GPU Accelerated Compositing in Chrome

Vangelis Kokkevis, with a little help from Tom Wiltzius & the Chrome GPU team

updated May 2012

Summary

This document provides background and details on the implementation of hardware-accelerated compositing in Chrome.

Introduction

Traditionally, web browsers relied entirely on the CPU to render web page content. With capable GPUs becoming an integral part of even the smallest of devices and with rich media such as video and 3D graphics playing an increasingly important role to the web experience, attention has turned on finding ways to make more effective utilization of the underlying hardware to achieve better performance and power savings. There's clear indication that getting the GPU directly involved with compositing the contents of a web page can result in very significant speedups. The largest gains are to be had from eliminating unnecessary (and very slow) copies of large data, especially copies from video memory to system memory. The most obvious candidates for such optimizations are the <video> element when using a hardware decoder and the WebGL canvas, both of which can generate their results in areas of memory that that CPU doesn't have fast access to.

Delegating compositing of the page layers to the GPU provides other benefits as well. In most cases, the GPU can achieve far better efficiency than the CPU (both in terms of speed and power draw) in drawing and compositing operations that involve large numbers of pixels as the hardware is designed specifically for these types of workloads. Utilizing the GPU for these operations also provides parallelism between the CPU and GPU, which can operate at the same time to create an efficient graphics pipeline.

Part One: WebKit Rendering Basics & The Software Path

The source code for the WebKit rendering engine is vast and 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 WebKit renders pages. We'll start with a basic overview of how things worked before the GPU became involved and then revise that understanding to explain how the GPU changes things.

Nodes and the DOM tree

In WebKit, the contents of a web page are internally stored as a tree of Node objects called the DOM tree. Each HTML element on a page as well as text that occurs between elements is associated with a Node. The top level Node of the DOM tree is always a Document Node.

From Nodes to RenderObjects

Each node in the DOM tree that produces visual output has a corresponding *RenderObject*. RenderObject's are stored in a parallel tree structure, called the *Render Tree.* A RenderObject knows how to present (paint) the contents of the Node on a display surface. It does so by issuing the necessary draw calls to a *GraphicsContext.* A GraphicsContext is ultimately responsible for writing the pixels into a bitmap that gets displayed to the screen. In Chrome, the GraphicsContext wraps Skia, our 2D drawing library, and most GraphicsContext calls become calls to an SkCanvas or SkPlatformCanvas (see this document for more on how Chrome uses Skia).

In the software path, there's one GraphicsContext for the entire page, and all RenderObjects paint into this single shared GraphicsContext.

From RenderObjects to RenderLayers

Each RenderObject is associated with a *RenderLayer* either directly or indirectly via an ancestor RenderObject.

RenderObjects that share the same coordinate space (e.g. are affected by the same CSS transform) typically belong to the same RenderLayer. RenderLayers exist so that the elements of the page are composited in the correct order to properly display overlapping content, semi-transparent elements, etc. There's a number of conditions that will trigger the creation of a new RenderLayer for a particular RenderObject, as defined in RenderBoxModelObject::requiresLayer() and overwritten for some derived classes. In general a RenderObject warrants the creation of a RenderLayer if one of the following holds:

- It's the root object for the page
- It has explicit CSS position properties (relative, absolute or a transform)
- It is transparent
- Has overflow, an alpha mask or reflection
- Has a CSS filter
- Corresponds to <canvas> element that has a 3D (WebGL) context or an accelerated 2D context
- Corresponds to a <video> element

Notice that there isn't a one-to-one correspondence between RenderObjects and RenderLayers. A particular RenderObject is associated either with the RenderLayer that was created for it, if there is one, or with the RenderLayer of the first ancestor that has one.

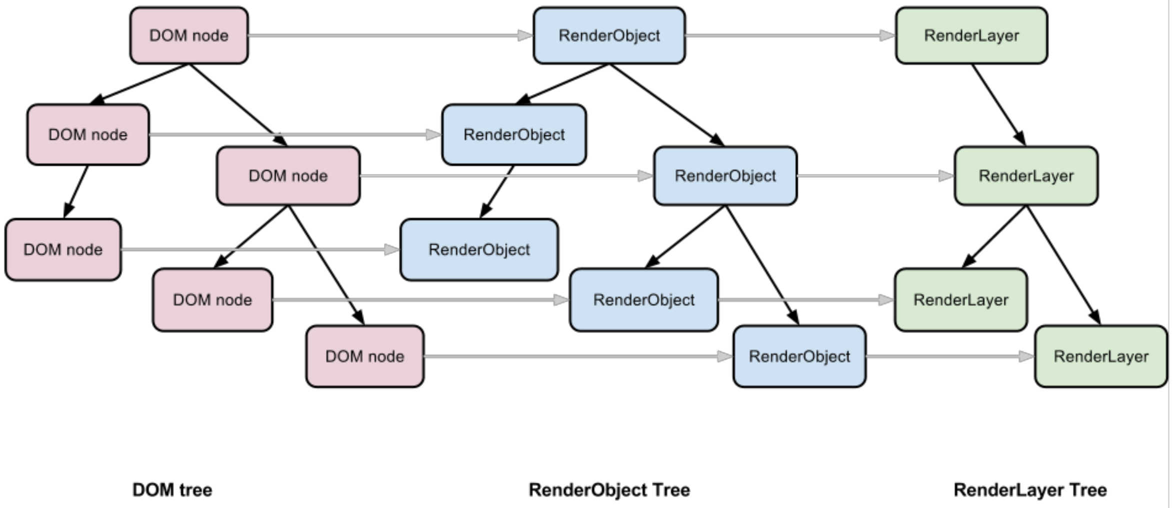
RenderLayers form a tree hierarchy as well. The root node is the RenderLayer corresponding to the root element in the page and the descendants of every node are layers visually contained within the parent layer. The children of each RenderLayer are kept into two sorted lists both sorted in ascending order, the *negZOrderList* containing child layers with negative z-indices (and hence layers that go below the current layer) and the *posZOrderList* contain child layers with positive z-indices (layers that go above the current layer).

Putting it Together: Many Trees

In summary, there are conceptually three parallel tree structures in place that serve slightly different purposes for rendering:

- The ***DOM tree,*** which is our fundamental retained model

- The ***RenderObject tree,*** which has a 1:1 mapping to the DOM tree's visible nodes. RenderObjects know how to paint their corresponding DOM nodes.
- The ***RenderLayer tree,*** made up of *RenderLayers* that map to a RenderObject on the RenderObject tree. The mapping is many-to-one, as each RenderObject is either associated with its own RenderLayer or the RenderLayer of its first ancestor that has one. The RenderLayer tree preserves z-ordering amongst layers.



The Software Rendering Path
WebKit fundamentally renders a web page by traversing the RenderLayer hierarchy starting from the root layer. The WebKit codebase contains two distinct code paths for rendering the contents of a page, the *software path* and *hardware accelerated path.* The software path is the traditional model.

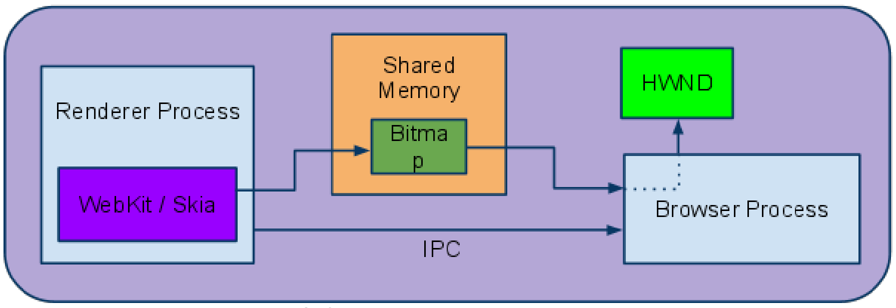
In the software path, the page is rendered by sequentially painting all the RenderLayers, from back to front. The RenderLayer hierarchy is traversed recursively starting from the root and the bulk of the work is done in RenderLayer::paintLayer() which performs the following basic steps (the list of steps is simplified here for clarity):

a. Determines whether the layer intersects the damage rect for an early out.
b. Recursively paints the layers below this one by calling paintLayer() for the layers in the negZOrderList.
c. Asks RenderObjects associated with this RenderLayer to paint themselves.
d. This is done by recursing down the RenderObject tree starting with the RenderObject which created the layer. Traversal stops whenever a RenderObject associated with a different RenderLayer is found.
e. Recursively paints the layers above this one by calling paintLayer() for the layers in the posZOrderList.

In this mode RenderObjects paint themselves into the destination bitmap by issuing draw calls into a single shared GraphicsContext (implemented in Chrome via Skia).

Note that the GraphicsContext itself has no concept of layers, but to make drawing of semi-transparent layers correct there's a caveat: semi-transparent RenderLayers call GraphicsContext::beginTransparencyLayer() before asking their RenderObjects to draw. In the Skia implementation, the call to beginTransparencyLayer() causes all subsequent draw calls to render in a separate bitmap, which gets composited with the original one when the layer drawing is complete and a matching call to endTransparencyLayer() is made to the GraphicsLayer after all the semi-transparent RenderLayer's RenderObjects have drawn.

From WebKit to the Screen



Software Rendering Architecture

Once all the RenderLayers are done painting into the shared bitmap the bitmap still needs to make it onto the screen. In Chrome, the bitmap resides in shared memory and control of it is passed to the Browser process via IPC. The Browser process is then responsible for drawing that bitmap in the appropriate tab / window via the OS's windowing APIs (using e.g. the relevant HWND on Windows).

Part 2: Hardware Basics

Recall that the WebKit codebase contains two distinct code paths for rendering the contents of a page, the software path and hardware accelerated path. Now that you understand the software rendering path, we can explain the hardware path in terms of how it differs from the software path.

As the name suggests, the hardware accelerated path is there to make use of GPU acceleration for compositing some of the RenderLayer contents. Code for it lives behind the ACCELERATED_COMPOSITING compile-time flag.

Chrome currently uses the hardware accelerated path when at least one of the page's RenderLayer's requires hardware acceleration, or when the --forced-compositing-mode flag is turned on. This flag is currently on by default in Chrome on Android and ChromeOS. Eventually this will also be the case for Chrome on other platforms. Safari on the Mac (and most likely iOS) follows the hardware accelerated path and makes heavy use of Apple's proprietary CoreAnimation API.

Introducing the Compositor

In the hardware accelerated path, some (but not all) of the RenderLayers get their own *backing surface* (layers with their own backing surfaces are called *compositing layers)* into which they paint instead of drawing directly into the common bitmap for the page. A subsequent compositing pass composites all the backing surfaces onto the destination bitmap. We still start with the RenderLayer tree and end up with a single bitmap, but this two-phase approach allows the compositor to perform additional work on a per-compositing-layer basis.

For instance, the compositor is responsible for applying the necessary transformations (as specified by the layer's CSS transform properties) to each compositing layer's bitmap before compositing it. Further, since painting of the layers is decoupled from compositing, invalidating one of these layers only results in repainting the contents of that layer alone and recompositing.

In contrast, with the software path, invalidating any layer requires repainting all layers (at least the overlapping portions of them) below and above it which unnecessarily taxes the CPU.

More Trees: From RenderLayers to GraphicsLayers

Recall that in the software path there was a single GraphicsContext for the entire page. With accelerated compositing, we need a GraphicsContext for each compositing layer so that each layer can draw into a separate bitmap. Recall further that we conceptually already have a set of parallel tree structures, each more sparse than the last and responsible for a subtree of the previous: the DOM tree, the RenderObject tree, and the RenderLayer tree. With the introduction of compositing, we add an additional conceptual tree: the *GraphicsLayer* tree. Each RenderLayer either has its own GraphicsLayer (if it is a compositing layer) or uses the GraphicsLayer of its first ancestor that has one. This is similar to RenderObject's relationship with RenderLayers. Each GraphicsLayer has a GraphicsContext for the associated RenderLayers to draw into.

While in theory every single RenderLayer could paint itself into a separate backing surface to avoid unnecessary repaints, in practice this could be quite wasteful in terms of memory (vram especially). In the current WebKit implementation, the following conditions are some of those that cause a RenderLayer to get its own compositing layer (see the CompositingReasons enum in RenderLayer.h for a longer list):

- Layer has 3D or perspective transform CSS properties
- Layer is used by <video> element using accelerated video decoding
- Layer is used by a <canvas> element with a 3D context or accelerated 2D context
- Layer is used for a composited plugin
- Layer uses a CSS animation for its opacity or uses an animated webkit transform
- Layer uses accelerated CSS filters
- Layer with a composited descendant has information that needs to be in the composited layer tree, such as a clip or reflection
- Layer has a sibling with a lower z-index which has a compositing layer (in other words the layer is rendered on top of a composited layer)

Significantly, this means that pages with composited RenderLayers will always render via the compositor. Other pages may or may not, depending on the status of the --forced-compositing-mode flag.

The Code

Code related to the compositor lives inside WebCore, behind the USE(ACCELERATED_COMPOSITING) guards. Part of the code is shared among all platforms and part of it is Chrome-specific. Thankfully, the WebKit code is structured such that implementing the compositor for Chrome required no changes to the core WebKit codebase and all the Chrome-specific code are provided in platform-specific source files that live in platform/graphics/chromium. We did a similar thing with GraphicsContextSkia, our Skia-based implementation of GraphicsContext.

Whither the GPU?

So how does the GPU come into play? With the addition of the accelerated compositor, in order to eliminate costly memory transfers, the final rendering of the browser's tab area is handled directly by the GPU. This is a significant departure from the current model in which the Renderer process passes (via IPC and shared memory) over a bitmap with the page's contents to the Browser process for display (see "From WebKit to the Screen" above).

In the h/w accelerated architecture, compositing of the h/w accelerated layers (i.e. composited layers, those with backing surfaces) with the rest of the page contents happens on the GPU via calls to the platform specific 3D APIs (GL / D3D). The code ultimately responsible for making these calls is encapsulated in a library running inside the Renderer process, the *Compositor.* The Compositor library is essentially using the GPU to composite rectangular areas of the page (i.e. all those compositing layers) into a single bitmap, which is the final page image.

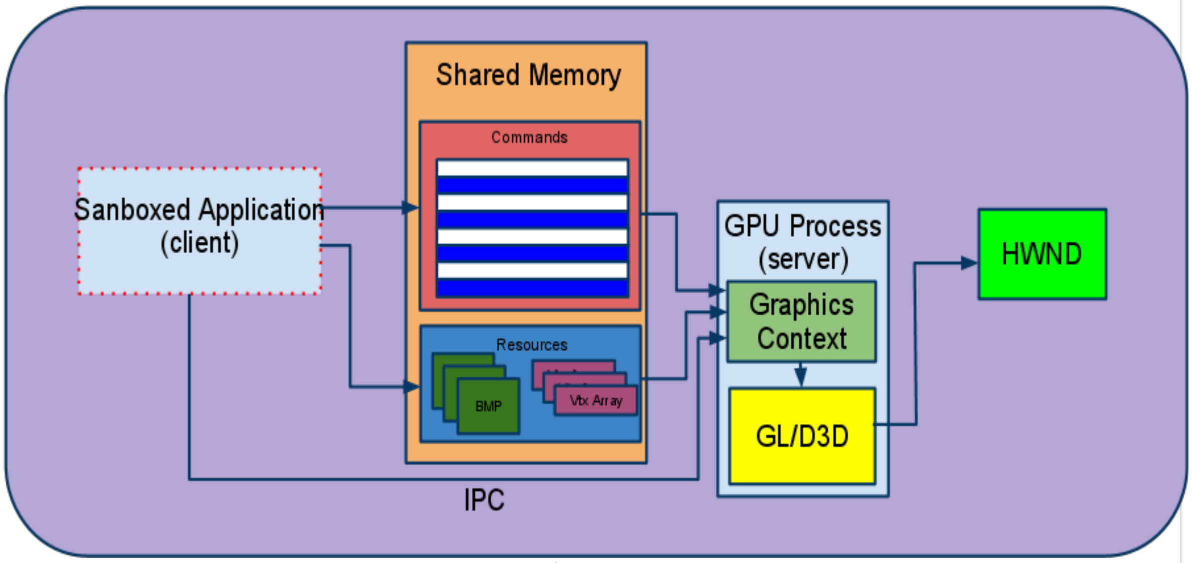
Architectural Interlude: The GPU Process

Before we go any further exploring the GPU commands the compositor generates, its important to understand how the renderer process issues any commands to the GPU at all. In Chrome's multi-process model, we have a dedicated process for this task: the *GPU process.* The GPU process exists primarily for security reasons.

Restricted by its sandbox, the Renderer process (where WebKit and the compositor live) cannot directly issue calls to the 3D APIs provided by the OS (we use Direct3D on Windows, OpenGL everywhere else). For that reason we use a separate process to do the rendering. We call this process the GPU Process. The GPU process is specifically designed to provide access to the system's 3D

APIs from within the Renderer sandbox or the even more restrictive Native Client "jail". It works via a client-server model as follows:

- The *client* (code running in the Renderer or within a NaCl module), instead of issuing calls directly to the system APIs, serializes them and puts them in a ring buffer (the *command buffer)* residing in memory shared between itself and the server process.
- The *server* (GPU process running in a less restrictive sandbox that allows access to the platform's 3D APIs) picks up the serialized commands from shared memory, parses them and executes the appropriate graphics calls, outputting directly to a window.



The GPU Process

The commands accepted by the GPU process are patterned closely after the GL ES 2.0 API (for example there's a command corresponding to glClear, one to glDrawArrays, etc). Since most GL calls don't have return values, the client and server can work mostly asynchronously which keeps the performance overhead fairly low. Any necessary synchronization between the client and the server, such as the client notifying the server that there's additional work to be done, is handled via an IPC mechanism. It's also worth noting that in addition to providing storage for the command buffer, shared memory is used for passing larger resources such as bitmaps for textures, vertex arrays, etc between the client and the server.

From the client's perspective, an application has the option to either write commands directly into the command buffer or use the GL ES 2.0 API via a client side library that we provide which handles the serialization behind the scenes. Both the compositor and WebGL currently use the GL ES client side library for convenience. On the server side, commands received via the command buffer are converted to calls into either desktop OpenGL (on Mac, Linux/ChromeOS, and Android) or Direct3D (on Windows) via ANGLE.

Currently Chrome uses a single GPU process per browser instance, serving requests from all the renderer processes and any plugin processes. The GPU process, while single threaded, can multiplex between multiple command buffers, each one of which is associated with its own rendering context.

The GPU process architecture offers several benefits including:

- Security: The bulk of the rendering logic remains in the sandboxed Renderer process.
- Robustness: A GPU process crash (e.g. due to faulty drivers) doesn't bring down the browser.
- Uniformity: Standardizing on OpenGL ES 2.0 as the rendering API for the browser regardless of the platform allows for a single, easier to maintain codebase across all OS ports of Chrome.
- Parallelism: The Renderer can quickly issue commands into the command buffer and go back to CPU-intensive rendering activities, leaving the GPU process to process them. We can make good use of both processes on multi-core machines as well as the GPU and CPU simultaneously thanks to this pipeline.

With that explanation out of the way, we can go back to explaining how the GL commands and resources are generated inside of the renderer process by the compositor.

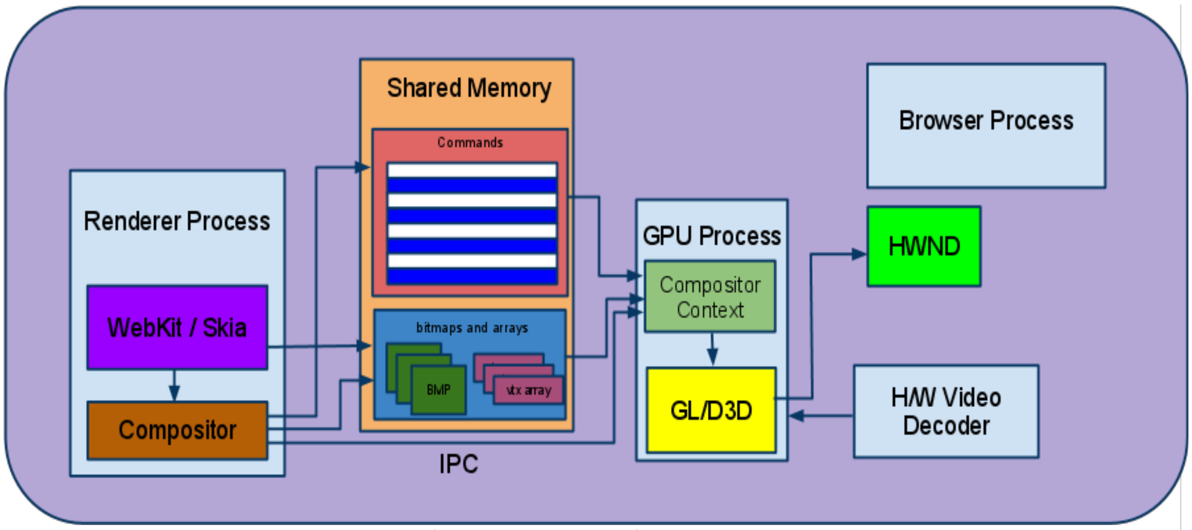
Part 3: Rendering with the Compositor in the Hardware Path

The compositor is implemented on top of the GL ES 2.0 client library which proxies the graphics calls to the GPU process (using the method explained above).

When a page renders via the compositor, all of its pixels are drawn directly onto the window via the GPU process. The compositor maintains a hierarchy of GraphicsLayers which is constructed by traversing the RenderLayer tree and updated as the page changes. With the exception of WebGL and video layers, the contents of each of the GraphicsLayers are first drawn into a system memory bitmap (just as was the case in the software path): each RenderLayer asks all of its RenderObjects to paint themselves into the GraphicsLayer’s GraphicsContext, which is backed by a bitmap in shared system memory. This bitmap is then passed to the GPU process (using the resource transfer machinery explained above in the GPU Process section), and then the GPU process uploads the bitmap to the GPU as a texture. The compositor keeps track of which GraphicsLayers have changed since the last time they were drawn and only updates the textures as needed.

Once all the textures are uploaded to the GPU, rendering the contents of a page is simply a matter of doing a depth first traversal of the GraphicsLayer hierarchy and issuing a GL command to draw a texture quad for each layer with the previously-uploaded texture. A texture quad is simply a 4-gon (i.e. rectangle) on the screen that’s filled with the given texture (in our case, the relevant GraphicsLayer’s contents).

Note that doing the traversal depth-first ensures proper z-ordering of GraphicsLayers, and the z- ordering of each GraphicsLayer’s RenderLayers is guaranteed earlier when the RenderObjects are rasterized into the texture.



Compositing with the GPU process

The Code

The bulk of the Chromium implementation for the compositor lives in WebCore's platform/graphics/chromium directory. The compositing logic is mostly in LayerRendererChromium.cpp and the implementations of the various composited layer types are in {Content|Video|Image} LayerChromium.cpp files.

Part 4: Optimize! Smooth Rendering

Now we know roughly how to draw a page using the compositor: the page is divided up into layers, layers are rasterized into textures, textures are uploaded to the GPU, and the compositor tells the GPU to put all the textures together into the final screen image.

Next comes understanding how we manage to do all of this 60 times a second so animation, scrolling, and other page interactions are smooth. To explain that, we need to introduce a few more concepts that amount to optimizations of the above rendering techniques.

Damage

Thus far we've explained only how to draw an entire web page. This is roughly how WebKit draws pages when they're first loaded. But typically during user interaction with the page only parts of it change. Enter the damage rect.

When the webpage visually changes (because e.g. JS has modified CSS styles, a CSS animation is running, or the user has scrolled the viewport) WebKit keeps track of what parts of the screen need to be updated. The result is a *damage rectangle* whose coordinates indicate the part of the page that needs to be repainted. When painting, we traverse the RenderLayer tree and only repaint the parts of each RenderLayer that intersect with the damage rect, skipping the layer entirely if it doesn't overlap with the damage rect. This prevents us from having to repaint the entire page every time any part of it changes, an obvious performance win.

Tiling

In software, recall that we're painting into a single shared bitmap in system memory, so we can modify any subregion of this bitmap easily. In hardware it's more difficult: recall that after rasterizing the contents of a RenderLayer into a bitmap, that bitmap is uploaded to the GPU as a texture. RenderLayers can be arbitrarily large: they can be up to the size of the entire page, including areas that aren't normally visible (e.g. if a arbitrarily-large <div> has the overflow:scroll CSS property set). This is problematic because we can't make textures arbitrarily large: doing so would exceed the GPU's maximum texture size, or at least take up a lot of VRAM. It almost might take a long time to initially paint and upload this giant texture.

One solution to the texture-size problem would be to only paint and upload what's currently visible. This means the texture never needs to be larger than the viewport, getting around the issues outlined above. However, this introduces a new problem: every time the viewport changes (e.g. because of a scroll) we would need to repaint and reupload this texture. This is too slow.

The solution to this problem is *tiling*. Each layer is split up into tiles (currently of a fixed size, 256x256 pixels). We determine which parts of the layer are needed on the GPU and only paint + upload those tiles. We can save tiles in a number of places: we only need to upload those that are currently visible and intersect the damage rect (i.e. are *invalidated* on the GPU because they're stale). We can further optimize by tracking any occluded tiles (those that aren't visible because some combination of opaque layers is covering them entirely) and avoid uploading these occluded tiles. Once we have all updated tiles uploaded the compositor can composite them all together into the final screen image on the GPU.

Tiling texture gives us one final benefit: *texture streaming*. When a large GraphicsLayer does need to be uploaded to the GPU, it can be done several tiles at a time to conserve memory bandwidth. This prevents the GPU from stalling during a long texture upload; the upload can be broken up over the course of several frames.

Case Study: Scrolling in Software & Hardware

In software, we do shift-and-backfill scrolling: as the page moves up or down we shift the previous final screen image bitmap up however many pixels we've scrolled, and in the gap left over we repaint. This repainting works the same way any repainting in software does: we have a damage rect that makes up the scrolled margin and ask WebKit to repaint the page clipped to that damage rect. When this is done we have a complete screen bitmap again and can draw it on the screen.

Scrolling in hardware works very differently, and highlights the compositor's power. Because during a scroll the page contents typically aren't changing, only the position of the viewport, we can make use of the fact that the updated layer contents may already be on the GPU.

In the simple case, if the scroll is only a small amount (less than the size of a tile), we may not need any new tiles to represent the viewport contents. We can then skip rasterization and uploading entirely and simply reposition the tile textures by drawing texture quads with the same textures but different coordinates during the final composition step. This is extremely cheap on the GPU, and since we're not painting at all it's light on CPU resources, too.

When a scroll moves a previously unvisible-tile into view, then we will have to rasterize and upload those tiles, but no other tiles in the GraphicsLayer are affected. We can also do this ahead of time by rasterizing/uploading tiles beyond the current viewport. When we paint tiles that aren't yet visible but we expect will be soon (for instance, because a large scroll is taking place) we call it *tile prepainting*.

The Threaded Compositor

Driving rendering with the compositor becomes even more powerful when moved to its own thread. In software, WebKit rendering runs on the Renderer process's main thread along with other activities like most JavaScript execution. This is a significant limitation, as JavaScript can't run at the same time as rendering. Given a limited frame budget, contention in the main thread can cause slowdowns leading to lower, variable frame rates.

The threaded compositor helps alleviate this contention. Threaded compositing works the same way that compositing on the main thread does, but as the name implies it occurs on a separate thread and operates on a copy of the current rendering state.

There are two sides to the threaded compositor: the main-thread side, and the (poorly named) “impl” side, which is the compositor thread’s half. The main thread has a CCLayerTreeHost, which is its copy of the layer tree, and the impl thread has a CCLayerTreeHostImpl, which is *its* copy of the layer tree. Similar naming conventions are followed throughout.

Conceptually these two layer trees are completely separate, and the compositor (impl) thread’s copy can be used to produce frames without any interaction with the main thread. This means the main thread can be busy running expensive JavaScript and the compositor can still be happily translating textures on the GPU without interruption. This allows us to get long, fluid, scrolls updated at 60 FPS even when the main thread is busy – the compositor can tick out frames reliably, which is impossible when unknown interactions on the main thread might get in the way of ticks. In order to achieve this, the flow of input events is slightly modified with the threaded compositor – input events are received in the Browser process as normal, but are forwarded to the compositor first and from there to the Renderer main thread. This lets the compositor handle interactions like scrolls without being dependent on the Renderer main thread.

Keep in mind that while the compositor is generating frames with its copy of the the layer tree this layer tree isn’t changing at all (e.g. styles updated by JavaScript are not applied). Periodic synchronization is required to keep the compositor thread’s copy of the layer tree up-to-date. This synchronization requires blocking both the main thread and the compositor thread. The CCScheduler class decides when this synchronization will occur. Importantly, the only operations that require blocking both the main and compositor threads are those during a *commit* from the main thread to the compositor thread: tree synchronization and copies of main-thread-side-rasterized layer bitmaps. Everything else can happen asynchronously.

Additionally, it’s important to note that the compositor can only generate frames on its own when the part of the page that needs to be displayed has already been rasterized. If the main thread is busy (e.g. running JS) and the user scrolls to a part of the page that has not yet been rasterized (or has been rasterized but those textures have not yet been committed to the compositor thread) we have nothing to display, so we *checkerboard* or just draw white.

In addition to scrolls, the compositor knows how to drive a number of other page updates on its own. Anything that can be done without needing to ask WebKit to repaint anything can be handled on the compositor thread without interruption. Thus far CSS animations are the only other major compositor-driven page update.

Significantly more information about how the compositor thread and main thread interact can be found in the Compositor Thread Architecture document.

An interesting consequence of having composition separated from painting is that we can better define frame boundaries. The compositor obeys vsync – that is, the compositor will only generate new frames as often as they can be displayed by the monitor, and only issue a SwapBuffers call between monitor refreshes. This prevents visual tearing on the screen (which would occur if the front and back buffers were swapped halfway through a monitor's refresh), and caps our displayed framerate to whatever the monitor's refresh rate is (typically 60 Hz).

Additionally, because our goal is high framerate with low variance we can potentially generate cheap frames by compositing older states of the 'world'. This means that we actually have multiple notions of a framerate: how often we're recompositing and drawing on the screen vs. how often we're refreshing the screen contents from our fundamental model (i.e. the DOM) on the main thread. In cases like checkerboarded fling scrolls, these two numbers may be different.

Architecture Interlude Two: Multiple Sources of GL Commands in the Renderer

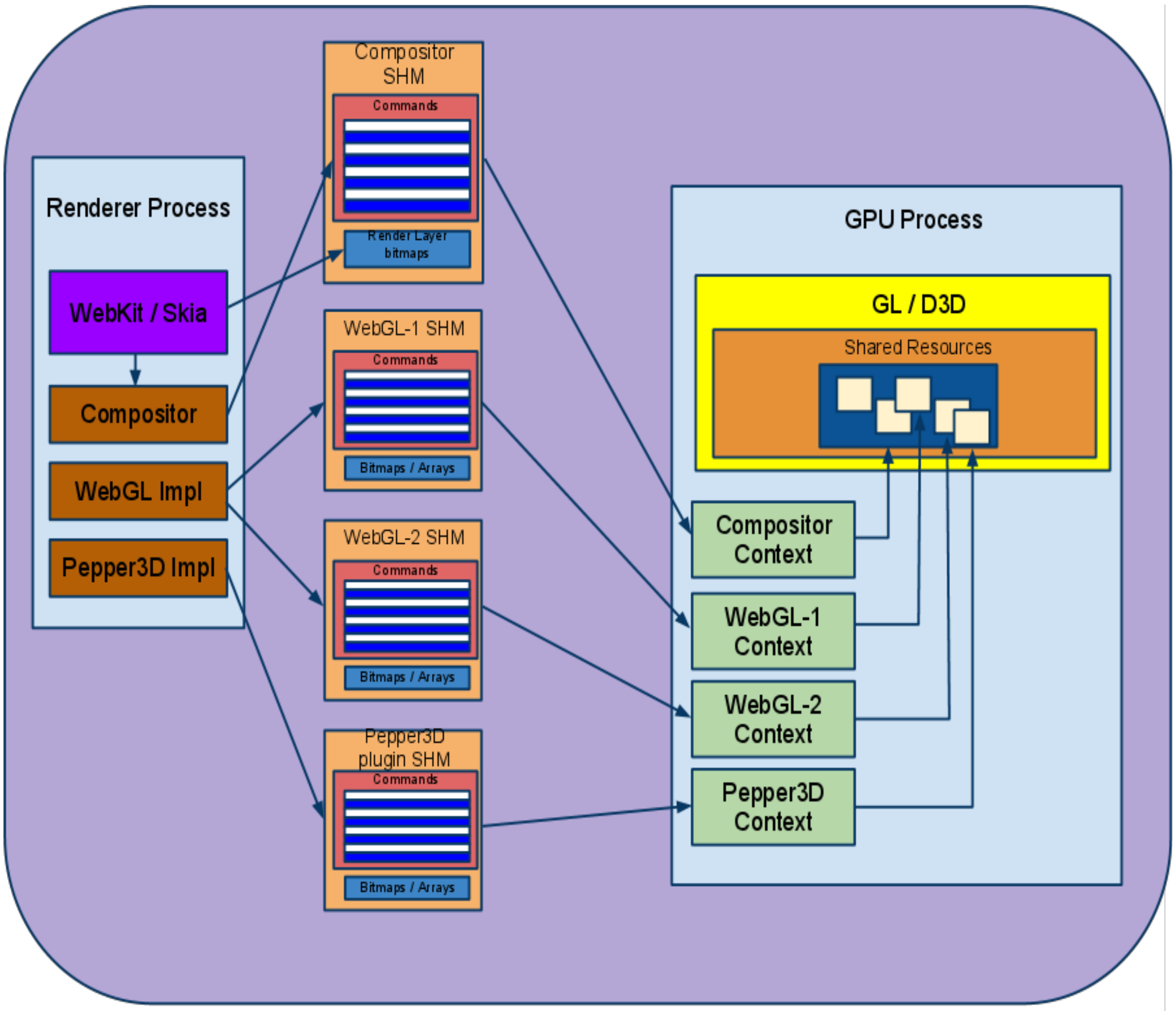
Thus far we've explained how the GPU gets used by the compositor, but Chrome makes use of the GPU for various other operations as well. Chrome achieves this by having multiple GPU-process-side graphics contexts for each renderer. Non-compositor sources of GL in the renderer include accelerated Canvas2D and WebGL.

At a minimum, when doing h/w accelerated compositing, the GPU process handles a single graphics context, the one used by the compositor. However, in the presence of GPU- accelerated content in the page (such as WebGL or Pepper3D plugin instances), the GPU process needs to be able to juggle multiple graphics contexts, each associated with its own Command Buffer, Shared Memory, IPC channel and a GL context.

Before we go any further, some term disambiguation: it’s confusing that we refer to the GPU-process-side contexts for content regions as “graphics contexts,” because these are *different* than the GraphicsContexts used in the renderer. The GraphicsContexts used in the renderer are backed by Skia, our 2D drawing library, which usually draws into a bitmap (although not always, but we’ll get to that). The graphics contexts in the GPU process are associated with a GL context (and the necessary command buffer, shared memory, IPC channel, etc to talk to that GL context from the renderer). The compositor has a GPU-process-side graphics context. Most GraphicsLayers just paint into a Renderer-process-side GraphicsContext and the bitmap backing up that GraphicsContext gets uploaded to the GPU via the compositor’s GPU-process-side graphics context. However, some special types of GraphicsLayers create content directly on the GPU, rather than via a renderer-process-side GraphicsContext. These special types of GraphicsLayers get their own GPU-process-side-graphics contexts, too.

Composition of GraphicsLayers whose contents are created directly on GPU works as follows: instead of rendering straight into the backbuffer, they render into a texture (using a Frame Buffer Object) that the compositor context grabs and uses when rendering that GraphicsLayer. It's important to note that in order for the compositor's GL context to have access to a texture generated by an offscreen GL context (i.e. the GL contexts used for the other GraphicsLayers’ FBOs), all GL contexts used by the GPU process are created such that they share resources.

The resulting architecture looks like this:



Handling multiple contexts

Part Five: Using the GPU Beyond Compositing

Sources of GL in the Renderer: WebGL

WebGL is a very basic source of GL in the renderer process. WebGL closely resembles OpenGL ES 2.0, the GL dialect we use throughout Chrome, so there are relatively few translations that need to happen before these get passed to the GPU process.

More information on how GL commands are passed through the command buffer to the GPU process, validated for security, etc. can be found in the command buffer documentation.

Sources of GL in the Renderer: Ganesh

Accelerated Canvas2D is another source of commands in the Renderer. Canvas2D is a 2D drawing API and modifications to the Canvas2D element only come from JavaScript. As such rather than rendering Canvas elements like the rest of WebKit's RenderObjects Canvas2D elements are more like wrappers to Skia, our 2D drawing library.

Canvas2D rendering in software uses Skia to issue drawing commands into a GraphicsContext (i.e. SkCanvas, image buffer, or bitmap), and then copies the contents of this bitmap into the single shared bitmap WebKit is using for the page's content (recall that there's a single GraphicsContext for WebKit in software mode and that this GraphicsContext wraps Skia and behind the scenes Skia is pointing at a single bitmap in shared memory).

In hardware, Canvas2D rendering goes down a very different path. The canvas element gets its own compositing layer (that is, a RenderLayer with its own GraphicsLayer and GraphicsContext). The SkPicture instance behind the GraphicsContext is switched to have a different backend than normal – rather than being backed by a bitmap, it's backed by Ganesh, a GL backend for Skia. Ganesh, rather than rasterizing into a bitmap, generates equivalent GL commands. So as Canvas2D commands are issued from JavaScript they get translated into GraphicsContext API calls and from there to SkPicture recording calls. When we need to get the contents of the SkPicture (i.e. when we need the contents of that layer uploaded to the GPU) we play back the SkPicture as a stream of GL commands.

This stream of GL commands is shuttled through the command buffer to the GPU process and onto the GPU, where they're rendered into an FBO. When the compositor comes along to generate a frame, it grabs the contents of this FBO and uses it for the canvas element's GraphicsLayer.

Going all the Way: Accelerated Painting

With Ganesh, there's no theoretical reason we can't do *all* WebKit rendering on the GPU. Rather than rasterizing RenderLayers into bitmaps and uploading them to the GPU before compositing, we can back each RenderLayer's GraphicsContext with an instance of Skia-with-Ganesh. When in the painting phase of rendering, the RenderLayers can record SkPicture commands to represent the layer. The SkPicture can then be played back to rasterize the layer, generating GL commands via Ganesh and sending them straight to an FBO on the GPU. During the compositing phase of rendering, these FBOs can be composited just as Canvas2D or WebGL FBOs are rendered today.

While this should “just work” there are a lot of details to be worked out and this area of work is not yet completed.

Appendix One: Grafix 4 N00bs glossary

bitmap: a buffer of pixel values in memory (main memory or the GPU’s video RAM)
texture: a bitmap meant to be applied to a 3D model on the GPU
texture quad: a texture applied to a very simple model: a four-pointed polygon, e.g. a rectangle. Useful when all you want is to display the texture as a flat rectangular surface, potentially translated (either in 2D or 3D), which is exactly what we do when compositing.
painting: in our terms, the phase of rendering where RenderObjects make calls into the GraphicsContext API to make a visual representation of themselves
rasterization: in our terms, the phase of rendering where the bitmaps backing up RenderLayers are filled. This can occur immediately as GraphicsContext calls are by the RenderObjects, or it can occur later if we’re using SkPicture record for painting and SkPicture playback for rasterization.
compositing: in our terms, the phase of rendering that combines RenderLayer’s textures into a final screen image
drawing: in our terms, the phase of rendering that actually puts pixels onto the screen (i.e. puts the final screen image onto the screen).
backbuffer: when double-buffering, the screen buffer that’s rendered into, not the one that’s currently being displayed
frontbuffer: when double-buffering, the screen buffer that’s currently being displayed, not the one that’s currently being rendered into
swapbuffers: switching the front and back buffers
Frame Buffer Object: OpenGL term for a texture that can be rendered to off-screen as if it were a normal screen buffer (e.g. the backbuffer). Useful for us because we want to be able to render to textures and then composite these textures; with FBOs we can pretend to give e.g. WebGL its own frame and it need not worry about anything else going on on the page.
damage: the area of the screen that has been “dirtied” by user interaction or programmatic changes (e.g. JavaScript changing styles). This is the area of the screen that needs to be re-painted when updating.
retained mode: a rendering method where the graphics system maintains a complete model of the objects to be rendered. The web platform is retained in that the DOM is the model, and the platform (i.e. the browser) keeps track of the state of the DOM and its API (i.e. JavaScript’s access to the DOM) can be used to modify it or query the current state of it, but the browser can render from the model at any time without any instruction from JavaScript.
immediate mode: a rendering method where the graphics system doesn’t keep track of the overall scene state but rather immediately executes any commands given to it and forgets about them. To redraw the whole scene all commands need to be re-issued. Direct3D is immediate mode, as is Canvas2D.

Appendix Two: Relevant flags

If you are curious about the structure of the composited layers, use the ‘show composited layer borders’ flag. You can also try the ‘threaded compositing’, ‘threaded animation’. All available in about:flags.

As mentioned earlier, accelerated compositing in WebKit (and Chromium) kicks in only if certain types of content appear on the page. An easy trick to force a page to switch over to the compositor is to supply a -webkit-transform:translateZ(0) for an element in the page.

Appendix Three: Debugging Composited Layers

Using the --show-composited-layer-borders flag will display borders around layers, and uses colors to display information about the layers, or tiles within layers:

- Green - The border around the outside of a composited layer.
- Dark Blue - The border around the outside of a "render surface". Surfaces are textures used as intermediate targets while drawing the frame.
- Purple - The border around the outside of a surface's reflection.
- Cyan - The border around a tile within a tiled composited layer. Large composited layers are broken up into tiles to avoid using large textures.
- Red - The border around a composited layer, or a tile within one, for which the texture is not valid or present. Red can indicate a compositor bug, where the texture is lost, but typically indicates the compositor has reached its memory limits, and the red layers/tiles were unable to fit within those limits.

Subpages (1): GPU Architecture Roadmap

Comments

You do not have permission to add comments.