Synchronous compositing for WebView

aelias@chromium.org

(doc originally written in Jan 2013, copied to chromium.org and updated for reference)

Problem

The Chrome compositor now has a software mode, but that on its own doesn't solve all the uses/abuses of existing Android WebView embedders. In the existing Android WebView, rendering is:

- **Nonmodal**. Nowhere in the WebView API does the user explicitly specify this WebView should be in "hardware mode" or "software mode". Rather, canvases are provided one by one to the draw call and they can be either hardware-backed or software-backed. Although hardware draws can only happen when the WebView is attached to a hardware-accelerated View tree, software draws to a side canvas may occur at any time.
- **Synchronous**. The draw call must block until it has finished rendering the frame into the canvas.
- **Just-in-time**. The scroll offset and scale factor (which may also be applied via matrix) may have changed just a few instructions before the draw call. The WebView may also be initialized to be larger than the screen, and each frame only draws to the current clip rect.
- **No missing content in software**. If the draw is to a software canvas, there are never any missing or blurry tiles (but if it's a hardware-backed canvas, checkerboard is OK).

Here are two representative examples of use cases of these properties:

- A third-party web browser backed by WebView, with an onscreen hardware canvas allowing the user to scroll around. It wants to take a scaled-down "thumbnail" of the entire document. To do this, it creates a small software canvas and applies to it a matrix with a 0.1 scale factor, then calls webView.draw(softwareCanvas). (This is the recommended way to take thumbnails, favored over capturePicture which is semi-deprecated.)
- An application initializes the WebView to the size of the underlying document, which is 20 viewports high. It then moves it around with a matrix, ignoring the WebView's concept of scroll offset entirely. It doesn't expect to see any overdraw or other performance impact.

I think the Chromium WebView needs to make a serious attempt to reproduce these properties of the Android WebView precisely and not simply approximate or hack around them, or the backwards compatibility problems may be intractable. Fortunately, impl-side painting unlocks ways of doing so.

Solution outline [updated]

Here is the simplest plan I've come up with that fulfills all the above requirements. It depends on impl-side painting.

First, we would take advantage of WebView being single-process and **merge the renderer compositor impl thread with the browser UI thread**. This would make our threading model closely match Android browser's (which has a UI thread, a WebKit thread and a raster thread). Our WebKit thread just talks to a message loop of some kind and doesn't really care whether it's a newly created thread or a wrapper around an existing one.

Gesture event flow:

1. A gesture comes in from Android Java code.
2. Where we would have sent an IPC, we instead send to a new class SyncInputEventFilter, which calls CC directly. There are then three cases:
    a. If the event hit the root layer, CC rejects it and lets WebView-specific code handle scrolling.
    b. If the event hit a sublayer, CC absorbs it and scrolls the sublayer.
    c. If the event hit a slow-scroll region, CC rejects it with HandleOnMainThread and the SyncInputEventFilter sends an IPC to the render process.

Whenever WebView is called to draw:

1. In WebView.onDraw(), take note of the current transform, scale and clip rect, taking into account all the latest Android matrices.
2. Set some state on the LayerTreeHostImpl with that information.
3. Call CC via "beginFrame"/"deadline" to immediately start drawing.
4. Compute layer draw properties, adding the additional transform and clip rect at the root layer during calcDrawProperties.
5. Here the approach forks:
    - If software canvas (specified by OutputSurface::ForcedSoftwareDraw):
        i. Walk the layer tree in a mode that only produces resource-free DrawQuads. (We would have "virtual bool supportsForcedSoftwareDraw() const;" to LayerImpl that only returns true for PictureLayerImpl and SolidColorLayerImpl. Secondly, PictureLayerImpl would be told to only produce PictureDrawQuads.)
        ii. Create a local SoftwareRenderer just for this frame, and raster the PictureDrawQuads onto the software canvas.
    - If hardware canvas: Produce GL commands as usual and push them to a command buffer.
6. Now return from WebView.onDraw().
7. If it was a hardware draw, the draw functor will be called; without changing threads, make the command-buffer execute the entire GL stream immediately against the system

context, and clean up the GL state to leave the system context unchanged.

Unpacking some of the implications of this approach:

- "Normal" software compositing is only partially used - a temporary cc::SoftwareRenderer instance can be used that employs the 'tile free software compositing' mode.
- We don't have a browser compositor.
- The renderer compositor doesn't spontaneously schedule frames. It doesn't do any drawing until it's called in the draw functor.
- Given that the software path doesn't need them, we want to hold off on generating any tiles until the first time we receive a hardware draw.
    - Hence TileManager will have a memory-limit of zero whenever we are not attached to a hardware-accelerated view.
- Performance-wise, this software path has the following trade-off:
    - Slower at scrolling/pinching than tiled compositing because SkPicture performance is nondeterministic.
    - Faster at going from a fresh invalidate to the final canvas because there is no tile in between.

  This tradeoff is reasonable given that software WebView is commonly used for thumbnail/snapshot style uses. Note that Android browser's software path also rasters directly from SkPictures, so the scrolling performance wouldn't constitute a regression.
- We cannot issue any GL outside of draw functor calls since we're not permitted to use the context outside of draw functor. However, we're allowed to create and use gralloc() buffers for GL tiles at any time.
- We can reuse essentially the same software draw path for both capturePicture and regular software compositing, since an SkPicture can also be the target of a draw. The Skia R-tree optimization should do the right thing if we apply a clip and matrix to the canvas.

Render surfaces and alpha

For hardware draws, render surfaces and alpha blending should "just work". However, for software draws, problems arise:

- Draws outside of view tree attachment may happen. If we create a render-surface resource in these cases and hold onto it, we may never receive another draw call and there's no obvious time when we should deallocate it.
- For alpha blending between different layers within CC, the code normally relies on the existence of tiles as a buffer to do it correctly. But we have no tiles.

In both these cases we have the option of either creating a per-frame single-shot render surface, or ignoring the web platform specification and drawing incorrectly. As the classic WebView ignores the web platform specification here as well, for v1 we will go with the latter:

- In the case of advanced render-surface-based features such as clipped 3d transformed layers and CSS filters, simply ignoring them is fine as no existing WebView apps can be using them.
- For alpha, we need to support it in some form. The current code hacks the SkPicture at draw time to draw directly onto the root layer. This will lead to incorrect color values in complex layers because alpha blending is not commutative, but in simple cases it will look OK.

In the long run, the only way to become web-standards compliant for software draws is to support software tiling in WebView (since mallocing huge intermediate buffers isn't viable).

Promotion to hardware rendering mode

The WebView only gets access to a GL context if and when it's first attached to a hardware Android View. Prior to that, it will only receive software draw calls. Therefore, we must initialize CC in a baseline software-only mode, while being ready to get suddenly "promoted" to a full GL renderer. (Note that even after this promotion, we still need to support software draws at any time -- therefore this full GL mode is a strict superset of the software-only mode.) When we're detached from the hardware View, we should evict all the GL resources and get demoted back to baseline software mode.

When the WebView is attached to a hardware accelerated Android view tree, then a synchronous call is made to initialize CC into GL state, immediately prior to the first draw on the UI/compositor impl thread. The hardware initialization must be synchronous, and cannot wait for the main thread.

In terms of resource allocation, the classic Android WebView allocates GL tiles but never allocates software tiles. As mentioned above, we want to preserve this behavior. Therefore, we should avoid tile allocation while in the software-only mode, and at the moment of hardware initialization, create a TileManager and start to kick off raster tasks.
Similarly, if a layer such as a VideoLayer that we cannot display in baseline software mode was attached prior to GL initialization, we need to make sure it becomes displayable at this time as well. The output-surface recreation path originally designed for lost-context recovery does most of what we need here, both for hardware promotion and demotion.

List of problems:

- Cannot call LayerTreeHost::DeleteContentsTexturesOnImplThread since it the PrioritizedResourceManager cleanup depends on main thread being blocked
    - Note that we should never use PrioritizedResourceManager in the renderer on

    Android anyway. So we can simply if() out this code.
- Cannot update new capabilities/settings on the main thread
    - List: allow_partial_texture_updates, MaxPartialTextureUpdates, using_accelerated_painting
    - Plan is to set the set the settings correctly when initializing the software mode, but not make use of these settings until hardware initialization.
- The offscreen context will be set on hardware initialization, regardless of whether it's needed
    - Only set it when its needed, which requires moving that decision off the main thread.
- Whether to clean up resources in impl trees
    - Software mode should not be allocating resources, so should be fine not to clean up. But need to make sure this is the case with tests/DCHECKS
    - Alternatively, call clean up and make sure all layers do clean up their resources instead of leaking

Alternatives considered

Here are a bunch of other ideas I had that provide the desired synchronous behavior, but run into other difficulties of performance or complexity.

- **Keep the threads separate and do a synchronous IPC on draw (preferred solution in a previous version of this doc).** Draw is the main place where we care about synchronicity. We could block the browser UI thread on the renderer compositor impl thread during the draw to get synchronous behavior.
    - Problems: A) This would introduce a dependency on delegated rendering since the GL must be executed on the browser UI thread, and make the architecture generally more complicated. B) Apps also expect to be able to synchronously set the scroll position and it doesn't seem we could be precisely backwards compatible with this -- more generally, there may be "unknown unknown" subtle synchronicity assumptions made by apps that we would be unable to support in this threading model.
- **Block on WebKit main thread**. If we do all our software painting on the WebKit main thread, our behavior there will be compatible. It's tempting to say we would use WebKit-thread blocking for thumbnail-style uses and software compositing for scrolling-style uses.
    - Problems: A) We'd also be blocking the embedder's UI thread, exposing it to a high level of jank which never happened in the old WebView. Even in the one-shot thumbnail case, janking the embedder for 500ms is not acceptable. B) There is no reliable hint or heuristic we can use to decide if a given software draw would prefer performance or correctness.

- **Use software compositing with tiles**. Instead of painting directly from SkPictures in software mode, we could issue tile paint jobs to the raster thread at the latest scroll and scale, block until they're complete, then draw using the SoftwareRenderer. This would require the LayerTreeHostImpl to hold both a SoftwareRenderer and GLRenderer, taking advantage of the fact that our tiles are grallocs and therefore can be used by either.
    - Problem: It turns out that grallocs cannot be read back easily after all. The management of two types of buffers in CC at once would be complex and it would be difficult to avoid overusing memory.