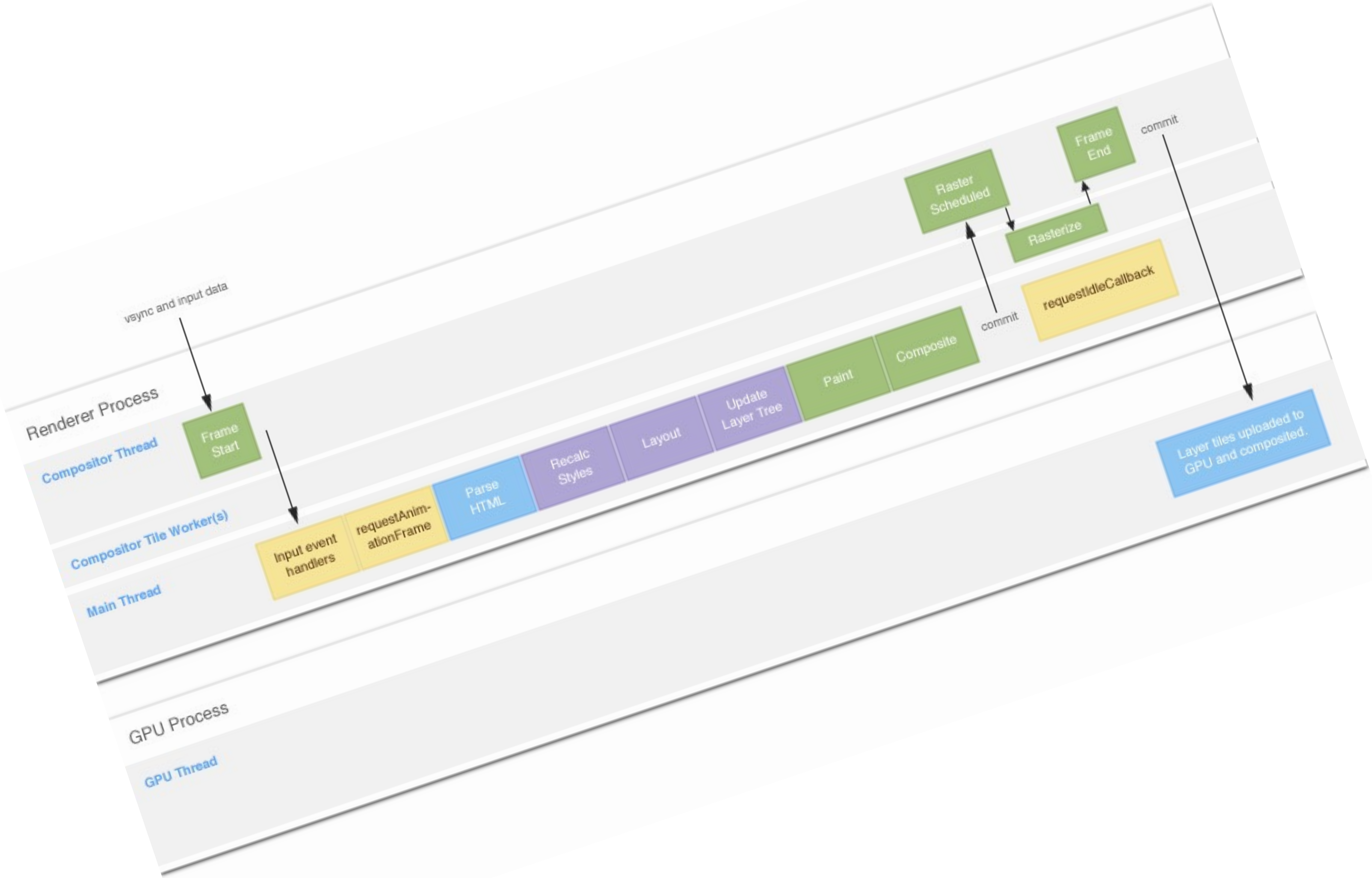

Renderer Process
Compositor Thread
Compositor Tile Worker(s)
Main Thread
GPU Process
GPU Thread
vsync and input data
Frame Start
Input event handlers
requestAnimationFrame
Parse HTML
Recalc Styles
Layout
Update Layer Tree
Paint
Composite
commit
Raster Scheduled
Rasterize
Frame End
requestIdleCallback
commit
Layer tiles uploaded to GPU and composited.

The Chromium Projects

Search this site

Home
Chromium
Chromium OS

Quick links
Report bugs
Discuss
خريطة الموقع

Other sites
Chromium Blog
Google Chrome Extensions

Except as otherwise noted, the content of this page is licensed under a Creative Commons Attribution 2.5 license, and examples are licensed under the BSD License.

For Developers > Design Documents >

GPU Accelerated Compositing in Chrome

Tom Wiltzius, Vangelis Kokkevis & the Chrome Graphics team

updated May 2014

This code is changing due to Slimming Paint and thus there may be large changes in the future. Note also that some class names may have changed (e.g. RenderObject to LayoutObject, RenderLayer to PaintLayer).

Summary

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

Introduction: Why Hardware Compositing?

Traditionally, web browsers relied entirely on the CPU to render web page content. With capable GPUs now an integral part of even the smallest of devices, attention has turned on finding ways to more effectively use this underlying hardware to achieve better performance and power savings. Using the GPU to composite the contents of a web page can result in very significant speedups.

The benefits of hardware compositing come in three flavors:

1. Compositing page layers on 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. The hardware is designed specifically for these types of workloads.
2. Expensive readbacks aren't necessary for content already on the GPU (such as accelerated video, Canvas2D, or WebGL).
3. Parallelism between the CPU and GPU, which can operate at the same time to create an efficient graphics pipeline.

Lastly, before we begin, a big disclaimer: the Chrome graphics stack has evolved substantially over the last several years. This document will focus on the most advanced architecture at the time of writing, which is not the shipping configuration on all platforms. For a breakdown of what's enabled where, see the GPU architecture roadmap. Code paths that aren't under active development will be covered here only minimally.

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.

Nodes and the DOM tree

In Blink, 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*. RenderObjects are stored in a parallel tree structure, called the *Render Tree*. A RenderObject knows how to 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 responsible for writing the pixels into a bitmap that eventually get displayed to the screen. In Chrome, the GraphicsContext wraps Skia, our 2D drawing library.

Traditionally most GraphicsContext calls became calls to an SkCanvas or SkPlatformCanvas, i.e. immediately painted into a software bitmap (see this document for more detail on this older model of how Chrome uses Skia). But to move painting off the main thread (covered in greater detail later in this document), these commands are now instead recorded into an SkPicture. The SkPicture is a serializable data structure that can capture and then later replay commands, similar to a display list.

From RenderObjects to RenderLayers

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

So how does the GPU come into play? The compositor can use the GPU to perform its drawing step. This is a significant departure from the old software rendering model in which the Renderer process passes (via IPC and shared memory) a bitmap with the page's contents over to the Browser process for display (see "The Legacy Software Rendering Path" appendix for more on how that works).

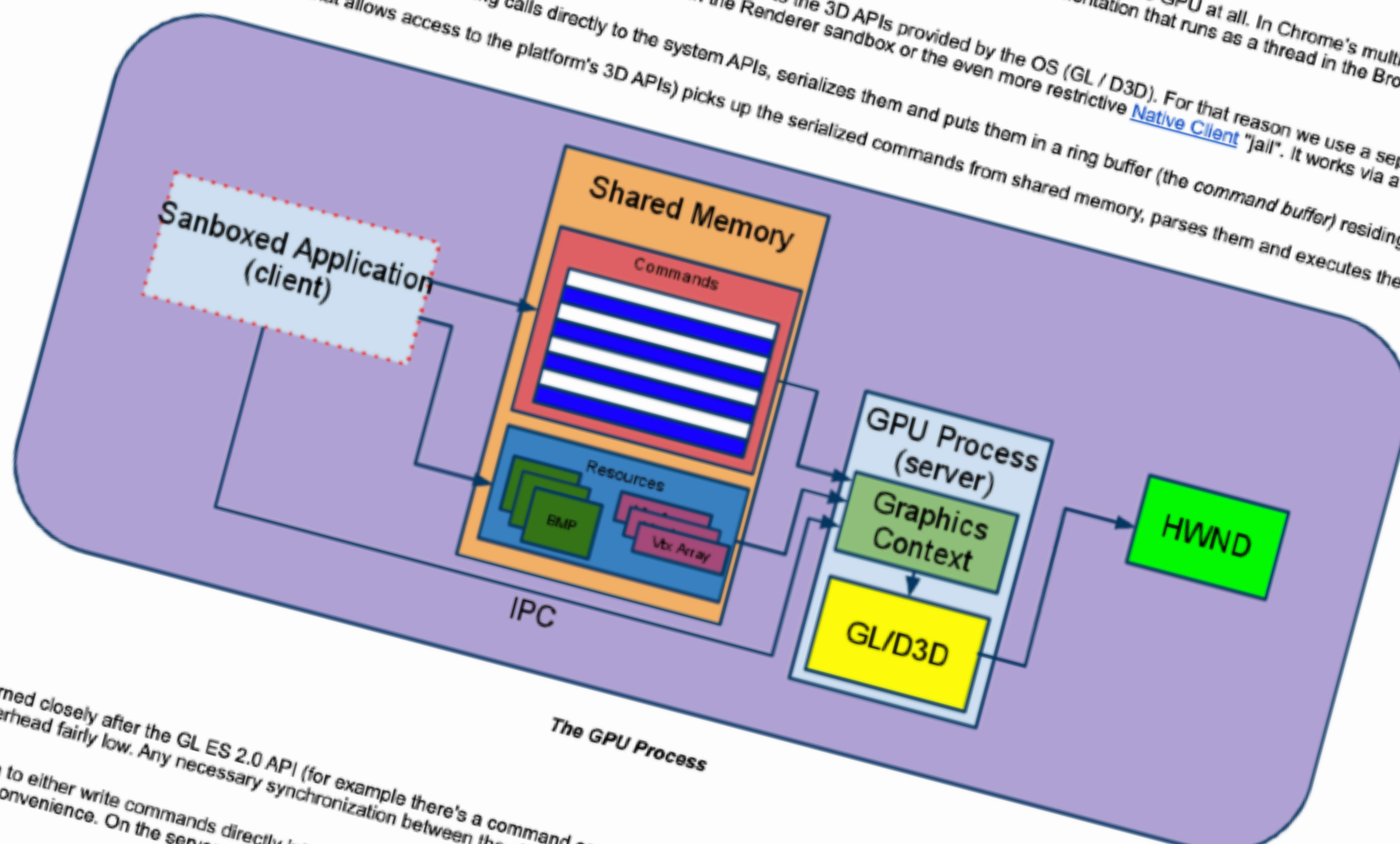
In the hardware accelerated architecture, compositing happens on the GPU via calls to the platform specific 3D APIs (D3D on Windows; GL everywhere else). The Renderer's compositor is essentially using the GPU to draw rectangular areas of the page (i.e. all those compositing layers, positioned relative to the viewport according to the layer tree's transform hierarchy) 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. Note that Android is an exception, where Chrome uses an in-process GPU implementation that runs as a thread in the Browser process. The GPU thread on Android otherwise behaves the same way as the GPU process on other platforms.

Restricted by its sandbox, the Renderer process (which contains an instance of Blink and of cc) cannot directly issue calls to the 3D APIs provided by the OS (GL / D3D). For that reason we use a separate process to access the device. 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.



The GPU Process

The Command Buffer

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.

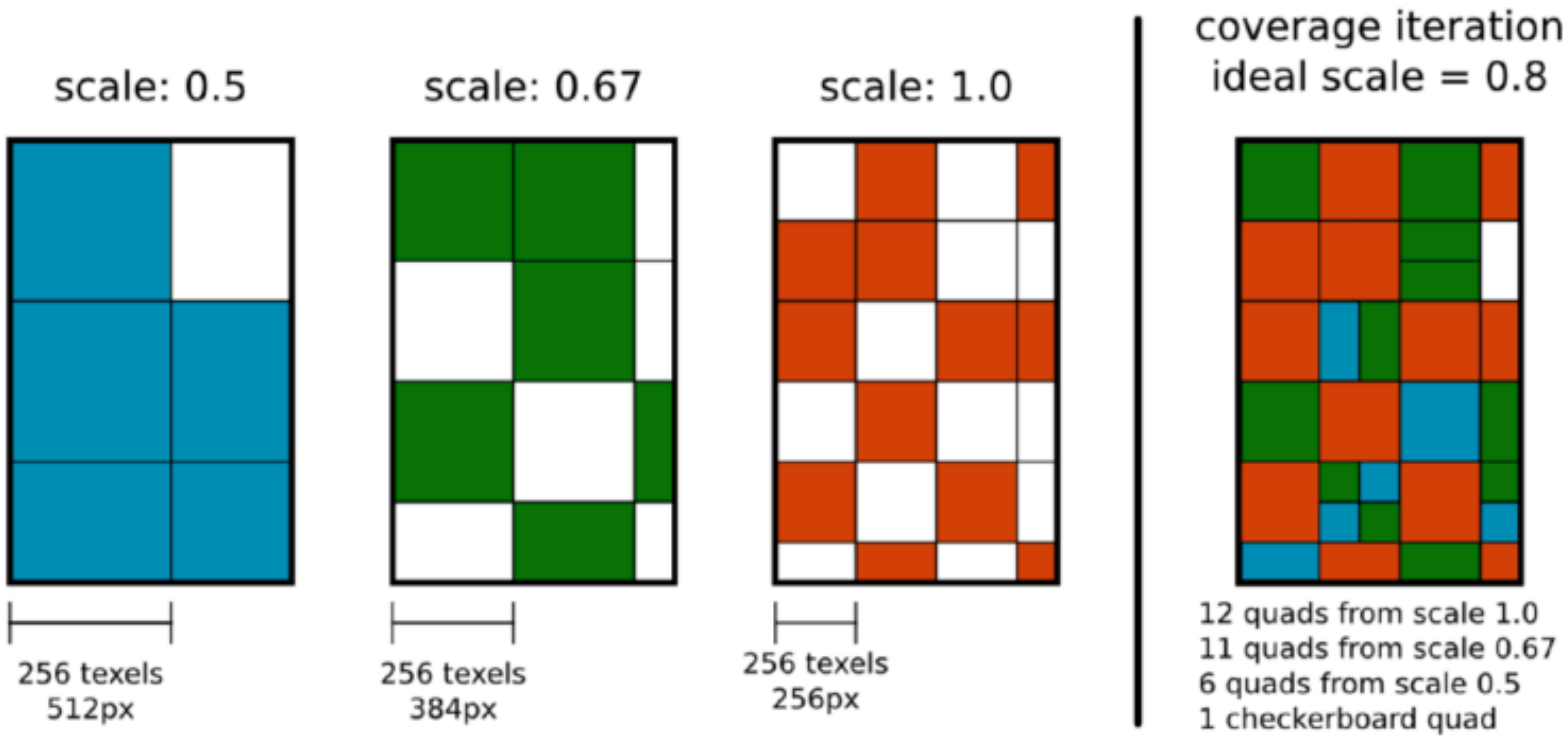
From the client's perspective, an application has the option to either write commands directly into the command buffer or use the GL ES 2.0 API via a client side library that we provide which handles the serialization behind the scenes. Both the compositor and WebGL currently use the GL ES client side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

Resource Sharing & Synchronization

In addition to providing storage for the command buffer, Chrome uses shared memory for passing larger resources such as bitmaps for textures, vertex arrays, etc between the client and the server. See the command buffer documentation for more about the

13 Handling Incremental Invalidation
14 Tile sharing between trees
15 Hosts and Trees Class Diagram
16 Picture Pile overview
17 PictureLayerTiling Overview
18 Iterating Through Tilings
19

Iterating Through Tilings



If you have a set of sparse tilings with resources, what ends up getting drawn on screen?

PictureLayerTilingSet::CoverageIterator allows us to have arbitrary number of tilings at arbitrary scales and still fill the screen as best we can.

cc unfortunately still uses the language of “draw” and “swap” in a number of places, despite the fact that it no longer does either of these things. “Draw” in cc means constructing a compositor frame full of quads and render passes for eventual drawing on screen. “Swap” in cc means submitting that frame to the display compositor via a CompositorFrameSink. These frames get sent to the viz SurfaceAggregator where compositor frames from all frame producers are aggregated together.

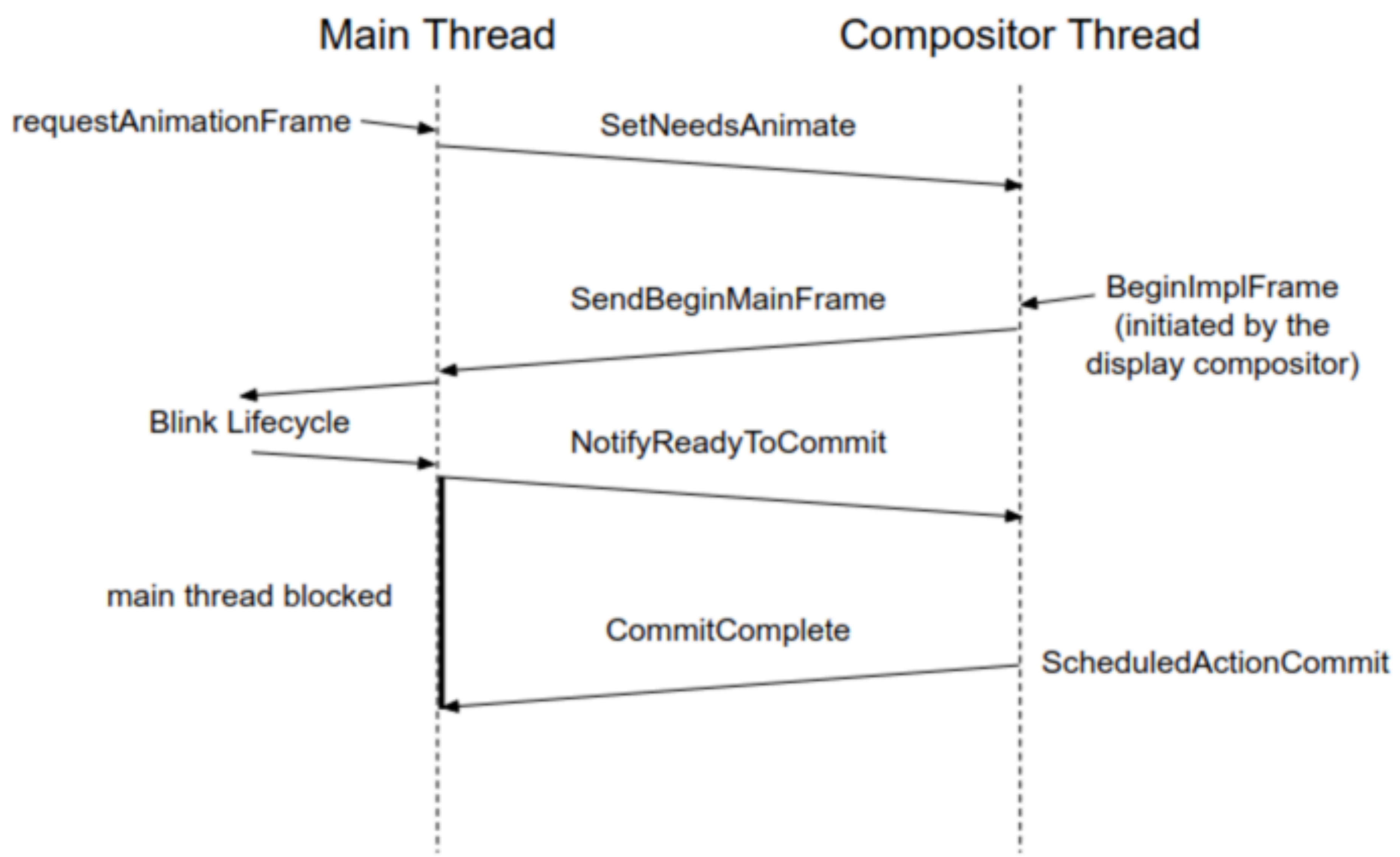
Input Data Flow Overview

The other main piece of input to cc is user input, such as mouse clicks, mouse wheels, and touch gestures. In the renderer process, input is forwarded from the browser process. It is processed by ui::InputHandlerProxy (a cc::InputHandlerClient).

Some of this input is forwarded to the LayerTreeHostImpl (a cc::InputHandler) at specific times. This allows it to modify the active layer’s property tree and scroll or pinch as needed. Some input can’t be handled by the compositor thread (e.g. there’s a synchronous Javascript touch or wheel handler) and so that input is forwarded along to Blink to handle directly. This input flow follows the opposite path of the content data flow in the section above.

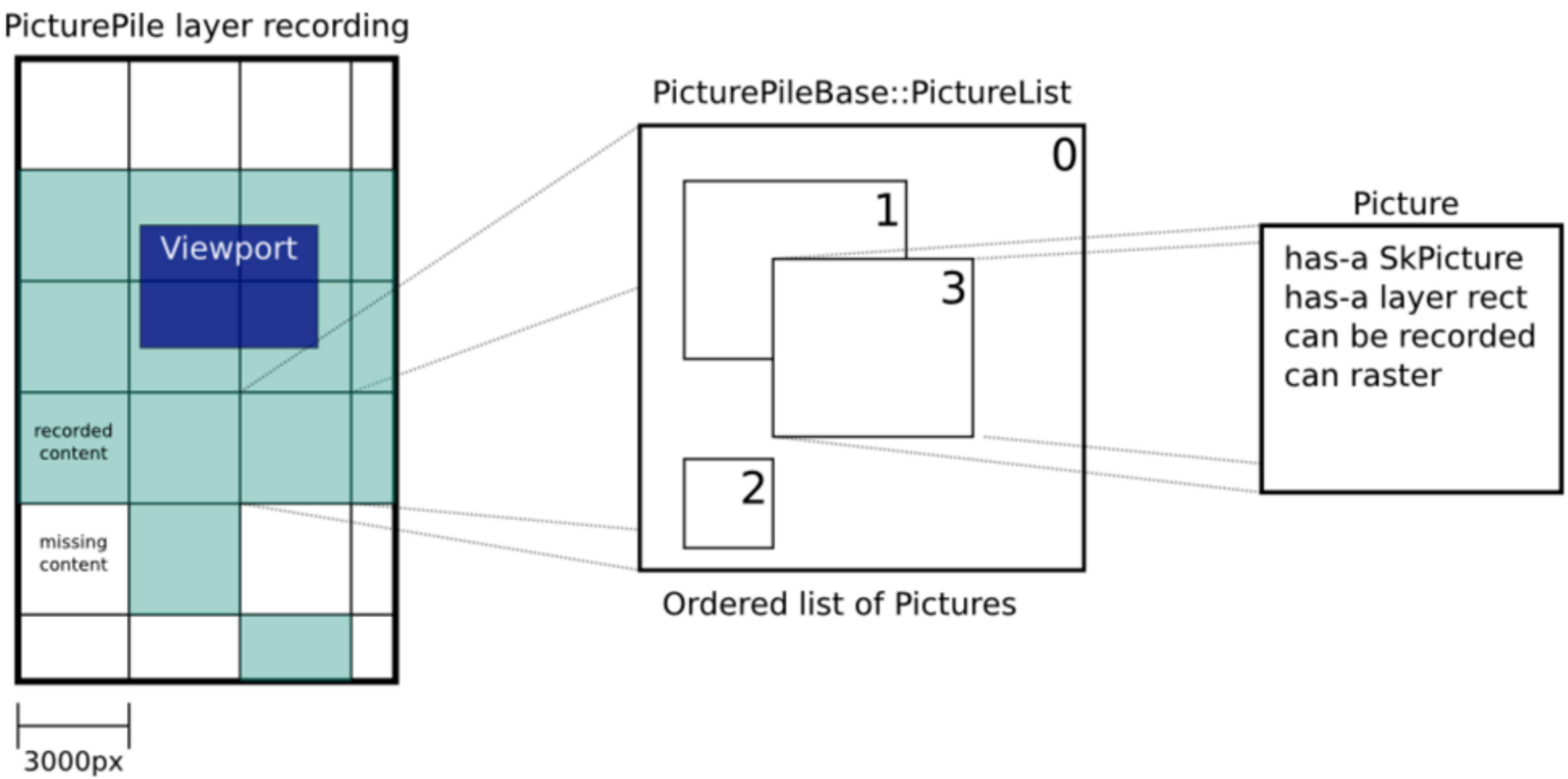
Commit Flow

Commit is a method of getting data atomically from the main thread to the compositor thread. (Even when running in single threaded mode, this operation occurs to move data into the right data structures.) Rather than sending an ipc, commit is done by blocking the main thread and copying data over.



The main thread can request a commit in several ways. Most webpages request one via requestAnimationFrame, which eventually calls SetNeedsAnimate on LayerTreeHost. Additionally, modifying any of cc’s inputs (e.g. a layer property, such as its transform or a change to the layer’s content), will call either SetNeedsAnimate, SetNeedsUpdate, or SetNeedsCommit on LayerTreeHost. The different SetNeeds functions allow for

Picture Pile overview



quad file:^src/cc/ package:^chromium$ | Search Code

all (search all code)

Feedback Shortcuts

[chromium] //src/cc/README.md

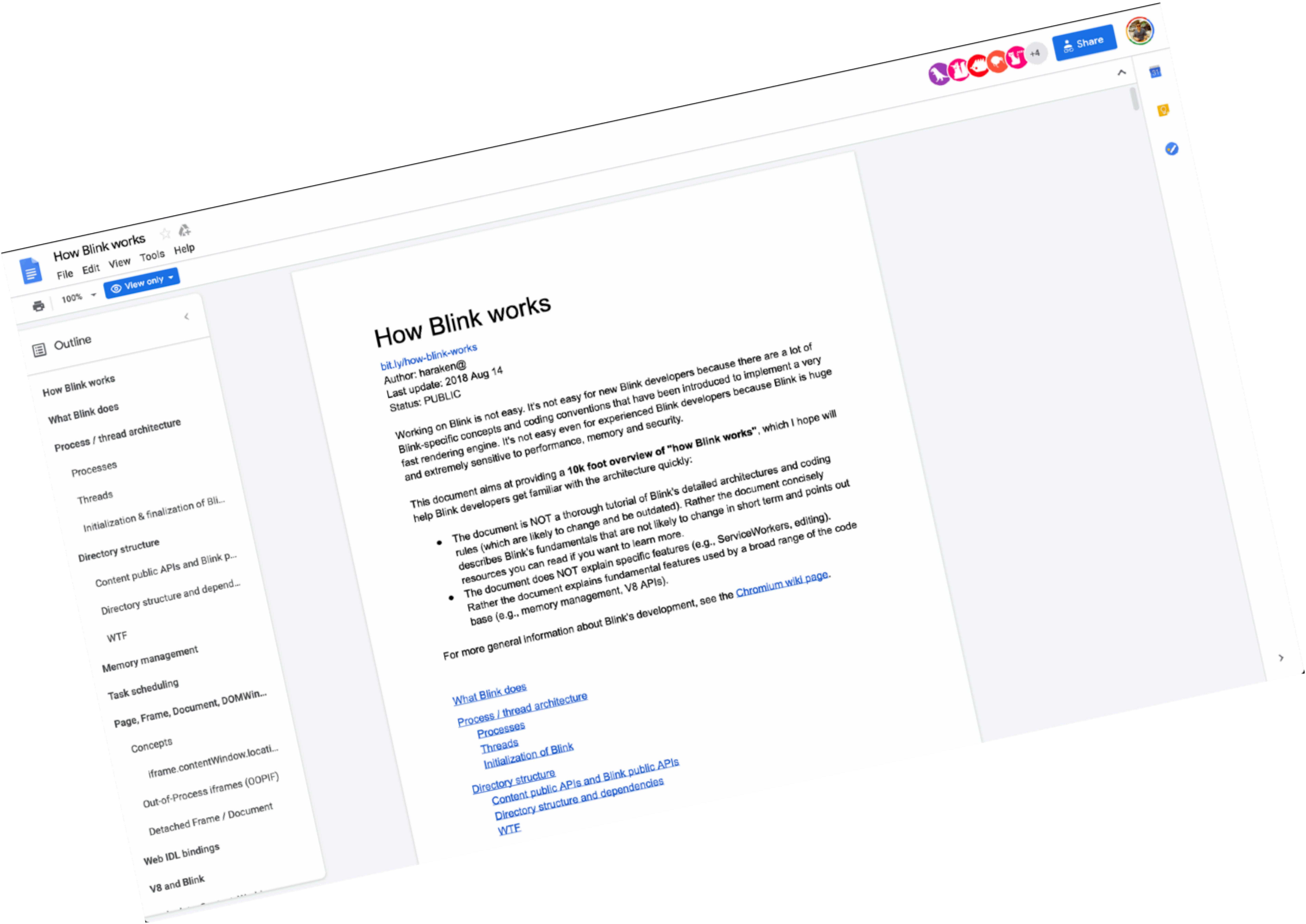
Files Outline

- animation
- base
- benchmarks
- debug
- input
- ipc
- layers
- metrics
- mojo_embedder
- mojom
- paint
- raster
- resources
- scheduler
- test
- tiles
- trees
- BUILD.gn
- DEPS
- OWNERS
- PRESUBMIT.py
- README.md
- cc.gni
- cc_export.h
- memory.md
- typemaps.gni

README.md

History | Layers | Find | Goto | Link | View in | Related files

```
# cc/

This directory contains a compositor, used in both the renderer and the
browser.  In the renderer, Blink is the client.  In the browser, both
ui and Android browser compositor are the clients.

The public API of the compositor is LayerTreeHost and Layer and its
derived types.  Embedders create a LayerTreeHost (single, multithreaded,
or synchronous) and then attach a tree of Layers to it.

When Layers are updated they request a commit, which takes the structure
of the tree of Layers, the data on each Layer, and the data of its host and
atomically pushes it all to a tree of LayerImpls and a LayerTreeHostImpl
and LayerTreeImpl.  The main thread (which owns the tree of Layers
and the embedder) is blocked during this commit operation.

The commit is from the main thread Layer tree to the pending tree in
multithreaded mode.  The pending tree is a staging tree for
rasterization.  When enough rasterization has completed for
invalidations, the pending tree is ready to activate.  Activate is an
analogous operation to commit, and pushes data from the pending tree to
the active tree.  The pending tree exists so that all of the updates
from the main thread can be displayed to the user atomically while
the previous frame can be scrolled or animated.

The single threaded compositor commits directly to the active
tree and then stops drawing until the content is ready to be drawn.

The active tree is responsible for drawing.  The Scheduler and its
SchedulerStateMachine decide when to draw (along with when to commit,
etc etc).  "Drawing" in a compositor consists of LayerImpl::AppendQuads
which batches up a set of DrawQuads and RenderPasses into a
CompositorFrame which is sent via a CompositorFrameSink.

CompositorFrames from individual compositors are sent to the
SurfaceManager (currently in the browser process).  The
SurfaceAggregator combines all CompositorFrames together and asks
the Display to finally draw the frame via Renderer, which is either
a viz::GLRenderer or a SoftwareRenderer, which finally draws the entire
composited browser contents into a backbuffer or a bitmap, respectively.

Design documents for the graphics stack can be found at
[chromium-graphics](https://www.chromium.org/developers/design-documents/chromium-graphics).

## Other Docs

* [How cc Works](../docs/how_cc_works.md)

## Glossaries

### Active CompositorFrame

### Active Tree
```



How Blink works

bit.ly/how-blink-works
Author: haraken@
Last update: 2018 Aug 14
Status: PUBLIC

Working on Blink is not easy. It's not easy for new Blink developers because there are a lot of Blink-specific concepts and coding conventions that have been introduced to implement a very fast rendering engine. It's not easy even for experienced Blink developers because Blink is huge and extremely sensitive to performance, memory and security.

This document aims at providing a **10k foot overview of "how Blink works"**, which I hope will help Blink developers get familiar with the architecture quickly:

- The document is NOT a thorough tutorial of Blink's detailed architectures and coding rules (which are likely to change and be outdated). Rather the document concisely describes Blink's fundamentals that are not likely to change in short term and points out resources you can read if you want to learn more.
- The document does NOT explain specific features (e.g., ServiceWorkers, editing). Rather the document explains fundamental features used by a broad range of the code base (e.g., memory management, V8 APIs).

For more general information about Blink's development, see the Chromium wiki page.


Handling Incremental Invalidation
Tile sharing between trees
Hosts and Trees Class Diagram
Picture Pile overview
PictureLayerTiling Overview
Iterating Through Tilings
Iterating Through Tilings
scale: 0.5
256 texels
512px
cc unfortunately still us
"Draw" in cc means cor
that frame to the displ
all frame producers are
If you have a set of sparse tilings with resources, what ends up getting drawn on
debug
input
ipc
layers
metrics
mojo_embedder
mojom
ui and Android browser compositor are
The public API of
Layer and its
le, multithreaded,
kes the structure
lata of its host
ayerTreeHostImpl
of Layers
Share
How Blink works
File Edit View Tools Help
100%
View only
Outline
How Blink works
What Blink does
Process / thread architecture
Processes
Threads
Initialization & finalization of Bli...
Directory structure
Content public APIs and Blink p...
Directory structure and depend...
WTF
Memory management
Task scheduling
Page, Frame, Document, DOMWin...
Concepts
iframe.contentWindow.locati...
Out-of-Process iframes (OOPIF)
Detached Frame / Document
Web IDL bindings
V8 and Blink
How Blink works
bit.ly/how-blink-works
Author: haraken@
Last update: 2018 Aug 14
Status: PUBLIC
Working on Blink is not easy. It's not easy for new Blink developers because there are a lot of Blink-specific concepts and coding conventions that have been introduced to implement a very fast rendering engine. It's not easy even for experienced Blink developers because Blink is huge and extremely sensitive to performance, memory and security.
This document aims at providing a 10k foot overview of "how Blink works", which I hope will help Blink developers get familiar with the architecture quickly:
The document is NOT a thorough tutorial of Blink's detailed architectures and coding rules (which are likely to change and be outdated). Rather the document concisely describes Blink's fundamentals that are not likely to change in short term and points out resources you can read if you want to learn more.
The document does NOT explain specific features (e.g., ServiceWorkers, editing). Rather the document explains fundamental features used by a broad range of the code base (e.g., memory management, V8 APIs).
For more general information about Blink's development, see the Chromium wiki page.
What Blink does
Process / thread architecture
Processes
Threads
Initialization of Blink
Directory structure
Content public APIs and Blink public APIs
Directory structure and dependencies
WTF
Architectural Interlude: The GPU P
Before we go any further exploring the GPU comm
the GPU process. The GPU process exists primari
behaves the same way as the GPU process on othe
Restricted by its sandbox, the Renderer process (which
the GPU Process. The GPU process is specifically desig
The client (code running in the Renderer or within a
process.
The server (GPU process running in a less restrictive sa
GPU Process
GPU Thread
Sandboxed Application (client)
Commands
The main thread can request a commit in several ways. Most webpages request one via requestAnimationFrame, which eventually calls SetNeedsAnimate on LayerTreeHost. Additionally, modifying any of cc's inputs (e.g. a layer property, such as its transform or a change to the layer's content), will call either SetNeedsAnimate, SetNeedsUpdate, or SetNeedsCommit on LayerTreeHost. The different SetNeeds functions allow for
memory shared between itself and the server
and executes the appropriate graphics calls.
we call this process

bit.ly/code-to-pixels-learning-material