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.

# Compositor Thread Architecture

<jamesr, enne, vangelis, nduca> @chromium.org

## Goals

The main render thread is a pretty scary place. This is where HTML, CSS, Javascript and pretty much everything on the web platform runs... or originates. It routinely stalls for tens to hundreds of milliseconds. On ARM, stalls can be seconds long. Sadly, it is not feasible to prevent all these stalls: style recalculation, synchronous network requests, long painting times, garbage collection, all these things have content-dependent costs.

The compositor thread architecture allows us to snapshot a version of the page and allow the user to scroll and see animations directly on the snapshot, presenting the illusion that the page is running smoothly.

## Background

Some background on the basic frontend compositor archtecture, as well as Chrome's gpu architecture, can be found here: http://dev.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome

## Basic Approach

The compositor is architected into two halves: the main thread half, and the "impl thread" half. The word "impl" is horribly chosen, sorry! :)

The main thread half of the world is a typical layer tree. A layer has transformation, size, and content. Layers are filled in on-demand: layers can be damaged (setNeedsDisplayInRect). The compositor decides when to run the layer delegate to tell it to paint. This is similar to InvalidateRect/Paint model you see in most operating systems, but just with layers. Layers have children, and can clip/reflect/etc, allowing all sorts of neat visual effects to be created.

The impl-side of the compositor is hidden from users of the layer tree. It is a nearly-complete clone of the main thread tree --- when we have a layer on the main thread, it has a corresponding layer on the impl thread. Our naming is a little strange but:

- LayerChromium : the main thread version of a layer
- CCLayerImpl : the impl-side verison of a layer

The main thread tree is a model of what webkit wants to draw. The main thread paints layer contents into textures. These are handed to the impl tree. The impl tree is actually what gets drawn to the screen. We can draw the impl tree anytime, even while the main thread is blocked.

Users of the LayerChromium tree can specify that layers are scrollable. By routing all input events to the impl thread before passing them to the main thread, we can scroll and redraw the tree without ever consulting the main thread. This allows us to implement “smooth scrolling” even when the main thread is blocked.

Users of the LayerChromium tree can add animations to the tree. We can run those animations on the impl tree, allowing hitch-free animations.

## Tree Synchronization, Hosts and Commits

Every tab in Chromium has a different layer tree. Each tab has a layer tree host, which manages the tab-specific state for the tree. Again:

- CCLayerTreeHost: owns a tree of LayerChromiums, CCLayerTreeHost::m_rootLayer
- CCLayerTreeHostImpl: owns a tree of CCLayerImpls, CCLayerTreeHostImp::m_rootLayer

These two trees, the main thread tree and the impl tree are completely isolated from one another. The impl tree is effectively a copy of the main thread tree, although with a different data type. We manually synchronize the impl tree to the main thread tree periodically, a process we call “commit”. A commit is a recursive walk over the main tree’s layers where we push “pushPropertiesTo” the impl-side equivalent of a layer. We do this on the impl thread with the main thread completely **blocked.**

The basic logic of when to perform a commit is delayed. When the main tree changes, we simply make a note that a commit is needed (setNeedsCommit). When a layer’s contents change, e.g. we change a HTML div text somehow, we treat it as a commit. Later (under the discretion of a scheduler, discussed later) we decide to perform the commit. A commit is a blocking operation but still very cheap: it typically takes no more than a few milliseconds.

An aside on our primitive thread model: we assume that both the main thread and the impl thread are message loops. E.g. they have postTask and postDelayedTask primitives. We try to keep both threads idle as often as possible and prefer async programming to taking a lock and blocking the thread.

The commit flow is as follows (see CCThreadProxy for implementation):

- The main thread gets damage. This turns into a setNeedsCommit
- We post the setNeedCommit message to the impl thread.
- The impl thread passes the setNeedsCommit to the CCScheduler. Note, the scheduler is an impl-side concept --- it cannot access any state on the main thread.
- The scheduler will consider the overall system state (whether we recently drew, when the next frame is, many other things) and will eventually say “okay, begin a commit”
- The beginFrameAndCommit command from the scheduler turns into a postTask back to the main thread.
- When the bFAC message runs on the main thread we do the following things:

    - Apply any impl-side scrolls to the main thread
    - Call the requestAnimationFrame callback
    - Perform any pending layout (namely, HTML layout)
    - Paint any layers that need to be painted (software rasterization)
- Once painting is done, we post a message to the impl thread saying “beginCommit.” The main thread then waits on a “commit done” event. This event will be signaled by the impl thread when it finishes the commit.
- The beginCommit message on the impl thread uploads textures, and then synchronizes the trees. When both are done, it signals the “commit done” event, which unblocks the main thread which was previously waiting on that event. This finishes the commit and the two trees are synchronized.

At this point, the impl tree can draw as often as it wants without consulting the main thread. Similarly, the main thread (thus javascript, etc) can mutate the main thread tree as much as it wants without consulting the impl thread.

We have one very important rule in the CCThreadProxy architecture: the main thread can make blocking calls to the impl thread, but the impl thread cannot make a blocking call to the main thread. Breaking this rule can lead to deadlocks.

## CCProxy

To allow development of the threaded compositor while still shipping a single-threaded compositor, we have made it possible to run the same basic two-tree architecture in both single- and threaded modes. In single threaded mode, we still have two trees and delayed commits, but simply run a different synchronization/scheduling algorithm and host the tree on the main thread. This is implemented by the CCProxy interface, which abstracts the types of communication that go on between the main thread and the impl thread. For instance:

- setNeedsCommit: tells the proxy to schedule a commit of the main thread tree to the impl tree
- setNeedsRedraw: tells the proxy to draw the impl tree (without synchronizing the trees)
- setVisible: tells the proxy to make the impl visible/invisible.
- compositeAndReadback(void* buf)
- … lots more

Thus, there are two subclasses of CCProxy:

- CCSingleThreadProxy: runs the compositor in single thread mode, where the impl tree exists and is drawn on the main thread
- CCThreadProxy: runs the compositor on another thread, the impl thread. This is the “threaded compositor” mode.

### CCScheduler

In addition to synchronizing trees, we have a lot of logic in the compositor that deals with when to commit, when to draw, whether to run animations, when to upload textures, and so on. This logic is not specific to whether the impl is running on the compositor thread or the main thread, so is put inside a standalone class called the CCScheduler. The scheduler exists logically as part of the impl side of tree, and thus in threaded mode lives on the impl thread.

The scheduler itself is a very simple class that glues together two key systems:

- CCFrameRateController: decides when good times to draw are. It listens to the underlying OS' vsync api to detect vsync intervals as well as progress updates from the GPU. Its job is to dynamically pick a target frame rate and periodically kick the scheduler and tell it "now is a good time to draw."
- CCSchedulerStateMachine: tracks all the state of the compositing operation, e.g. screen dirty, commit needed, commit begun, and so on. We try to keep all corner case logic ("oops, you cant draw now") inside this code so that it can be exhaustively unit tested.

## Input Handling

A key use of the compositor thread is to scroll pages smoothly even when the main thread is blocked. We do this by intercepting input events before they arrive on the main thread's event loop and redirecting them onto the impl thread.

Once on the impl thread, they hit the WebCompositorInputHandler. This handler looks at the events and can ask the impl tree to try to scroll particular layers. However, scrolls can sometimes fail: WebKit does not give every scrollable area a layer (and associated clip objects). Therefore, on the impl tree, we track on each layer areas that cannot be impl-side scrolled. If the scroll request from the WebCompositorInputHandler fails because of hitting one of these areas, then we post the scrolling event to the main thread for normal processing. We call main-thread handled scrolls "slow scrolls" and impl-thread-side scrolls "fast scrolls."

## Memory Management

The compositor is based around the idea of caching the contents of a layer in texture (or other GPU-friendly representation). This uses memory, of course. Chrome, being a tabbed browser, can sometimes have hundreds of tabs open and we need to somehow manage memory between all those tabs.
We use a two-level memory management scheme. In the GPU process, we have a GpuMemoryManager that tracks the visibility of all the tabs, and the association of graphics contexts to those tabs. Roughly, it figures out which graphics contexts should get what amount of the total GPU resources based on visibility and recently-used-ness. The global memory manager also factors in the workload requested by each tab, so that a big gmail tab can actually get more than, for example, a little popup window.

At the compositor level, each LayertTreeHost/Impl pair get an allocation from the GPU process for a certain memory budget. They are to do their best to not exceed this memory budget. We do this by prioritizing all the tiles on all layers, and then giving out memory budget to each tile in descending priority order until we hit our limit. Prioritization includes things like visibility, distance from viewport, whether the tile is on an animating layer, and whether the current layer velocity is likely to bring the tile onscreen.

## Texture Upload

One challenge with all these textures is that we rasterize them on the main thread of the renderer process, but need to actually get them into the GPU memory. This requires handing information about these textures (and their contents) to the impl thread, then to the GPU process, and once there, into the GL/D3D driver. Done naively, this causes us to copy a single texture over and over again, something we definitely don't want to do.

We have two tricks that we use right now to make this a bit faster. To understand them, an aside on “painting” versus “rasterization.”

- Painting is the word we use for telling webkit to dump a part of its RenderObject tree to a GraphicsContext. We can pass the painting routine a GraphicsContext implementation that executes the commands as it receives them, or we can pass it a recording context that simply writes down the commands as it receives them.
- Rasterization is the word we use for actually executing graphics context commands. We typically execute the rasterization commands with the CPU (software rendering) but could also execute them directly with the GPU using Ganesh.
- Upload: this is us actually taking the contents of a rasterized bitmap in main memory and sending it to the GPU as a texture.

With these definitions in mind, we deal with texture upload with the following tricks:

- Per-tile painting: we pass WebKit paint a recording context that simply records the GraphicsContext operations into an SkPicture data structure. We can then rasterize several texture tiles from that one picture.
- SHM upload: instead of rasterizing into a void* from the renderer heap, we allocate a shared memory buffer and upload into that instead. The GPU process then issues its glTex* operations using that shared memory, avoiding one texture copy.

The holy grail of texture upload is “zero copy” upload. With such a scheme, we manage to get a raw pointer inside the renderer process’ sandbox to GPU memory, which we software-rasterize directly into. We can’t yet do this anywhere, but it is something we fantasize about.

## Animation

We allow animations to be added with layers. They allow you to fade or translate layers using “curves,” which are keyframed representations of the position or opacity of a layer over time. Although animations are added on the main thread, they are executed on the impl thread. Animations done with the compositor are thus “hitch free.”

## Terminology

Threads:

- WebKit thread == Main Thread. This is the thread on which the LayerChromium hierarchy lives.
- Compositor thread == the thread on which we will perform compositing. We call it impl thread because this is where the implementation of compositing happens.
- IO Thread == the chromium thread that receives IPCs

Impl thread is a word we use often. The compositor can operate in either single or threaded mode. Impl thread merely means “this lives on the impl half of the system.” Seeing the word “impl thread” does not mean that that code only runs on the compositor thread -- it just means that it handles data that is part of the impl part of the architecture.

Suffixes indicate which thread data lives on:

- Impl class lives on the compositor thread. Eg cclayertreehostimpl
- Lack of suffix means data on the main thread. Eg cclayertreehost

We use words that are ordinarily synonyms to mean very important and distinct steps in the updating of the screen:

- Painting: this is the process of asking Layers for their content. This is where we ask webkit to tell us what is on a layer. We might then rasterize that content into a bitmap using software, or we might do something fancier. Painting is a main thread operation.
- Drawing: this is the process of taking the layer tree and smashing it together with OpenGL onto the screen. Drawing is an impl-thread operation.

## Comments

You do not have permission to add comments.

Sign in | Report Abuse | Print Page | Remove Access | Powered By Google Sites