Multithreaded Rasterization

@nduca, @enne, @vangelis (and many others)

Implementation status:

crbug.com/169282, and https://code.google.com/p/chromium/issues/list?q=label:Cr-Internals-GPU-ImplSidePainting

This feature is referred to as "multithreaded painting" and "impl-side painting" in some forums. It is now launched on Chrome Android 25.

Background & Problem Statement

Our current compositor thread architecture is built around the idea of rasterizing layers on the main webkit thread and then, on the compositor thread, drawing the bits of the layers that we have based on their various animated and scrolled positions. This allows us to move the page up and down, e.g. due to finger dragging, without having to block on the webkit thread. When a tile is exposed that does not have contents, we draw a checkerboard and wait for the main thread to rasterize that tile.

We want to be able to fill in checkerboards without requiring a new commit, since that requires going to a busy webkit thread and pulling in a whole new tree + damage. We also want to be able to render tiles at multiple resolutions, and quality levels. These kinds of tricks reduce memory pressure, avoid the jarring interruption of checkerboards.

The Excessive Checkerboarding Problem

A lot of our unwanted checkerboarding comes from invalidates getting intermixed with "requests" from the impl thread to fill in missing tiles. In the current architecture, we can only rasterize tiles on the main thread, using webkit's rendering data structures. If webkit's rendering tree is completely unchanged, then the page scrolls, all the rasterization requests that go to the main thread are easily satisifed by webkit.

However, any time javascript changes the rendering tree, we have the following problem: we have some "newly exposed tiles" that the compositor thread needs to prevent checkerboarding. But, annoyingly, any of the previously-painted tiles that webkit says were invalidated. We can only paint the new rendering tree -- the old rendering tree is gone. So, we have two options at this point:

1. Draw the new tiles with the new rendering tree, and redraw the old tiles with the new rendering tree

2. Draw only the new tiles, and let the old tiles stick around.

#2 doesn't work well at all, of course: if you have a page that toggles between green and blue constantly, what you'd see is a random mix of green and blue page at any given moment. We want to preserve the "atomicity of rendering" --- meaning that the complete state of a web page at rAF time is what gets put on the screen.

There is a variant on 2 where we draw the new tiles, as well as any old tiles that are *onscreen*. If a tile is offscreen, then we make a note that is is invalid, but dont repaint it. In the green-blue scenario, this causes the screen to be green or blue, but never both, as long as you dont scroll. We ship this on Chrome Android m18. Even so, this is undesirable: if you scroll, you'll see a mix of content. This is expedient performance wise, but makes us all feel dirty.

Our other source of heavy checkerboarding is latency related. The work we do on the main thread is based on as scroll position update message that comes from the impl thread. This message is itself not very latent, arriving on the main thread milliseconds after it is sent. However, paints for a new set of tiles can take 300ms + to complete, even with the relaxed atomicity approach described above. By the time we have painted all 300ms worth of work, the page has scrolled way past the original scroll position, and half of the tiles we worked hard to prepare are irrelevant. We have discussed a variety of solutions here, but the real core problem is that the main thread cannot be updated fast enough with the new scroll positions to really ever keep up properly.

Planned Solution

Display lists. Namely, SkPictures, modified a bit to support partial updating. We call this a Picture pile, a name borrowed from the awesome folks behind Android Browser. The idea is to only capture a display list of the webkit rendering tree on the main thread. Then, do rasterization on the impl thread, which is much more responsive.

On main thread, web content is turned into PictureLayers. Picture layers make a recording of the layer into a PicturePile. We track invalidations in SkRegions and during the display list capture process, decide between re-capturing the entire layer or just grabbing the invalidated area and drawing it on-top of the previously recorded base layer.

During commit, we pass these PicturePiles to a PictureLayerImpl. Recall, layers can change in scale over time, under animation, pinch zoom, etc. To handle this, a PictureLayerImpl manages one or more PictureLayerTiling objects (via a PictureLayerTilingSet), which is a decomposition of the layer's entire contents into tiles at a picture screenspace resolution. So for example, a 512x512 layer might have a tiling into 4 256x256 tiles for a 1:1 ratio of screenspace pixels to content pixels, but also 1 256x256 tile for a 1:2 ratio of screenspace to conten space. We manage these tilings dyanmically.

A tiling itself takes the layers entire size, not just the visible part, and breaks it up into Tiles. Each tile represents a rectangle of the PicturePile painted into a Resource ID [think, GL texture], at a given resolution and quality setting.

Every tile is given a set of TilePriority values by the PictureLayerImpl based on its screen space position, animation and scroll velocity, and picture contents. These different priorities encode how soon, in time units, the tile could be visually useful onscreen. Key metrics are things like "how soon will it be visible" and "how soon will it be crisp" and "is this a tile we'd use if a crisp one wasn't available?"

These Tiles are registered to the TileManager, which keeps these tiles sorted based on their priority and some global priority states. Tiles are binned in orders of urgency (needed now, needed in the next second, needed eventually, never going to be needed) and then sorted within their bin. The total GPU Memory budget is then assigned in decreasing priority order to these tiles. Tiles that are given permission to use memory are then added to a rasterization queue if needed.

The raster thread scheduler is a very simple solution: on the impl thread, we simply pop from the raster queue, dispatch the raster task. We keep a certain number of jobs enqueued per thread, opting to not enqueue them all so that if the prioritization changes much in the future, we wont do redundant work.

JPEG/PNG/etc bitmaps are stored in the display lists in still-encoded form to keep display list recording cost low and memory footprint small. Thus, the first time we draw a bitmap, a costly decode and downsample operation may be needed. Thus, before dispatch, tiles are "cracked open" to determine whether any bitmaps need to be decompressed, using the SkLazyPixelRef interface to WebCore's ImageDecodingStore. If decoding is needed, the tile is held in a side queue while a decoding task is dispatched to the raster threads. When the decodes are done, raster tasks are enqueued.

This approach fixes the “atomicity of commits” problem by allowing us to servie checkerboard misses without havin to go to the laggy, potentially changed main thread. In the previous example, when the compositor sees a checkerboarded tile, we can rasterize it without having to start a commit flow, allowing us to disallow commits entirely during flings and other heavy animation use cases.

Hitch-free commits

A key challenge with this approach is switching from the old tree to the new tree. In the existing architecture, when we go to switch to the new tree, we have painted and uploaded all the tiles, so the tree can be immediately switched.

In the impl-side painting architecture, we need to create PictureLayerImpl's in order to begin rasterizing them. Moreover, those impls need to be attached together to the LayerTreeHostImpl in order to get their screenspace positions, which are essential in computing their priorities.

The obvious way to do this is to simply commit the main tree to the impl tree like we usually do. However, if we do that, then the impl tree now has holes in it where there were invalidations. At this point, the impl-side has two options when vsync comes around: checkerboard, or drop the frame. Neither is very cool.

Our solution is the LayerTreeImpl. Whereas the previous architecture's LayerTreeHostImpl had a root layer and all its associated state, we instead introduce LayerTreeImpl, which has all the state associated with a layer tree: scrolling info, viewport, background color, etc. The LTHI then stores not one, but two LayerTreeImpl's: the active tree is the one we are drawing, while the pending tree is the one we are rasterizing. Priority is given to the active tree, but once the pending tree is fully painted, we activate it and throw away the old one. This allows us to switch between old and new trees without janking.

Handling Giant SkPictures

One potential challenge to impl-side painting compared to our existing painting model is that the SkPicture for a given layer are potentially unbounded. We plan to mitigate this by limiting the PicturePile's size to a 10,000px (emperically determined) portion of the total layer size cenetered around the viewport at the time of the picture pile's first creation. When the impl thread starts needing tiles outside the pile's area, we will asynchronously trigger the main thread to go update the pile around the new viewport center.

Choosing the scale at which to raster

Whenever we compute the draw properties for a PictureLayerImpl, we also decide what tilings it should have, or in other words, at what scales it should have sets of tiles. To do this we track two scale values: The ideal scale, and the raster scale. The ideal scale is the scale at which we should create tiles to give the texels in the tile a 1:1 correspondence with pixels on the screen. The raster scale is the high-resolution scale at which we are currently creating tiles. When we set the raster scale to be equal to the ideal scale, we get crisp tiles. This is what we'd like to have at all times, but we limit this for performance reasons. During a pinch gesture, or an accelerated animation, the raster scale lags behind the ideal scale. CSS can change the scale of a layer through the DOM, and we limit how often it is allowed to change the raster scale. This decision to reset the raster scale to the ideal or leave it alone is made in PictureLayerImpl::ManageTilings. Whenever the raster scale changes, we add a tiling both at the raster scale, and at a low resolution related to the raster scale. These tilings are marked as HIGH_RESOLUTION and LOW_RESOLUTION and are given priority as we raster tiles for the layer.

Texture Upload

One key challenge on lowend devcies is that uploading a single 256x256 texture can take many milliseconds, sometimes as crazy as 3-5ms. Because of this, we have to carefully throttle our texture uploads so that we dont drop a frame. To do this, we are adopting a new approach of async texture uploads. Instad of issuing standard glTexImage calls, we instead place textures into shared memory and then instruct the GPU process to do the upload when-convenient. This enables the GPU process to do the upload during idle times, or even on another thread. The compositor then polls the GPU process via the query infrastructure to determine if the upload is complete. Only when the upload is complete will we draw with it.

Handling setPictureListener

If the embedder has a picture listener, we need to send a serialized SkPicture to the embedding process. We would need to, at every impl-side swapbuffers, serialize our SkPictures for all the active layers (plus the bitmaps) and send them to the main thread.

Followup Work

The initial impl-side painting implementation is expected to enable the following followup use cases:

> **Low-res tiles:** For tiles that take a long time to rasterize, we may want to rasterize them at half or third resolution. This often dramatically reduces (5-6x anecdotally) raster cost and allows us to avoid checkerboarding during fling. However, it is worth noting that some Android users criticized this behavior on ICS devices as making fonts look too ugly. High-dpi devices may change the UX impact of this behavior on users.
>
> **Just-in-time scaling:** We currently do resizing of content at many layers in the pipeline. For example, we rasterize layers at their content resolution without consideration to their screenspace transform. Thus, a layer that is -webkit-transform: scale(0.5)'d will actually paint at its full size. Similarly, we resize images inside webkit at their content resolution. We could reduce rasterization/decode costs and memory footprint if we could do all of this scaling using the draw-time transforms on the impl thread.
>
> **Accelerated painting:** An interesting property of impl-side painting is that it cleans up our accelerated painting story. We would store the SkPicture for a layer, and then can decide to rasterize a layer with the GPU without having to involve the main thread at all in the process.