Surfaces

jamesr@chromium.org

Goals

Surfaces are a concept to allow graphical **embedding** of **heterogeneous untrusting** clients **efficiently** into one scene.

- **embedding** - the core concept of a surface is that it may contain references to surfaces from the same client or different clients
- **heterogeneous** - rendering to a surface does not require that a client use a specific library or toolkit, only that it be able to speak the surface protocol
- **untrusting** - a client does not have to trust its embedding or embedded clients in order to render into a surface. Having access to a surface from another client does not provide any access to that surface's contents.
- **efficiently** - rendering to a surface should have minimal overhead compared to rendering directly to the native screen and should be equally efficient regardless of the embedding depth.

Use cases

In Chromium, we can use surfaces for many of the embedding cases we have today:

- embedding a blink-rendered tab in the browser's UI
- embedding a plugin or video within a page
- embedding an iframe rendered from one process into an iframe rendered in a different process

Concepts

A **Surface** is a fixed size rectangle that can be rendered into by submitting frames. A client can create a surface by asking the SurfaceManager to construct one for a client (possibly itself) to render into. A Surface can be identified by two IDs generated by the SurfaceManager - one for identifying the surface when issuing frames to render to the surface, one to identify the surface when rendering from it to embed.

A **Frame** is a series of quads to draw, some of which may reference other surfaces. A frame also contains a set of resources and associated synchronization points. Here's a (rough) outline of the structure of a frame:

- List of prerequisite sequence numbers for the frame
- List of resources with associated synchronization points
- List of passes, each of which contains
    - Transform of the pass
    - Damage and output rects
    - List of quads within the pass, each of which has
        - Transform / rect / clip state / opacity / blend mode / etc (may be shared with other quads)
        - Material - maybe solid color, texture, surface, etc
        - Material-specific properties such as color, texture ID, surface ID

The act of submitting a frame generates an implicit sequence number that can be used to synchronize presentation with other frames, potentially submitted by other clients. A surface identifier + sequence number tuple unique identifies a frame for all clients of an instance of the service (and there will typically only be one surface service in the system).

A **Display** is a physical display (when Chromium is the operating system) or a native OS window object (when Chromium is running inside another operating system). The surface service provides a surface for each display for a designated client to issue frames to. In the case of Chromium running on windows, for example, the surface service would generate a surface identifier for each top-level HWND and provide them to the browser UI code to render into.

Of particular note is that on Mac we can construct a display wrapping an IOSurface for each tab and let CoreAnimation composite the tabs with the browser UI. This does not provide the bandwidth benefits of ÜberCompositor but it does allow everything outside of the browser process to use the same presentation path as platforms using Aura/ÜberCompositor and reduce a lot of platform-specific complexity in our code.

Processing model

For clients:

Whenever a client wants to update the rendering of a surface it owns, it generates a new frame and submits it to the SurfaceManager. This frame may contain quads that references surfaces being embedded by this client. A client does not have to issue a new frame whenever a surface it embeds updates its rendering. Issuing a frame also transfers ownership of resources (GL textures, software shared memory buffers).

Whenever a client wants to start embedding another client, it first generates an appropriately sized surface through the SurfaceManager and then sends it to the client. The embedding

client can start immediately issuing frames referencing the new surface or it may wait to receive an acknowledgement from the embedded client that the surface is ready, depending on the desired application semantics.

Resizing is analogous to creating a new surface.

For the service:

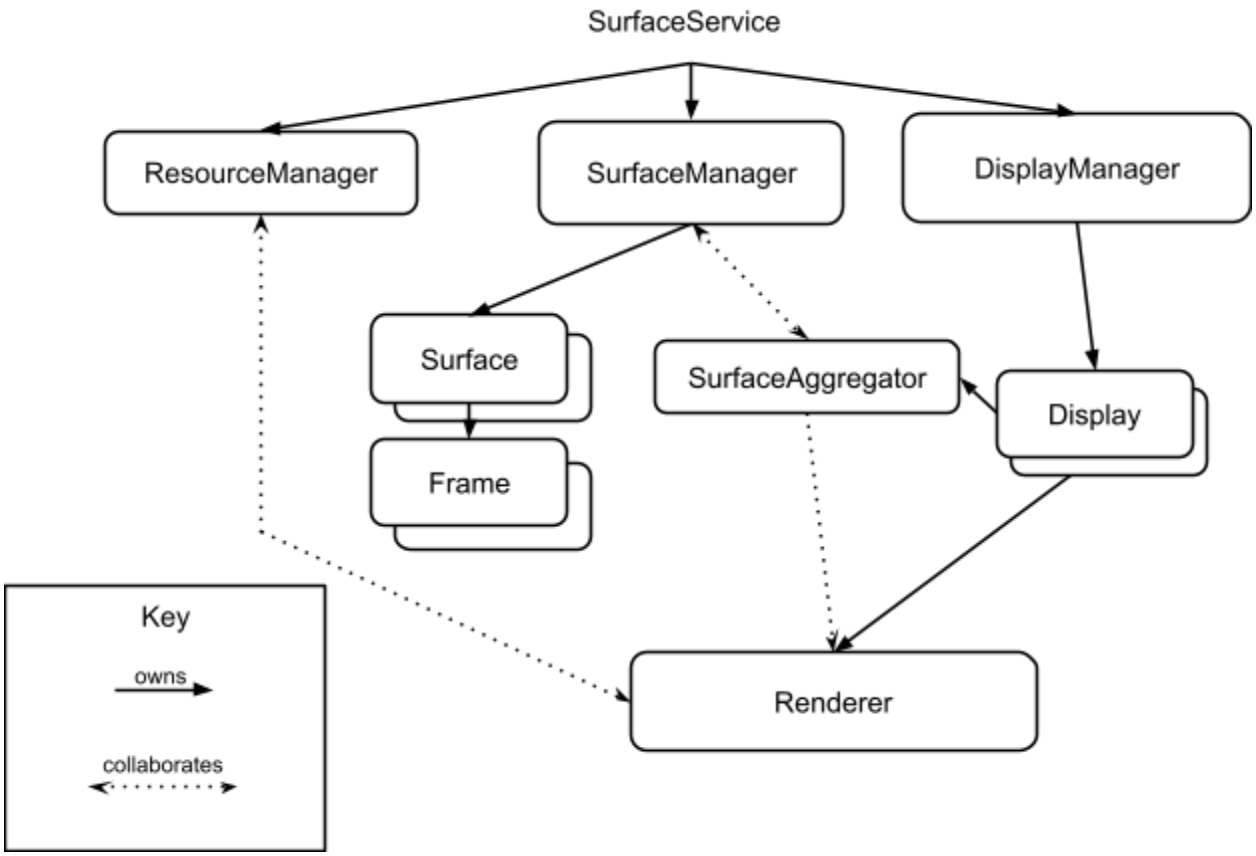
Whenever the manager receives a new frame, it performs some sanity checks on the frame (i.e. making sure it only references frames that the client should be referencing) and then saves it in the surface's potential frame list along. Whenever this frame's prerequisites are satisfied, it is moved into the eligible frame list for the surface. Only one frame can be rendered for a given surface at a time, but a client is allowed to pipeline multiple frames.

Whenever a display is ready for a new frame and something has changed, the service aggregates frames from the surfaces that contribute to that display and then renders them. The aggregation algorithm is simple:

1. Start with the most recent eligible frame issued to the surface associated with the display
2. Iterate through the quads in the frame in draw order, keeping track of the current clip and transform
    a. If the quad is not a reference to another surface, draw it
    b. If the quad is a reference to another surface, find the most recent eligible frame issued to the referenced surface
        i. If there is no such frame, or if this surface has already been visited in this aggregation (i.e. there's a cycle) skip the surface quad
        ii. Otherwise, recursively repeat this algorithm from step 2

When the service knows that it will never render from a given frame again - for instance if it has started rendering a newer frame for a given surface or if the embedding client has told the manager that it wants to destroy a surface - the service sends an acknowledgement to the client with a set of resources to return to the client along with associated synchronization points.

SurfaceService structure



The **SurfaceManager** keeps track of all surfaces created in the system. For each surface, it keeps track of:

- The client that created the surface and will be embedding (rendering from) the surface
- The client that will be rendering into the surface (may be the same as the creator)
- List of submitted frames for the given surface

The **ResourceProvider** keeps track of resources that the service has ownership of and how to return ownership to clients. For GL textures, for instance, this means managing mailboxes.

The **DisplayManager** keeps track of all displays that the surface service is responsible for rendering into. For each display, the DisplayManager owns a surface used to render into the display as well as some state for hit testing against surfaces and determining which surfaces contributed to the display's last produced frame.

There is only one instance of each of the Manager types in an instance of the service.

A **SurfaceAggregator** implements the aggregation algorithm and knows how to submit an aggregated frame to a renderer. Aggregators are (nearly) stateless and can be created whenever necessary.

A **Renderer** translates an aggregated frame into draw commands appropriate for a given display. In GPU rendering mode, this means GL draw calls and a swap into the display. In software rendering mode, this means skia calls into the appropriate SkCanvas.

Synchronization

Resource synchronization is the same as it is with ÜberCompositor, with the slight simplification that the pipeline depth is not influenced by the nesting level of the embedding.

Clients can optionally synchronize frames with each other using the prerequisite / postrequisite synchronization points. This has to be done with care but can be useful to do things like prevent resize guttering. 99% of the use cases in Chromium will not require any explicit synchronization between different surfaces - in nearly all cases it's perfectly fine (and desirable) to let clients render independently of each other.

Here's an example of a possible gutter prevention algorithm. Assume that client Alice is embedding client Bob and wants to resize its surface for Bob from 100x100 to 200x200. If Bob responds fast enough to Alice's resize message, Alice wants to make sure that Bob shows up at 200x200 in the same frame as Alice's decorations.

Start conditions:
Alice is embedding Bob. Alice owns a 100x100 surface that Bob is rendering into. Alice and Bob both have pending frames referencing the 100x100 surface.

Sequence for Alice:

- Alice decides to resize Bob to 200x200 and change decorations that Alice is rendering.
- Alice requests a new 200x200 surface from the SurfaceManager
- Alice sends Bob a resize request and a handle to the new 200x200 surface
- Alice starts a timeout
    - If Bob responds to the resize message before the timeout:
        - Alice issues the first frame referencing the 200x200 surface with a prerequisite sequence number that it got from Bob
    - If Bob doesn't respond before the timeout:
        - Alice issues a frame referencing the 100x100 surface and appropriate quads to stretch or gutter as appropriate
- Regardless of when the resize response comes in, Alice issues a destroy call for the 100x100 surface to the SurfaceManager after starting to issues frames referencing the 200x200 surface.

Sequence for Bob:

- Bob receives a resize message with the new surface identifier

- Bob issues a new frame appropriate for a 200x200 surface which generates a sequence number for the frame
- Bob sends a resize response to Alice with this sequence number

End conditions:
Alice and Bob are referencing a 200x200 surface
The SurfaceManager knows that the 100x100 surface can be destroyed as soon as the service no longer needs it.

If Bob is slow to respond, Alice may stall or submit one or more frames that gutter. However if Bob responds fast enough the service can guarantee using the sequence numbers that the new frame from Bob and the new decorations from Alice show up on screen at the same time.