Surfaces’ Display Scheduler Phase II Design Doc

Status: ***Work in Progress***
Authors: tansell@google.com
Reviewers: brianderson@google.com
Last Updated: 2015/04/02

Assumed Knowledge

- Chrome Browser, Renderer and GPU architecture
- Surfaces
- Compositing, BeginFrame

Stage 0 - **(complete)** brianderson implements a “stop gap” DisplayScheduler

Stage 1 - Management of time spent rendering by the Display Scheduler
(passing deadline and resource usage)

Stage 2 - Management of GPU resources by the Display Scheduler
(ownership of output_surface creation, passing in output_surface)

Stage 3 - Management of memory resources by the Display Scheduler

Estimators

Task Length Estimator

Input(task name) -> Time taken for each task to take from start to finish
Output(task name, current task length) -> Estimated time for completion + confidence

Slack Estimator

Input -> Delta between when a task requested to start and when it actually started.
Output -> 95%, 99% and 99.9% slack time

Surfaces “Content Request” Addition

Interface

```
class BeginFrameArgs {
  …
}

class BeginFrameResources {
  // Number of threads of execution that are allow to run simultaneously.
  size_t number_of_execution_threads;

  // const ReturnedResourceArray& resources,
  // OutputSurface surface(context?)_to_draw_into,
  // whatever is needed for partial update?
}

class CC_SURFACES_EXPORT SurfaceFactory {
  // Existing methods
  // ----------------------------------------------------
  void Create(SurfaceId surface_id);
  void Destroy(SurfaceId surface_id);
  void SubmitFrame(SurfaceId surface_id,
                   scoped_ptr<CompositorFrame> frame,
                   const DrawCallback& callback);

  // New methods
  // ----------------------------------------------------

  // Tell surfaces that the contents of a surface needs to be updated. If the
  // updates will continue to occur, such as when animating, set continuous to
  // true.
  void ContentNeedsUpdating(SurfaceFactoryClient* ptr, bool continuous, xxx parent hint, my surface id hint);

  // Replaces SubmitFrame
  void SubmitRequestedFrame(
                   SurfaceId surface_id,
                   SurfaceContentRequestId frame_id,
                   scoped_ptr<CompositorFrame> frame,
                   const DrawCallback& callback);
}

class CC_SURFACES_EXPORT SurfaceFactoryClient {
  // Existing methods
  // ----------------------------------------------------

  // Only used for async returning of resources now
  virtual void ReturnResources(const ReturnedResourceArray& resources) = 0;
  // New methods
  // ----------------------------------------------------

  // Request for the client to render/update the contents of a surface.
  virtual void RenderSurfaceContent(
      // Is this actually needed?
      SurfaceId surface_id,
      // Opaque identifier for the content begin requested.
      // The SurfaceFactoryClient must call SubmitFrame(surface_id, frame_id, content)
      SurfaceContentRequestId frame_id,
      // The details needed to render this content.
      const BeginFrameArgs& args,
      // The resources that should be used to render this content.
      const BeginFrameResources& resources) = 0;

}
```

Requests without pipelining

```
// Start requesting frame updates
(on xxx) factory->ContentNeedsUpdating(true);

// Render the first frame
Surface frame_id = 1;
(on yyy) client->RenderSurfaceContent(surface_id, frame_id, xxxxx);
// Client renders content
(on xxx) factory->SubmitRequestedFrame(surface_id, 1, yyy);

// Render the second frame
frame_id++;
(on yyy) client->RenderSurfaceContent(surface_id, frame_id, xxxxx);
…….
(on xxx) factory->ContentNeedsUpdating(false);
(on xxx) factory->SubmitRequestedFrame(surface_id, 2, yyy);

// No more frames requested
```

Requests with pipelining

```
// Start requesting frame updates
factory->ContentNeedsUpdating(true);

// Render the first frame
Surface first_frame_id = 1;
client->RenderSurfaceContent(surface_id, first_frame_id, xxxxx);
```

```
// Requesting the second frame, while the first frame is still pending, IE before the
SubmitRequestedFrame is called
Surface second_frame_id = 2;
client->RenderSurfaceContent(surface_id, frame_id, xxxxx);

// First frame comes back
factory->SubmitRequestedFrame(surface_id, 1, xxxxx);
…….
factory->ContentNeedsUpdating(false);
// Second frame comes back
factory->SubmitRequestedFrame(surface_id, 2, yyyy);

// No more frames requested
```

ProduceFrameTask

- For either a DelegatedFrameHost or ui::Compositor
- Starts with a BeginFrame message
- Ends with SubmitFrame call

DelegatedFrameHost

- OnBeginFrame
    - **IPC Browser -> IPC Renderer**
    - cc::Scheduler::OnBeginFrame
        - BeginImplFrame
            - BeginMainFrame / Commit / Activate
            - ImplFrameDeadline
                - DrawAndSwap
                    - *LTHI::renderer_ is a DelegatingRenderer, GLRenderer or SoftwareRenderer*
                    - *LTHI::output_surface_ is a ?????*
                    - LTHI::DrawLayers
                        - renderer_->DrawFrame
                    - LTHI::SwapBuffers
                        - renderer_->SwapBuffers
                            - **IPC Render -> IPC Browser?**
        - FinishImplFrame

- DelegatedFrameHost->SwapDelegatedFrame

ui::Compositor

-

DrawAndSwapTask

- Draw
  - SurfaceAggregator::Aggregate
  - renderer_->DecideRenderPassAllocationsForFrame
  - renderer_->DrawFrame
- Swap
  - renderer_->SwapBuffers
  - renderer_->output_surface_->SwapBuffers
  - renderer_->output_surface_->client_->DidSwapBuffers
  - renderer_->output_surface_->client_->DidSwapBuffersComplete

----

Side Task 1 - Display fixing

- Rename Display to Window and add a Display object which all windows on the same physical display.
-
- Window objects → ui::gfx::NativeWindow? ui::gfx::NativeView?
  - Have 1 primary display they are currently on.
  - Have n “secondary” displays they are currently on.
  - A DirectRendering context?
  - A compositor?
  - A GPU used for rendering?

- Display objects → ui::gfx::Display
  - On ChromeOS, there is a 1:1 relationship between a Display and a Window
  - On Windows / Mac / Linux, there is a 1:many between a Display and a Window
  - Each Display has a bunch of properties;
    - BeginFrameSource
    - ColorProfile

```
content/browser/render_host/render_widget_host_view_aura.cc
  RenderWidgetHostViewAura::AddedToRootWindow
    delegated_frame_host_->SetCompositor(window_->GetHost()->compositor());

  window_ == aura::Window
  window_->GetHost() == aura::WindowTreeHost
  window_->GetHost()->compositor() == ui::Compositor

 WindowTreeHost::CreateCompositor - called in the window_tree_host_XXXXX impls
```

Renderer
- Submits Frames

cc::Renderer
- Found in cc/output/renderer.h
- Interface for a class which renders to cc::OutputSurface
- Implementations
  - **DirectRenderer** - base class for both GL and software renderers
    - **GLRenderer** - “handles drawing of composited render layers using gl”
    - **SoftwareRenderer** -
  - **DelegatingRenderer** -

cc::OutputSurface
- Found in cc/output/output_surface.h
- Interface which *“represents the output surface for a compositor”*
- Implementations

  - **OutputSurface**
    - **CompositorOutputSurface**
      - content/renderer/gpu/compositor_output_surface.h
      - **MailboxOutputSurface**
        - content/renderer/gpu/mailbox_output_surface.h
      - **DelegatedCompositorOutputSurface**
        - content/renderer/gpu/delegated_compositor_output_surface.h
    - **BrowserCompositorOutputSurface**
      - content/browser/compositor/browser_compositor_output_surface.h
      - **OffscreenBrowserCompositorOutputSurface**
        - content/browser/compositor/offscreen_browser_compositor_output_surface.h
      - **SoftwareBrowserCompositorOutputSurface**
        - content/browser/compositor/software_browser_compositor_output_surface.h
      - **GpuBrowserCompositorOutputSurface**
        - content/browser/compositor/gpu_browser_compositor_output_surface.h
        - GpuSurfacelessBrowserCompositorOutputSurface
          - content/browser/compositor/gpu_surfaceless_browser_compositor_output_surface.h
    - **SynchronousCompositorOutputSurface**
      - content/browser/android/in_process/synchronous_compositor_output_surface.h
    - **SurfaceDisplayOutputSurface**
      - cc/surfaces/surface_display_output_surface.h

  - android_webview/browser/parent_output_surface.h - ParentOutputSurface
  - components/view_manager/surfaces/surfaces_output_surface.h - DirectOutputSurface
  - mojo/cc/direct_output_surface.h - DirectOutputSurface
  - mojo/cc/output_surface_mojo.h - OutputSurfaceMojo
  - content/browser/renderer_host/compositor_impl_android.cc - OutputSurfaceWithoutParent

cc::Display
- On ChromeOS == “real physical display”
- On other platforms == “one chrome window”
- Has a cc::DirectRenderer (which could be either a GLRenderer or a SoftwareRenderer)
-

`cc::OnscreenDisplayClient`

`cc::SurfaceDisplayOutputSurface - fake output surface to allow ui::Compositor to use real output surface directly`

`cc::SurfaceFactory`

- `cc::SurfaceFactoryClient`
    - `ReturnResources()`
- `Create(id)`
- `Destroy(id)`
- `SubmitFrame(id)`

Design

- There are N

Scheduling Policy

OnBeginFrame
OnTaskFinish

Triggers

- A task finishes
- External Triggers
    - VSync

Surfaces Documentation - http://www.chromium.org/developers/design-documents/chromium-graphics/surfaces

- SurfaceManager -> SurfaceFactory
-

https://docs.google.com/document/d/1RxbffpK_GxPtZscXgIEN0N9ZT7IC8BObnbx9ynw92qg/edit
https://docs.google.com/document/d/1pSzIohikuHYGx4ZrkEKybzxwumMAs3eW4AxdVHIKiZY/edit

Brian's Surfaces Diagram -
https://docs.google.com/drawings/d/1-GuIwC2Ta8KSiPuOHvc2aI4rXLUZ03CReiTUTbqeehA/edit

Chrome Frame Synchronization -
https://docs.google.com/presentation/d/1q2WU0LusCyQFKDMjOSWLj3xGeOxMWmLzConrC8euJpA/edit#slide=id.g296ce4932_03860

Current Process

At any time;

- "Renderer" sends a SwapDelegatedFrame IPC message with a CompositorFrame data.
  - This process is totally async. A render could send thousands of SwapDelegatedFrame messages.
- The DelegatedFrameHost receives the SwapDelegatedFrame IPC message and forwards it to the SurfaceFactory.
- The SurfaceFactory calls into SurfaceManager and Surface
  - **The Surface does something random? Why does this exist?**
    - Surface sends a "Swap Ack" to the "renderer" via the DelegatedFrameHost?
  - The SurfaceManager checks if sequence has been satisfied. If so, tells the Display that is has been damaged.
- If the Display is damaged then;
  - SurfaceAggregator is used to create a CompositorFrame out of all the Surface's CompositorFrame.
  - The "Swap Ack" callbacks are sent to the "renderer" via the DelegatedFrameHost.
  - The composited CompositorFrame is rendered onto the real, onscreen output surface.

At any time;

- ui:Compositor calls Swap and submits an "empty frame" to SurfaceFactory?
- The SurfaceManager sees this frame and marks the Display as damaged.
- Somehow the browser now gets to draw?

Requirements

“I would like you to start rendering into this destination”

For scheduling to work, we need the following to occur;

1. The “surface client” can request that it needs to be updated.
2. We request a “surface client” to generate “a frame”.
    - **(Optional? - Can this work?)** The request includes “a frame” to use as the basis to generate into. This frame could be;
        i. The real “root frame”, or
        ii. A freshly minted, never before seen “frame”, or
        iii. A previously submitted “frame”.
3. The “surface client” responds to the request by submitting “a frame” (after updating its content).

- **(Optional? Enhancement)** A “now is a good time for any idle work” could be useful (like with the Blink scheduler).
- **(Optional? Enhancement)** A “now is a good time to do any prework” could be useful.

The system allows multiple “generate a frame” requests to be pending at any time. This could be used where pipelining is needed to hide IPC latency.

Strictly;

1. The “surface client” must **never** submit without a request.[1]
2. Once the “surface client” has submitted “a frame”, the surfaces system is now the owner of it.
    a. **(Optional? - Can this work?)** The “surface client” only gets “a frame” via the request mechanism.
3. The “surface client” must not do “rendering work” outside the requested period.

What does this change?

- “Surface clients” no longer need to manage frames at all.
    - The surface system gives you the frame to update.
    - The surface system will try and give you an existing frame to minimize the amount of work you need to do to update it.
    - The surface system manages if a client single, double or triple buffered depending on the needs of the system.

Alternative idea - “Swap Ack holding”

- Setting MAX_FRAMES_PENDING = 0? (Or should this be 1)
- When a renderer swaps, “hold onto” the swap ack until we want the renderer to

-----

[1] We could enforce this with a request containing an ID which must be included in the response.

Fundamentally;

- There are multiple surfaces
- A surface produces **Frame**s
- Surface should start producing a frame on a BeginFrame message.
    - Every BeginFrame message should have a corresponding Swap
- Surface notifies frame is ready with a Swap.
    - Surface must always Swap but can say “SwapAborted”

A display is the union of multiple surfaces.
A display has a set of properties, of which the “vsync interval” / “frame rate” at which the display refreshes.

Thus, each individual surface has state related to scheduling;

- What they are currently doing - IDLE or RENDERING
- The BeginFrameArgs they are currently using for processing.
- Model for how long rendering takes (IE an estimate of completion time)

The Scheduler has to deal with XXX types of tasks;

- Frame production for a given surface
    - Request a surface to produce a frame and wait for surface to produce said frame.
    - As there are normally multiple surfaces, there can be multiple of these tasks pending at any one time.
    - These tasks run on other threads, therefore;
        - Don’t block other tasks.
        - Can run in parallel.
    -
- Aggregate ready frames and display them on the screen. “Draw and Swap”.
    - This task runs on the scheduler thread, therefore when running, blocks scheduling.

Some limitations of Surfaces in their current implementation

- Surfaces only support 1 frame in the queue.
- The Browser “locks” its resources by submitting an empty frame to the surface. The lock is held between the Browser’s commit and draw. During this time, the DisplayScheduler cannot draw.
    - Removing this restriction could improve scheduling, but comes at the cost of memory since the browser needs to be double, rather than single buffered.
- Mojo has a bare-bones mojo_shell. We should replace it with the DisplayScheduler.

Buffering

(paraphrasing - needs fact checking - FIXME) To get content onto the screen, we store the content in a buffer. This buffer is represented by a “Surface” object which has a (tree of) CompositorFrame objects. At a given point in time, the Display will need updating. At that point the SurfaceAggregator is used to squash the tree of CompositorFrame objects onto the screen. Instead of generating new CompositorFrame objects every frame, we instead update them by applying a delta.

Browser UI single buffer hack

- The browser UI was originally just going to be another surface client like any other. This caused the browser UI to be doubled buffered which was a huge ?memory? regression. A hack was put in place where the browser UI instead draws to the “root” surface directly. This means that when the browser is drawing the UI we can not do any other updates to the output.
- This is actually just very similar to Partial Updates.

Partial Updates

https://code.google.com/p/chromium/issues/detail?id=419394
https://docs.google.com/document/d/1yvSVVgJ8bFyWjXGHpb8wDNtGdx8W5co7W0gbzjdFRj0/edit
(paraphrasing - needs fact checking) Partial updates is based on the idea that the closer temporarily two updates to a CompositorFrame occur, the less work that is needed. This means that double buffering increases the amount of work as it introduces more temporal delay between updating a CompositorFrame. If rendering is quick enough that we don’t need to double buffer and can get performance improvements.

Textures seem to be currently returned via MailboxOutputSurface::OnSwapAck ???
cc::ResourceProvider??

https://docs.google.com/document/d/1iWb1Y0ubpVZ0CKr1duA2EvQQfZd5mh1EE2Ha8rsJYnA/edit - Mac Partial Swaps

Choosing the Buffering Level

If a surface is single, double or tripled buffered - the surfaces client should not care. The choice is both a memory (we only have a limited amount of memory we can use) and time (if things are working well, then we can use cheaper primitives) scheduling decision.

For example;

1. 5 surface clients which all regularly draw within 1ms.
    - We are better of having a single *global* buffer and running each client sequentially (passing the single *global* buffer back and forth). This makes things faster too, as only the damaged areas of the screen ever need to be updated.

2. 1 surface client which has a huge standard deviation of rendering time greater than 16ms,
    - We want triple buffering because it decouples the rendering speed from the output speed.

    - As we only have a single surface client, triple buffering is probably a good use of memory too.

3. 3 surface clients which mostly around 10ms, but occasionally take more.
    - We need to run the clients in parallel, so we can’t use single *global* buffering like the first case.
    - We have multiple surface clients, so using triple buffer for each is probably going to require too much memory.
    - Thus we use double buffering.

4. 1 surface client which is extremely slow, 2 surface clients which are 5ms and one surface client which is always 1ms.
    - We can actually mix and match here.
    - We use triple buffering for the put the extremely slow client.
    - We use double buffering for the two normal client.
    - We just pass the global buffer to the fast reliable client.
    - *Note: This is actually what we could use for the “Browser UI single buffer hack” approach.*

We can also mix and match.

Possible “Policies”

“Policies” are heuristics which explain how you want something to operate. There is no “correct answer” and are always tradeoffs involved.

There are currently two primary locations that have been identified which are useful for policies to manipulate. An explanation of these “hooking” points is discussed later.

- Only allow n surfaces tasks to be running at the same time. (n == number of CPUs)
    - Requires some way of **restricting the task queue**.
- Prioritize the surface which is currently being interacted with.
    - This requires being able to **reordering task queue.**
    - Put the given surfaces tasks first.
- “Sin bin” for badly behaving surfaces which are slow.
    - This requires being able to **reordering task queue.**
    - Put the “sin bin” surface tasks after the good surface tasks.
- “Perfect frames” mode where all content aggregated together uses the same frame time.
    - Can be implemented through **task addition**.
    - Don’t add DrawAndSwap task until all surfaces are finished.
- Prioritize throughput.
    - Can be implemented through **task addition**.

    - Just add new tasks with the latest begin frame args when previous task finishes.
- Prioritize latency.
    - Can be implemented through **task addition**.
    - Only add new tasks on a BeginFrame messages.
    - Can improve throughput by adding “optional” tasks which are only done if we believe they can be finished by deadline.
- Prioritize battery (IE increase idle time).
    - Needs both **reordering task queue** and **task addition**.
    - This one is complicated, we want to reduce the number of tasks and we want to run all the tasks ASAP to increase the time we don’t have tasks pending.
- Individual surface “drop to 30fps / 15fps” mode.
    - Can be implemented through **task addition**.
- Global “drop to 30fps / 15fps” mode.
    - Can be implemented through **task addition**.

Reordering Task Queue

A policy can cause different behaviours to be observed by changing the order tasks are executed, maybe giving a certain type of task priority. This idea is also logically equivalent to “choosing which task to run next from the scheduling queue”.

Task Addition

A policy can change when things are being run by restricting which tasks are even added to the scheduling queue. This comes in two flavors;

- Only adding tasks to the queue dependent on other events.
- Looking at what tasks already exist in the queue and only adding new tasks when a given criteria is met.

Add Task Policy Examples

```
class Surface:
 bool processing() { return state_ == StateId::PROCESSING; }
 void SendBeginFrame(BeginFrameArgs args) { state_ = StateId::PROCESSING; … }
 void Swap???() { state_ = StateId::IDLE; … }

class Task:
 TimeTicks deadline; // Time when this task must be finished before.
 bool discardable;   // If this task can be discarded without consequence.
 TimeDelta run_time; // How long this task will probably take to finish.
 void start();
 void discard();
```

```
class ProduceFrameTask(Task):
 Surface target;      // Surface this task is for.
 BeginFrameArgs args;

class DrawAndSwapTask(Task):
 pass

class Running:
  void add(OwnerId id, Task t);
  void has_pending_task_for(OwnerId id);

class Queue:
  void add(OwnerId id, Task t);
  void has_pending_task_for(OwnerId id);
```

Add Task Policy - “Prioritize Throughput”

OnTaskFinish

```
  # Any surface which is idle gets a new render task.
  for surface in surfaces:
   if surface.processing() or queue.has_pending_task_for(surface):
    continue
   queue.add(surface, ProduceFrameTask(surface, self.latest_begin_frame_args))
```

OnBeginFrame

```
  self.latest_begin_frame_args = new_begin_frame_args
  queue.add(DrawAndSwapTask(deadline=new_begin_frame_args.deadline))
```

Add Task Policy - “Prioritize Latency”

OnTaskFinish

```
  # Any surface which is idle gets a optional catch up render task.
  for surface in surfaces:
   if surface.processing() or queue.has_pending_task_for(surface):
    continue
   queue.add(surface, ProduceFrameTask(
     surface, self.latest_begin_frame_args,
     discardable=True,
     deadline=self.latest_begin_frame_args.deadline - DrawAndSwap.estimated_run_time()))
```

OnBeginFrame(new_begin_frame_args)

```
  self.latest_begin_frame_args = new_begin_frame_args

  # Any surface which is idle gets a new render task.
  for surface in surfaces:
   if surface.processing() or queue.has_pending_task_for(surface):
    continue
   queue.add(surface, ProduceFrameTask(surface, self.latest_begin_frame_args))
  queue.add(display, DrawAndSwapTask(deadline=self.latest_begin_frame_args.deadline))
```

Add Task Policy - “Perfect Frames” (IE Wait for slowest)

```
OnTaskFinish
  # Wait until all the surfaces have finished, then DrawAndSwap
  for surface in surfaces:
   if surface.processing() or queue.has_pending_task_for(surface):
    # A surface is still pending, skip this BeginFrame
    return
  queue.add(display, DrawAndSwapTask(deadline=Now()))

OnBeginFrame(new_begin_frame_args)
  # Wait until all surfaces have finish
  for surface in surfaces:
   if surface.processing() or queue.has_pending_task_for(surface):
    # A surface is still pending, skip this BeginFrame
    return
  for surface in surfaces:
   queue.add(surface, ProduceFrameTask(surface, new_begin_frame_args))
```

Scheduler Base

```
OnSchedulerTick():
  while True:
   task = NextTaskPolicy(self.queue)
   if task is None:
    if not self.queue.empty():
     message_queue.AddDelayTask(OnSchedulerTick(), self.queue[0].task_time())
    return

   if task.is_discardable():
    if task.deadline < (now() + task.run_time()):
     task.discard()
     continue

   task.start()
```

SkPicture records on the compositor thread get turned into bitmaps on the GPU in one of two ways: either painted by Skia's software rasterizer into a bitmap and uploaded to the GPU as a texture, or painted by Skia's OpenGL backend (Ganesh) directly into textures on the GPU.

The Übercompositor performs all composition of both browser UI and renderer layers in a single drawing pass. Rather than drawing its quads itself, the Renderer hands all them to the browser where they are drawn in the location of a DelegatedRendererLayer in the Browser compositor's layer tree. For much more detail on this subject, see the übercompositor design doc.
https://docs.google.com/a/chromium.org/document/d/1ziMZtS5Hf8azogi2VjSE6XPaMwivZSyXAIIp0GgInNA/edit

From "A little bit about compositing and the GPU" at
http://www.chromium.org/developers/design-documents/oop-iframes/oop-iframes-rendering

> **The renderer sends an IPC to the browser when a new buffer is available and expects an ACK for every such message. This ACK contains a previously sent buffer to be reused by the GPU and a sync point (location in the GPU command buffer), or a NULL buffer, if nothing is available to be recycled.**
>
> Sync points are used to make sure the same buffer isn't being rendered into and drawn out of. When an embedding compositor issues the last draw call that uses the child renderer's image data to the GPU command buffer, it also inserts a sync point into the command stream and sends an ACK to the renderer with that number. Rendering into the recycled buffer is only allowed after the GPU processed all commands up to the sync point. Rendering is suspended until that happens.
>
> The Übercompositor
> The ubercompositor is being developed to improve rendering performance on Aura, by integrating compositing of UI and web content in the browser process. Under the new architecture, child compositors in each rendering process supply compositor frames to the parent compositor which are then composited into the screen for display.
>
> Each CompositorFrame contains a set of quads and associated texture references that are generated by a child compositor in a renderer process. Internally, TextureMailboxes are used to transfer textures to the browser process.

Whereas in the existing hardware path compositing mode the renderer sends a SwapBuffer message to the GPU to instruct it to display its uploaded texture to screen, in the ubercompositor architecture the renderer sends an IPC message directly to the browser containing the compositor frame. The parent compositor then interacts with the GPU process via IPC in order to do the final compositing.

Other related stuff;

Tile Prioritization Design -
https://docs.google.com/document/d/1tkwOlSlXiR320dFufuA_M-RF9L5LxFWmZFg5oW35rZk/edit#

Graphics Context Eviction -
https://docs.google.com/document/d/182Z4hVGa_lmDV5gYtzevyEelpgenTouqhLjh0eqkWsg/edit#heading=h.jcrsubd64icn

GPU Accelerated Compositing in Chrome -
http://dev.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome

Blink/Chrome GPU Rasterization: Phase 1
https://docs.google.com/document/d/1Vi1WNJmAneu1IrVygX7Zd1fV7S_2wzWuGTcgGmZVRyE/edit#heading=h.7gtvwy48curk

GPU Accelerated Rasterization: Phase 2
https://docs.google.com/document/d/1ZWOgutW0RxsTj7oatw7W04sdv_AnKZnMt3z3KikoDQo/edit

----

Two Modes

- Synchronous Rendering
  - *"I'll call you when I want content rendered." or "Please render me contents into this location."*
  -

- Asynchronous Rendering
  - *Here is a location. "Call me when you rendered something."*
  - Swap is used client->server to "notify that I've finished rendering".
    - Here is the "frame data" -> CompositorFrame which consists of Textures.
  - SwapAck is used server->client to "I've finished using the data you swapped".
    - Return any "frame data" for possible recycling.
  - Renders keep working while a Swap is a maximum number of Swap calls haven't received SwapAck messages.

DidSwapBuffers
DidSwapBuffersComplete