Chrome Rendering Model
(Ubercompositor++)

{danakj, piman, aelias, enne, nduca}@chromium.org

Status

Fully functioning in for content (tiled/picture, solid color layers). Turn it on in Ash/ChromiumOS with --enable-delegated-renderer.

skaslev working on software implementation
alexst working on webgl
junov working on canvas
danakj working on html5 video
piman working on pepper flash

Tracking bug: crbug.com/123444

Problems that concern us

P0 - Overdraw: on some machines, we consume memory bandwidth for every layer drawn. This is a critical problem on budget machines.

P1 - Software mode is too slow for some cases: fixed position layouts paint-aggregate poorly and Android shift scrolling isn't fast enough.

P3 - <webview> Tag and Out-of-Process iframes: require us to be able to chain-together compositors and make software-mode chain as well.

P3 - Pepper layers: Flash and PDF viewer could render more efficiently if they could export layers to their parent compositor.

P3 - UpdateRect and Swap messages are separate. This is a nightmare and causes all sorts of bad problems.

P3 - Architectural burden of different software/hardware mode: supporting all these rendering modes sucks.

Constraints

We will need to use minimal bandwidth to render the browser on GPU bandwidth-constrained machines, e.g CrOS devices.

We need to support some sort of software rendering mode. It shouldn’t suck.

We need to support nested composited renderers. Aura, pepper, <webview>/iframe tags are real things.

CCQuads: A Technical Aside to Existing CC Experts

typedef Vector<RenderPassList> CCQuads? We’re all calling this the quads architecture. Lets maybe make a data type called CCQuads so that we can avoid saying “quads” but really meaning “array of render passes that contain quads”.

Solution

We make the “compositor mode” an always-on type of thing. We remove the existing shift-scrolling software rendering mode and make a backend for the existing compositor that works with software-side 2D drawing commands. Aura, even, can be drawn using this software compositor, since compositing is simply a ‘layers of textures’ abstraction.

In parallel, we make the compositor’s CCDrawQuad and related architecture multiprocess. The existing compositor rendering pipeline converts the complex Layer Tree data structure into an array of rendering passes, each of which is an array of quads. To make this multi-process, we have the child-process compositor serialize its quad list. That quad list can be sent to a parent process, where it can be deserialized, and then either (a) rendered using whatever rendering system the parent process uses, or (b) sent up one more level to that process’ parent.

At each child->parent exchange, texture *data* are not passed, only resource IDs. Every process that renders is given a ResourceProvider interface that is used to allocate, upload, and delete texture resources. Once uploaded, these resources can be referred to by ID.

Instead of “swapBuffers” and “updateRect,” we instead will have “swapQuads.” This will pass a new set of quads and render passes to the parent compositor.

How this Works In Practice

- Ubercompositing: the “web compositor” exports a quad list to Aura compositor. The aura compositor adds these quads to its overall layer tree and renders all the quads in a single rendering pass. This gives us minimal rendering bandwidth.

- Software mode: we always have a web compositor mode. The renderer process uses a transport-dib-based ResourceProvider to export renderpasses + drawquads to

RenderWidgetHostView. The RenderWidgetHostView directly draws the Quad lists to its HWND(etc) using the platform's native drawing API.

- <webview> tag (etc): the child compositor exports its quads to the parent compositor. Scrolling and animation is done at the leaf nodes of the process tree.

- Pepper quads: we expose the RenderPass/QuadList/Quad abstraction to pepper. Flash and PDF can render using quads instead of 3D or 2D surfaces. Flash would export two big quads --- one for its video layer and one for its UI layer. PDF would export quads for each of its internal rendering chunks, doing scrolling simply by moving quads up and down, filling in the quads as time permitted.

- UpdateRect vs Swap: Everything is replaced with a CompositorFrame message.

- Architectural messiness: always-in-compositor mode cleans up tons of messes that we currently have.

- For external textures (canvas/webgl/video/flash), we need a mechanism to take ownership of the texture and transport it to the parent (browser) compositor. We accomplish this with texture mailboxes. The mailbox can be passed over process boundaries and read/written via a local GL texture id.

Next Steps/Opens

- Touch hit testing and IME might pass information over quads
- Pass windowed NPAPI positioning information via quads
- ~~Refinement of quad transport scheme and DelegatedRendererLayer~~
- ~~Separate RenderSurfaces and CCLayerImpl class hierarchy from LayerRendererChromium, so we can draw a frame without touching the layer tree.~~
- ~~Converting IDs between processes. E.g. render process sees one set of GL IDs, but parent aura process sees a different set of IDs for the same resources~~

Terminology Disambiguation

There are lots of words we throw around here. Definitions for things and their intended meanings. We can reduce the number of terms as we get further along in the project:

- Compositor: ambiguously, may mean a bunch of layers that can be rendered, or just an instance of a quad renderer that draws quads to the screen. Doesn't matter, the key meaning is that it can receive quads and respond to resource creation/uploading requests, and can get stuff that it is given onto the screen.

- Root compositor: the top of the compositor tree. Actually draws to the screen. Actually allocates backings for RenderPasses.

- Nested compositor: a compositor that has child compositors and itself is forwarding quads to a parent compositor. Might have layers, might not. It forwards resource management requests to its parent.

- Child compositor: a generic term meaning that the compositor does not draw, but rather forwards its quads to a parent process.

- Hosting compositor / Parent compositor: used to refer to a compositor instance that can receive quad lists and provide resources. It might be itself a layer compositor, or it might just be a quad renderer.

- Layer compositor: an instance of CCLayerTreeHost that allows creation of additional layers, animations, scrolling, etc. This is what Aura and the Web Compositor create.

- Renderer: the rendering parts of CC, enough to take a list of quads and smash it into a picture, using GL or 2D rendering primitives. Used by any compositor that wants to smash quads together into a picture, especially the root compositor.

- Renderer compositor / Web compositor: refers to an instance of the layer compositor where that is converting web content into quads.

- Aura or UI compositor: an instance of the compositor that renders the browser UI. It will contain DelegatedRendererLayers that themselves contain quads from child processes.

- Uber compositor: a browser configured so that all the composition operations are done at minimal bandwidth. This implies all composition is done in the browser process.

RenderPass, QuadList and DelegatedRendererLayer Details

Point of contact: danakj, enne
Bugs: crbug.com/131213 crbug.com/123445

- DelegatedRendererLayer is a cc::Layer, and is placeholder for where the nested compositor's RenderPasses will be inserted in the host's display.
- The layer is coupled with a DelegatedRendererLayerImpl. This impl layer receives a set of RenderPasses and DrawQuads.
- The impl layer's AppendQuads() needs to append all the quads for its render pass(es) by merging these passes into its compositor's set of passes.

    - This layer will always need to be considered as “having descendents that draw” so it will be made to own a render surface when it applies effects such as opacity. In this case the layer has its own RenderPass.
    - If the layer does not own a surface, then its transforms need to be applied to the quads that it appends. In this case the layer appends all the quads to the RenderPass for its target surface.
    - This means adjusting how AppendQuads() api looks, or adding another method for this. Has AppendContributingRenderPasses() to insert any render passes/quad lists into the compositor’s render pass list in correct order
        - Considered having have multiple DelegatedRendererLayers - one for each pass. But this is bad because the host doesn’t know ahead of time how many passes there will be for a given frame. It just knows where to draw it all - which is its single layer representing the root RenderPass of the child’s frame.

- Need to define quad/pass serializer class. DelegatingRenderer.
    - Need to define the ID renaming scheme (in GL case, each process sees different ID for a resource, even though its the same resource at the root)
        - Handled by ResourceProvider’s PrepareSendToParent(), PrepareSendToChild(), etc.

- RenderPasses are given to the parent compositor, and it does all drawing of intermediate pass/surface textures during its composite of the screen’s final output.
    - This means serializing damage inside quads/passes, so that the host compositor can limit the amount of drawing it does by scissoring to damage.
    - By providing damage, the host compositor can also cache render surface/pass intermediate textures when possible, providing compatibility with optimizations we are currently undertaking for a single independent (non-ubercomp) compositor. The host compositor then needs to remember/aggregate the damage rects that are serialized with the render pass, and store this alongside the textures.
    - The host compositor needs to hold textures that last across frames, in such a way that they are linked to render passes. We can do this by remapping the ResourceIds in the quads when they arrive in the parent compositor with the ResourceProvider. The host holds a texture for the RenderPass, linked by this ResourceId.
        - RenderPass intermediate textures do not require an intelligent texture manager, and can simply be allocated by the root compositor.
        - This work can be done for an independent (non-ubercomp) compositor by dropping the textures from RenderSurface and making GLRenderer manage these on its own.

    - All culling of quads/passes based on damage (scissoring) should be done in the host compositor, so that it can cache textures and draw what it deems appropriate. The child compositors should provide as many quads as might be needed by the host.
- Need NPAPI quad that has window positions.

ResourceProvider Details

Point of contact: piman, reveman, aelias

ResourceProvider's job is to allow allocation of texture resources that can then be referred to in DrawQuads.

ResourceProviderClient:

- Constructed with a communication channel to the a ResourceProviderHost.
- TextureUpdater talks to this class.
- SwapQuads(vector<cc::RenderPass>)

- createResource(width, height, type)
- map/unmap
- synchronization primitives
    - make sure we can upload data or render to the quads safely
        - either using a double-buffer model (can mean more painting, and more memory)
        - or pipelining of updates (incurs stronger coupling between compositors)
            - we may want to drive the client compositor scheduling from the host compositor.
- memory manager hooks

ResourceProviderHost:

- Responds to client requests
- Allocates the resources (depending on the Renderer capabilities).
- Tracks "in use by host compositor" state for synchronization.
- Software version: transport dibs for uploading, IPCs. Used when there is never going to be a GPU process (Android).
- GL version: regular GL fences and texture APIs. Quads flow over GPU command buffer.
- Software via GL version: uses transport dibs for uploading, but Quads flow via GPU process to avoid messy interactions with GPU mode
- Proxy version: proxies client calls to a parent ResourceProviderHost (pepper, browser tag use case)

Opens:

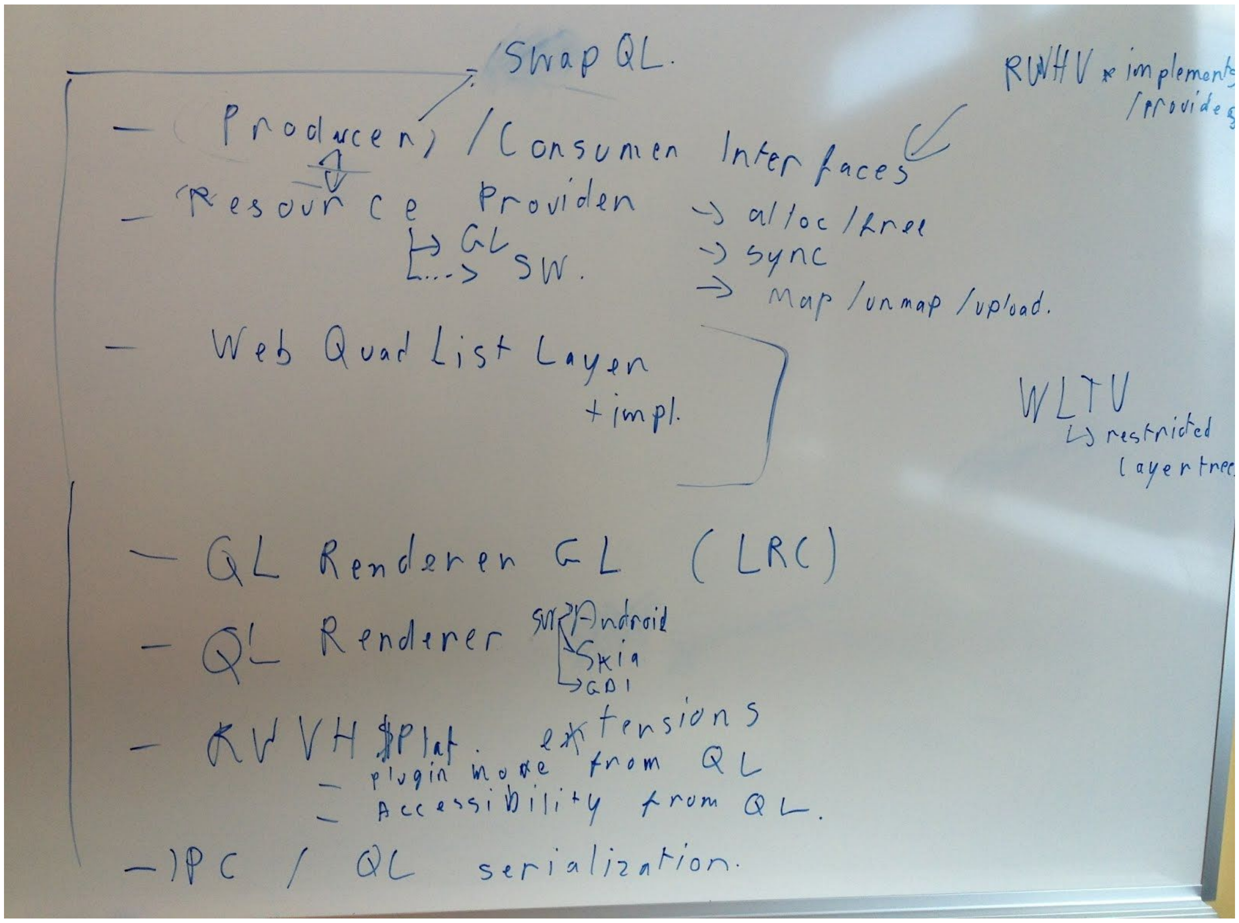
- Texture ids need to be either mapped or shared. If we can share contexts then we don't need to map?
- Validate that fences and map/unmap are sufficient primitives for cross-process resource mgmt
- Pepper/WebGL/... will need to go through the ResourceProviderClient to allocate their render target. Synchronization for the swap will need to go all the way to the host compositor, but we should try to keep it pipelined with proper application of fences.

Renderer (consumer of RenderPasses/DrawQuads)

Contact: aelias

- Takes a set of RenderPasses (holding sets of DrawQuads) as input. Draws the frame to its target output buffer.
- Renderer base class
    - GLRenderer
    - SoftwareRenderer (takes SoftwareOutputDevice) (2d/Software)
        - SoftwareOutputDevice for a SkiaCanvas
        - SoftwareOutputDevice for Android
        - Maybe GDI?
        - Independently instantiable in the RenderWidgetHostView so it can dump a set up RenderPasses in.
    - Implements DrawRenderPass.

Pretty Pictures



Swap QL.

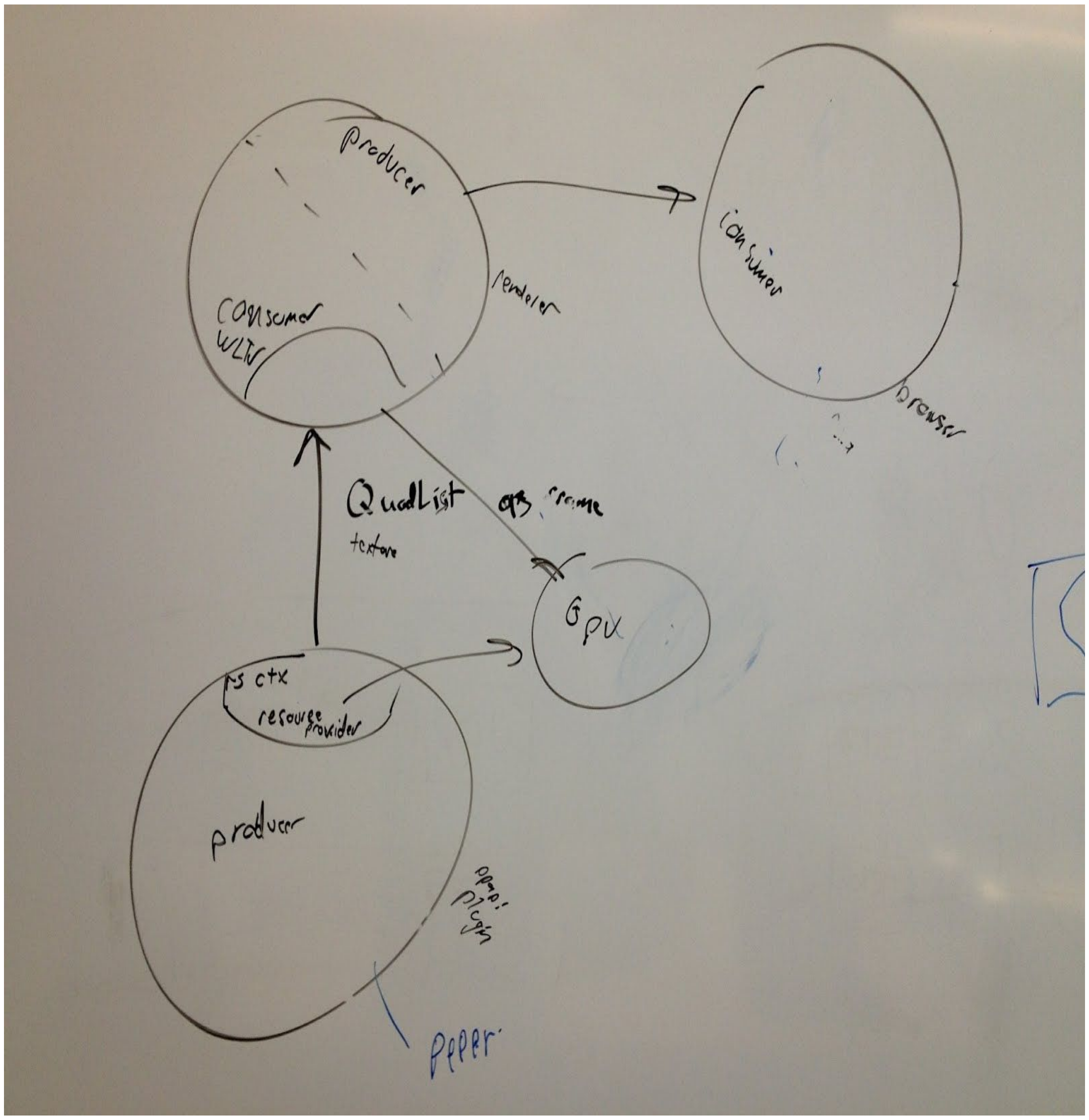
RWHV * implements /provides

- (Producer) / Consumer Interfaces
- Resource Provider
  - → GL
  - → SW.
  - → alloc / free
  - → sync
  - → map / unmap / upload.
- Web Quad List Layer + impl.

WLTV

↳ restricted layer tree

- QL Renderer GL (LRC)
- QL Renderer SW
  - → Android
  - → Skia
  - → GDI
- RWVH $Plat. extensions
  - plugin mode from QL
  - Accessibility from QL.
- IPC / QL serialization.



History

This document original was an internal doc before being moved here. Revision history should mostly be found by following that link (Googlers only, or by request).