Blink/Chrome GPU Rasterization: Phase 1

humper@, skaslev@, bsalomon@, nduca@, ernstm@

Status: this is shipped.

Our plan is to ship hardware-accelerated rasterization first on Android as quickly as we can, ideally early 2014. Only some pages will trigger GPU rasterization, using our GPU triggering heuristics. In GPU mode, PictureLayerImpls will be viewport-wide tiled and rasterized into textures **Part 1**: Blink Rendering Basics
The source code for the Blink rendering engine is vast, complex, and somewhat scarcely documented. In order to understand how GPU acceleration works in Chrome it's important to first understand the basic building blocks of how Blink renders pages.
In the impl thread. Our goal for phase 1 is a **minimum viable** product. There are many awesome followups that can be done, as discussed in the **phase 2 design doc**.

Status

To follow along:

- **Overall: http://crbug.com/316685**
- skbug.com: Area-GPUJank, Area=RasterVsGpu
- crbug.com: SlowRaster

The Problem

There is a family of Web content that needs to re-rasterize every frame. Especially in mobile web pages where screen space is at a premium, designers often show summaries of data that expand on touch. These are guaranteed to repaint themselves every frame, making them ideal candidates for an accelerated drawing mode. For specific examples, see the Silk Page Set.

Using the GPU to accelerate rasterization will have a number of benefits. In particular, web designers will be able to use rich HTML5 content with confidence that it will perform well on smaller hardware such as mobile phones. Furthermore, using the GPU on a mobile device to render frequently-updating portions of the screen should be much more gentle on the battery.

Background

Currently, the compositor divides a web page into (possibly overlapping) rectangular layers, which are then implemented in cc/ as PictureLayer and PictureLayerImpls.

When a region of the screen needs to be drawn, either due to an initial exposure or in response to some animation, the Skia commands needed to draw a layer are recorded into an SkPicture, and that picture is later used on the compositor thread to rasterize the tile by playing it back into a software canvas. A pile data structure is used to handle layers that have partial updates, e.g. a small change in a bigger layer.

This architecture is very flexible, but it underperforms when regions of the web page change frequently:

- http://jsfiddle.net/humper/yEX8u/3/: a div that grows and changes layout every frame. Though the layout is trivial, rasterizing the 2 second animation with software creates ~44MB worth of data, that even with our best upload system has to be copied 2 times to the GPU before being blitted to the screen. GPU could do this with probably 10kb worth of GL commands.
- http://jsfiddle.net/vBQHH/3/ and http://jsfiddle.net/vBQHH/4/ A jpeg with decoration borders & drop that expands covering a full Nexus 10 screen is 16mb of texture creations & uploads. The GPU commands for this are trivial, and RGB->YUV conversion could run GPU-side saving 30% of image-related costs.
- http://jsfiddle.net/cK? Thanks for the feedback, its great to discuss these kinds of things.B9D/5/ A polygonal mask that changes every frame to implement a wipe in effect

Proposed Solution

Allow layers that meet certain criteria (see below) to use the GPU (Skia Ganesh) to directly render their contents into the framebuffer.

We will support two modes of rasterization:

1. The software mode, where tiles are created and rasterized by Sw Skia
2. A tiled ganesh mode, where tiles are created and rasterized by Ganesh.

There was some early work done by skaslev@ and vangelis@ here that implements “tiled ganesh mode” as above, but using a new layer type and without using the Tile Manager. The lessons learned from that work have been folded into this effort.

An aside on Direct Ganesh Mode

There was previously also a “direct mode” that worked by having Ganesh draw layers direct to the render target. Direct mode has some nice properties, but does not work well with übercompositor. To keep us focused on a V1 of this system, we have moved direct mode to the phase 2 design doc.

Pixel Compatibility between Ganesh and Software Skia

The skia team has committed to making Ganesh pixel compatible with software skia, but there are numerous subtleties that can arise in corner cases around subpixel details and epsilon amounts of errors. Tile-by-tile decisions are thus not possible. Making a layer-by-layer decisions is possible, but in that case, rapid toggling between sw and ganesh might make the epsilon values more visible. The approach most-free of perceptual issues is to pick one mode for the full page and stick with it.

Current issues: skia project Area=RasterVsGpu

Why Picture Recordings are Still Useful

There is overhead associated in the impl-side painting architecture related to recording and maintaining pictures. Intuitively, recording commands for rasterization only to throw them away and record again on the very next frame is inefficient. So why not kill this step?

We believe that keeping the picture is the right solution, even in light of this concern. Several reasons for this:

1. Picture Recording can be sped up hugely by storing pictures for individual RenderObjects: crbug/313885

2. When a layer stops changing, caching to get into a lower-power state becomes harder: either Ganesh needs to detect how to cache, or you have to bake to texture and reimplement impl side painting.
3. When the main thread becomes unresponsive, and a new part of the page is exposed, a picture lets you show that part of the page instead of checkerboarding/hanging.
4. In some early deferred canvas experiments, a 10% perf win was observed from using the intermediate picture. We speculate this was due to caching effects (code locality).

5. Thus, our position is that we should add Ganesh to impl side painting. It preserves all the nice properties of our existing approach but lets us improve the use cases cited earlier.

We will, in parallel with this effort, try to speed up picture recording significantly so that it is less of an issue, even when the world is constantly changing.

Triggering

We want to launch GPU rasterization before we have it performing as-fast-or-faster than all web content. The web is full of corner cases and non-mobile content, and our first order goal is to make mobile web sites have access to the latest and greatest rasterization capabilities. Thus, in the short term, pages with this meta tag will trigger GPU rasterization:

```
<meta name="viewport"
 content="width=device-width, minimum-scale=1.0,
 initial-scale=1.0, user-scalable=yes">
```

In the medium term, and especially after we launch V1, we want to make all pages get GPU rasterized, not just ones with this viewport.

Our detailed plans on this front, as well as supporting rationale, are reflected in the GPU rasterization Triggering design doc.

Übercomp, Where to Run Ganesh/Sending pictures

There are a variety of interesting options for how to mix Ganesh with an ubercomp. During the creation of this design doc, we explored a large number of possible implementations. The current plan is to always bake to tiles:

- Pro:
  - Simple.
    - Extra GPU work on animation-heavy workloads May actually be a win over sw rast for many cases: 0 copy upload!
- Cons:
  - May need to replay the same SkPicture on many tiles
    - Mitigated with viewport-wide tiles

    - Some complexities with mixing uploads and FBO rendering into the same tiles, per @epenner

Benchmarking

Telemetry Benchmarks

We use the smoothness benchmark and the repaint benchmark to track our progress. Smoothness benchmark accurately measures end-to-end rendering performance. We use it to track improvements on app-like content (key_silk_cases) and to make sure we don't regress traditional scrolling pages (key_mobile_sites). The repaint benchmark emphasizes the cost recording and rasterization by invalidating the viewport every frame. This allows us to roughly measure the relative cost of GPU rasterization to CPU rasterization on real pages (key_mobile_sites).

A typical run to compare GPU vs. CPU rasterization on an Android device requires two measurements, e.g. for smoothness with the key_silk_cases:

```
tools/perf/run_measurement smoothness tools/perf/page_sets/key_silk_cases.py
--page-repeat=5 --reset-results
```

```
tools/perf/run_measurement smoothness tools/perf/page_sets/key_silk_cases.py
--page-repeat=5 --extra-browser-args="--enable-gpu-rasterization"
```

The first measurement creates the baseline CPU rasterization numbers. The second one is the GPU rasterization run. `tools/perf/results.html` will then contain data for the two runs and the speed-up/slow-down. Note that `--reset-results` deletes an existing `results.html` file.

If the page doesn't get triggered for GPU rasterization, in can be forced with `--extra-browser-args="--force-gpu-rasterization"`.

On platforms, where impl-side painting is not enabled by default, the following addition browser args need to be added: "`--enable-impl-side-painting --enable-threaded-compositing`"

Skia Benchmarks

To measure Skia's GPU vs. CPU rasterization performance outside of Chrome, the Skia SampleApp is a very useful tool. The ability to zoom in on parts of the page and measure rasterization time can help identify content that is expensive to rasterize. Skia Debugger can then be used to pinpoint costly draw commands.

For batch runs, Skia's bench_pictures tools is the right tools to compare the cost of GPU vs. CPU rasterization.

Tracing

For detailed diagnosis of performance issues we will use tracing.