on the UI thread. The implementation at the time merged the renderer compositor thread and the UI thread, so that renderer compositor can be accessed directly in UI calls. Everything except WebGL was using the GL context on the render thread. This is very close to how classic WebView is architected.
- Lollipop: Android split out the render thread in Lollipop. WebView took advantage of delegated compositing which shipped around the same time. UI onDraw still called to

renderer compositor directly which produced a delegated frame. The frame is passed to render thread, which has the “parent compositor” consume the delegated frame and draw with GL.

- M42: Using the context on render thread caused a lot of deadlocks because chromium code expected the GPU thread to always be available, which WebView cannot fulfill with render thread. So an effort to move all context from render thread to the separate GPU thread was made. In M42, everything except video (which cannot use EGLImage sharing without an extra copy) and the final composite is moved off to separate GPU thread.
- M50/Nougat: WebView became multiprocess (sandboxed renderer process) in N (off by default). The major blocker was the merged thread architecture used up until this point. A rewrite to split the threads, and switch to using synchronous IPCs was made for multiprocess, which is still the underlying architecture to this day (2018). The first version shipped in M50, and was being maximally correct, and can have up to 4 synchronous IPCs per frame per WebView; this was also significantly slower.
- Reduce synchronous IPCs:
    - Immediately in the following versions, input and computeScroll were made asynchronous without any practical compatibility issues.
    - Draw (except fling) was made semi-async with the futures pattern without breaking compatibility.
    - Draw for fling was made async by breaking computeScroll contract, which few apps relied on.
- Reducing unnecessary invalidate. Chromium compositor has the ability to stop a frame at any stage in order to save power. This generally did not work because WebView does not control its own compositing, and once invalidate is called, Android will produce a frame no matter what WebView does. So thus an effort was made to at least have the no-damage optimization (mostly) work WebView as well.