Surface Synchronization API Guide (public version)

Originally by: fsamuel@chromium.org, August 2018
Updated by: jonross@chromium.org, April 2020

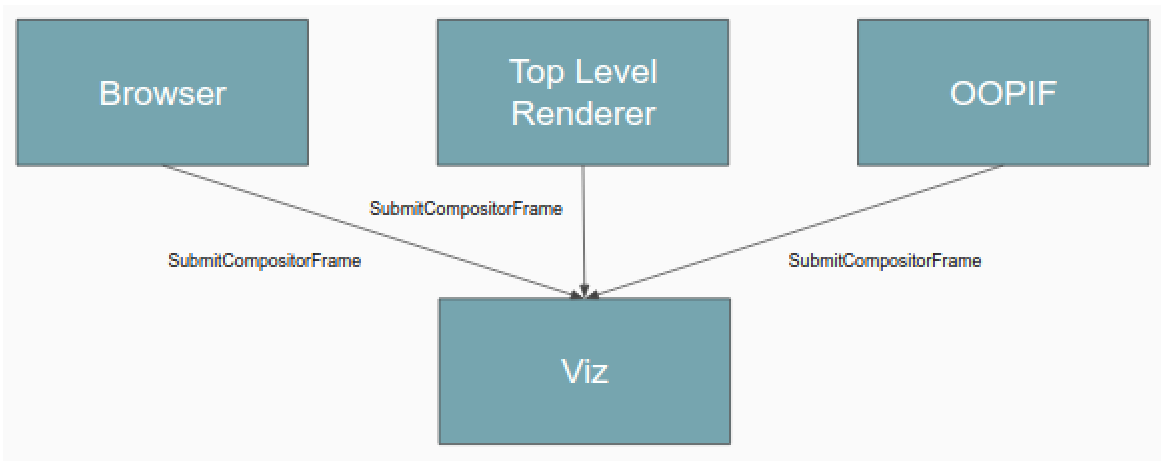
Overview
Frame Production without Surface Synchronization
Frame Production with Surface Synchronization
SurfaceId
Allocating a LocalSurfaceId
Understanding the SurfaceId Flow
Submission Flow
Embed Flow
Visualization
Frame activation
The SurfaceLayer
Background Color, Gutter or Stretching Content
Device Scale Factor Scaling
Deadline API
Deadline Inheritance
System Default Deadline
Late Arriving Dependencies
Maintaining Surface Synchronization Invariants
Multi-client Visual Snapshotting

Overview

The Chromium compositing system consists of two major components: the ***layer compositor*** *(called cc)* and the ***display compositor*** *(part of Viz)*. Viz clients typically host a layer compositor. The role of the layer compositor is roughly to build a scene graph of the content generated by a client and produce a final output frame, referred to as a CompositorFrame, that Viz consumes and composites into a single display frame.

Viz clients communicate with Viz via a CompositorFrameSink(Client) pair of interfaces.

Viz clients receive BeginFrame messages (typically in response to a system vsync signal), and respond asynchronously with either a CompositorFrame sent via a *SubmitCompositorFrame* or send back a *DidNotProduceFrame*, indicating they have no new content to display for the current BeginFrame. Viz provides a mechanism for backpressure: *DidReceiveCompositorFrameAck*. For each CompositorFrame submitted, Viz currently assumes a client will not submit an additional CompositorFrame until it receives a *DidReceiveCompositorFrameAck* for the previous frame.



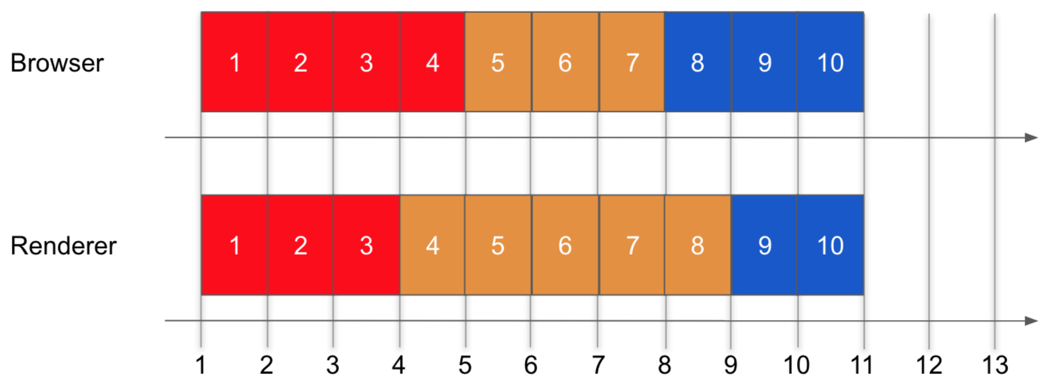
This diagram shows examples of various Viz clients in Chromium. Browser UI, the top level Blink renderer and out-of-process iframes are all Viz clients that submit CompositorFrames.

Viz's role is to collect CompositorFrames from all clients with updated content for a given vsync signal, and produce a single aggregated CompositorFrame for the display that is the composition of all the received CompositorFrames from all the clients with visuals to be displayed.

Prior to surface synchronization, there was no mechanism for Viz clients to coordinate and synchronize visuals amongst one another. Chromium depended on the fact that the browser UI thread hosted the display compositor and received CompositorFrames from various clients in order to synchronize browser UI with the top level renderer processes hosting Blink. This existing system did not support out-of-process iframes (as part of Site Isolation, Chromium's mitigation for Spectre). Surface synchronization was introduced in order to allow the display compositor to be moved out of the browser process and to enable site isolation as a first class citizen of the compositing system.

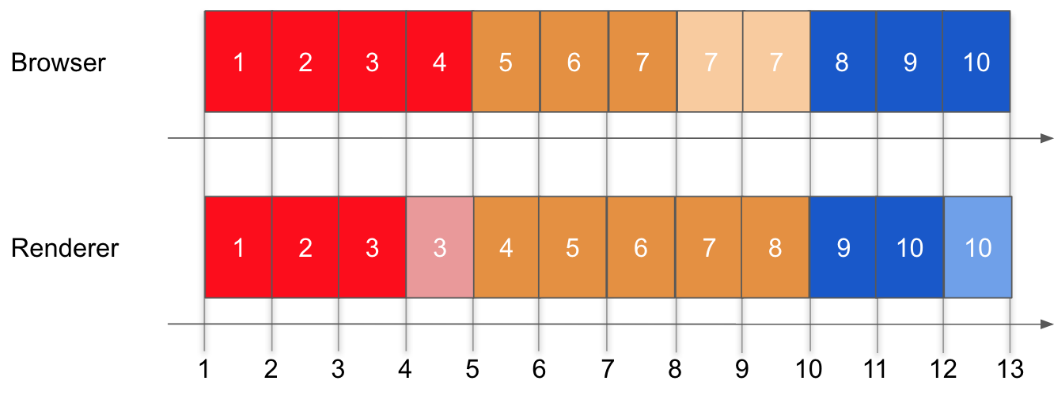
Surface synchronization works by creating dependencies called SurfaceIds across clients that are tracked by Viz. A parent client (also called embedder) can *embed* a child client by allocating a SurfaceId for the child and passing that SurfaceId down to the child so it can submit a corresponding CompositorFrame. The parent client's CompositorFrame will not be presented until the child submits its corresponding CompositorFrame (or until a parent-specified deadline passes). Likewise, the child's CompositorFrame will not be presented until the parent submits a new CompositorFrame referring to it.

Frame Production without Surface Synchronization



In this diagram, the x-axis corresponds to the start time of each frame, called BeginFrame. The three colors, red, yellow and blue correspond to visual properties that the browser wishes to synchronize: device scale factor, for example. Ideally, both the browser and the renderer should synchronously update their device scale factor. However, in this case, the renderer gets ahead of the browser at BeginFrame 4 and the browser gets ahead of the renderer at BeginFrame 8. See Figure below for how surface synchronizing resolves this.

Frame Production with Surface Synchronization



In this diagram, as above, the x-axis corresponds to the start time of each frame, called BeginFrame. In this case, surface synchronization is used to align frames across the browser and renderer. In particular, surface synchronization defers frame 4 from the renderer until BeginFrame 5 in order to give the browser an opportunity to catch up. Had frame 5 of the renderer arrived in time to go with that of the browser, then the renderer's 4th frame would have been skipped.

Furthermore, surface synchronization defers frame 8 from the browser by two BeginFrames in order to give the renderer an opportunity to catch up. The end result is the change in device scale factor appears to take effect atomically across clients. However, the time to change the device scale factor to blue jumped from nine BeginFrames in the previous diagram to ten BeginFrames with surface synchronization. The total time to complete the set of operations has increased. Synchronization is not a free operation, but the API has been designed to make it as cheap as possible. Sometimes synchronization can improve performance by avoiding generating spurious frames from a parent that would not be aligned with the child's visual properties.

SurfaceId

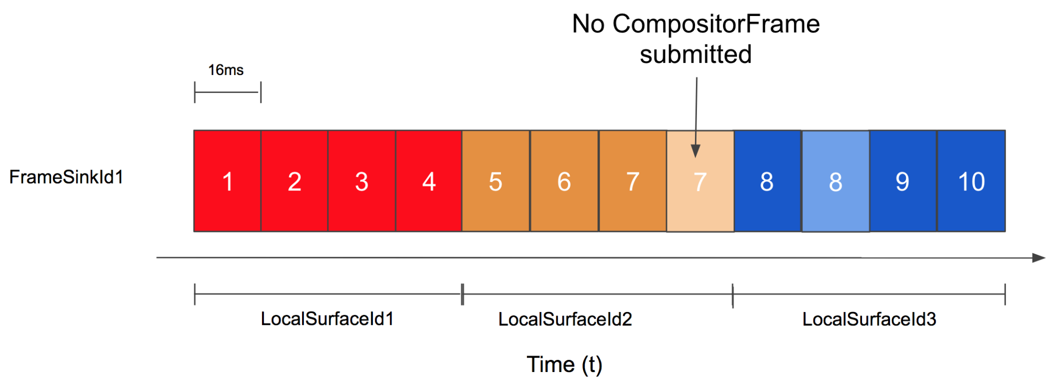
The SurfaceId is a fundamental component of the surface synchronization system. The SurfaceId is a globally unique identifier for a sequence of CompositorFrames from a particular client with a common set of properties such as *size* and *device scale factor*. The SurfaceId consists of two components: FrameSinkId and LocalSurfaceId. The FrameSinkId uniquely identifies a Viz client. The LocalSurfaceId uniquely identifies a sequence of CompositorFrames from that given client.

The LocalSurfaceId consists of three components that are key to synchronization:

- **parent_sequence_number:** A monotonically increasing number allocated by the embedder of the client.
- **child_sequence_number:** A monotonically increasing number allocated by the client identified by the FrameSinkId.
- **embed_token**: An unguessable token that uniquely identifies the parent/child embedding relationship. This identifier should change on reparenting.

Fundamentally, whenever a parent wishes to synchronize properties with its child or vice versa before producing new visuals, a new LocalSurfaceId must be allocated. The LocalSurfaceId serves as a fence that aligns parent and child CompositorFrames with a common set of properties. Note that synchronization events *defer* frame production and thus slow down the overall frame rate of both parent and child and so synchronization is expected to be a relatively rare event. As long as a client continues to submit CompositorFrames to a single LocalSurfaceId, it can produce frames at whatever rate is appropriate for its context.

Each client is expected to submit CompositorFrames to monotonically increasing SurfaceIds. In other words, if a client submits CompositorFrame to SurfaceId (2, 2), then the next valid surface the client can submit to must have a parent_sequence_number >= 2, child_sequence_number >=2 and either component strictly greater than 2. Thus, (3, 2), (2, 3), (3, 3), (4, 2), etc are all valid next SurfaceIds. (3, 1) is not a valid next LocalSurfaceId because it is not monotonically increasing on all components: the child component went backward.



This figure depicts ten CompositorFrames submitted to a single FrameSink of FrameSinkId1. When certain surface properties such as size change, the parent (or child) client will allocate a new LocalSurfaceId. Each surface has a unique SurfaceId = (FrameSinkId, LocalSurfaceId), and so three surfaces are depicted here: red, yellow, blue.

Allocating a LocalSurfaceId

Viz provides two utility classes to allocate LocalSurfaceIds: ParentLocalSurfaceIdAllocator and ChildLocalSurfaceIdAllocator. The expectation is the parent will allocate the first LocalSurfaceId for the child, and after the parent-child relationship is established (an embed_token is picked), the child is free to update its child component of the SurfaceId, while the parent can freely update its parent component. Assuming the usage of cc, when the parent allocates a new LocalSurfaceId and passes it to the child, the child is expected to call LayerTreeHost::SetLocalSurfaceIdAllocationFromParent.

When a child wishes to initiate synchronization, the new LocalSurfaceId will always be allocated on the impl (compositor) thread. If the main thread of the child would like a new LocalSurfaceId allocated in association with a particular operation, then it can call LayerTreeHost::RequestNewLocalSurfaceId. This triggers a commit if the child has a valid LocalSurfaceId from the parent and issues a request to the impl (compositor) thread to allocate a new LocalSurfaceId on the next SubmitCompositorFrame. Occasionally a new LocalSurfaceId must be allocated on the impl (compositor) thread in response to impl-initiated action: top bar scrolling, for example. In that case, a new LocalSurfaceId may be allocated even if not requested from LayerTreeHost.

Whenever the LocalSurfaceId changes from one frame submission to the next within cc, a RenderFrameMetadata object will be sent from the child to the parent. The RenderFrameMetadata contains the set of visual properties used to generate the last CompositorFrame.

Understanding the SurfaceId Flow

Surface synchronization can be understood as two flows (sequences of events) that start from a (Parent|Child)LocalSurfaceIdAllocator, fork off into *embed* and *submission* flows, and rejoin the first time a CompositorFrame corresponding to that new SurfaceId contributes to a display frame at aggregation time.

Submission Flow

A Submission Flow begins at LocalSurfaceId allocation time (whether at the parent or child allocator).

If the LocalSurfaceId is allocated by the parent, then the LocalSurfaceId propagates down to the child's main thread, triggers a commit and propagates to the child's compositor thread where the child will eventually submit a CompositorFrame to that LocalSurfaceId and ultimately the frame will be received by Viz.

If the LocalSurfaceId is allocated by the child, then that allocation begins at the impl (compositor) thread, and the next CompositorFrame submitted to Viz from the child will be to the newly allocated LocalSurfaceId.

Embed Flow

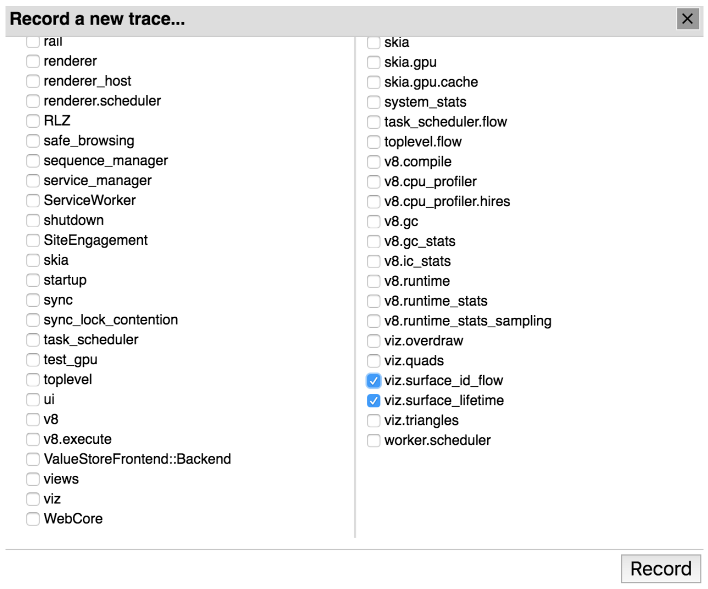
An Embed Flow begins at LocalSurfaceId allocation time (whether at the parent or child allocator).

If the LocalSurfaceId is allocated by the parent, then the newly allocated Id is embedded by a SurfaceLayer within the parent's layer (cc) compositor. The LocalSurfaceId propagates to the parent's impl (compositor) thread upon commit, and then will be incorporated in the parent compositor's next CompositorFrame. Viz will track the new SurfaceId (FrameSink/LocalSurfaceId) as an activation dependency (see Frame Activation). That CompositorFrame will be presented at aggregation time either when the dependency is resolved or a deadline specified at the time the SurfaceId is embedded in the SurfaceLayer (See the SurfaceLayer). The first time a new Surface is involved in aggregation is called the First Surface Embedding.

If the LocalSurfaceId is allocated by the chIld, then the newly allocated Id is propagated up to the parent client and embedded into a SurfaceLayer and the rest of the flow is similar to the above.

Visualization

The SurfaceId flow can be visualized in Chromium from about:tracing.



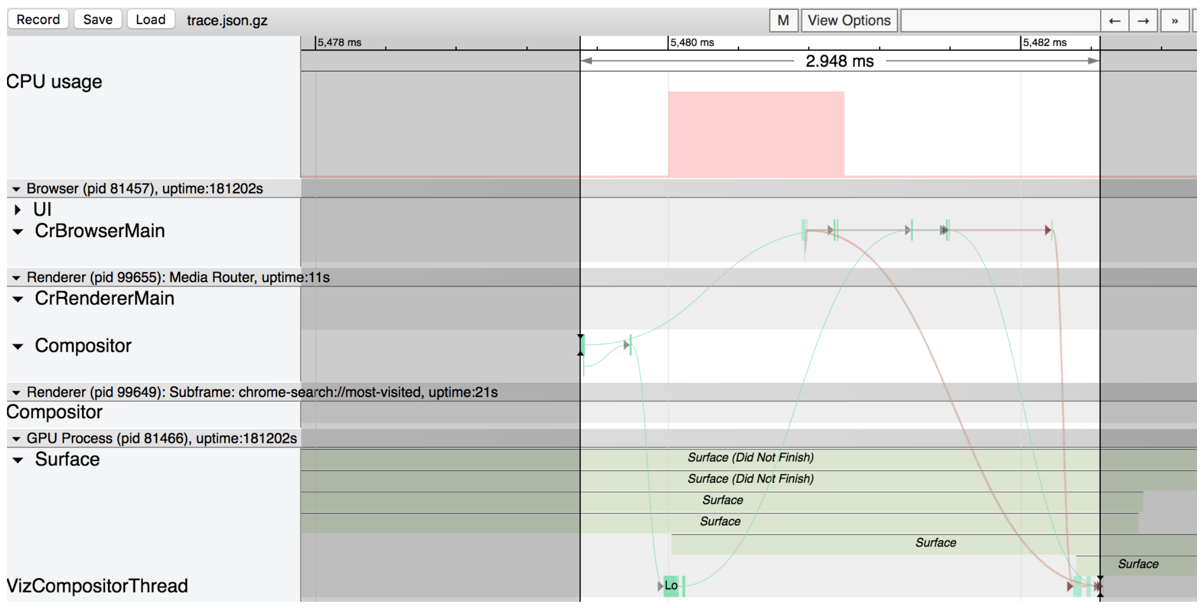
Record a new trace...

rail
renderer
renderer_host
renderer.scheduler
RLZ
safe_browsing
sequence_manager
service_manager
ServiceWorker
shutdown
SiteEngagement
skia
startup
sync
sync_lock_contention
task_scheduler
test_gpu
toplevel
ui
v8
v8.execute
ValueStoreFrontend::Backend
views
viz
WebCore

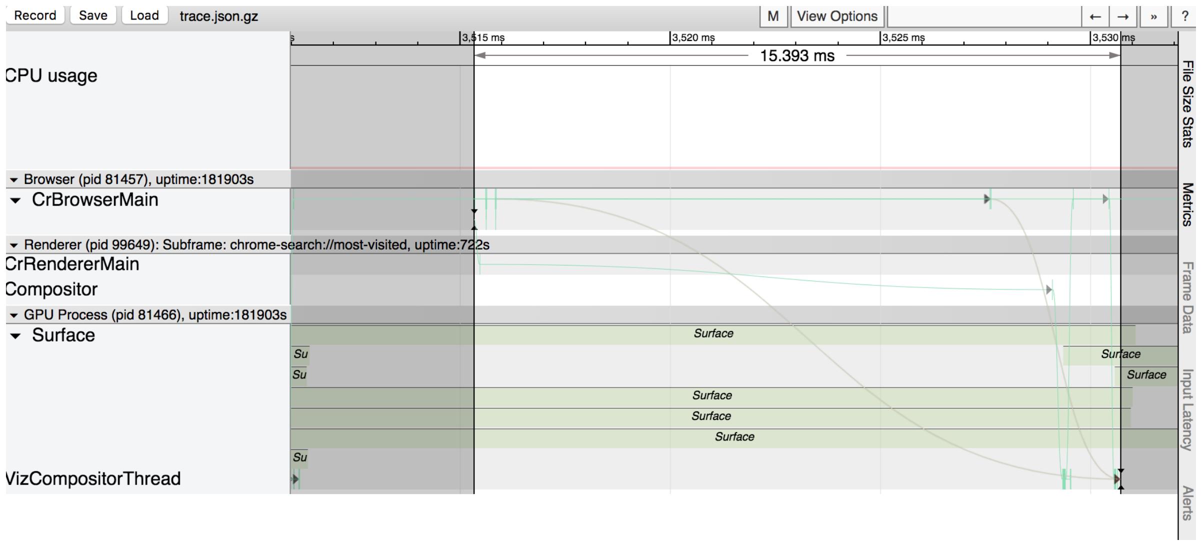
skia
skia.gpu
skia.gpu.cache
system_stats
task_scheduler.flow
toplevel.flow
v8.compile
v8.cpu_profiler
v8.cpu_profiler.hires
v8.gc
v8.gc_stats
v8.ic_stats
v8.runtime
v8.runtime_stats
v8.runtime_stats_sampling
viz.overdraw
viz.quads
[✓ viz.surface_id_flow]
[✓ viz.surface_lifetime]
viz.triangles
worker.scheduler

Record

To visualize SurfaceId flow in Chromium, go to about:tracing, click on Record, click on Edit Categories, unselect all the default Record Categories and select 'viz.surface_id_flow' and 'viz.surface_lifetime' in the 'Disabled by Default Categories'.



This diagram demonstrates a child-initiated surface synchronization event (auto-resize). In this case, the LocalSurfaceId is allocated on the Renderer's (child's) impl (compositor) thread and the LocalSurfaceId is passed up to the parent compositor where it is embedded in a SurfaceLayer and ultimately the parent submits a CompositorFrame to Viz. The end of this flow is 'FirstSurfaceEmbedding' where the two flows join at aggregation time. The entire synchronization time is 2.948ms.



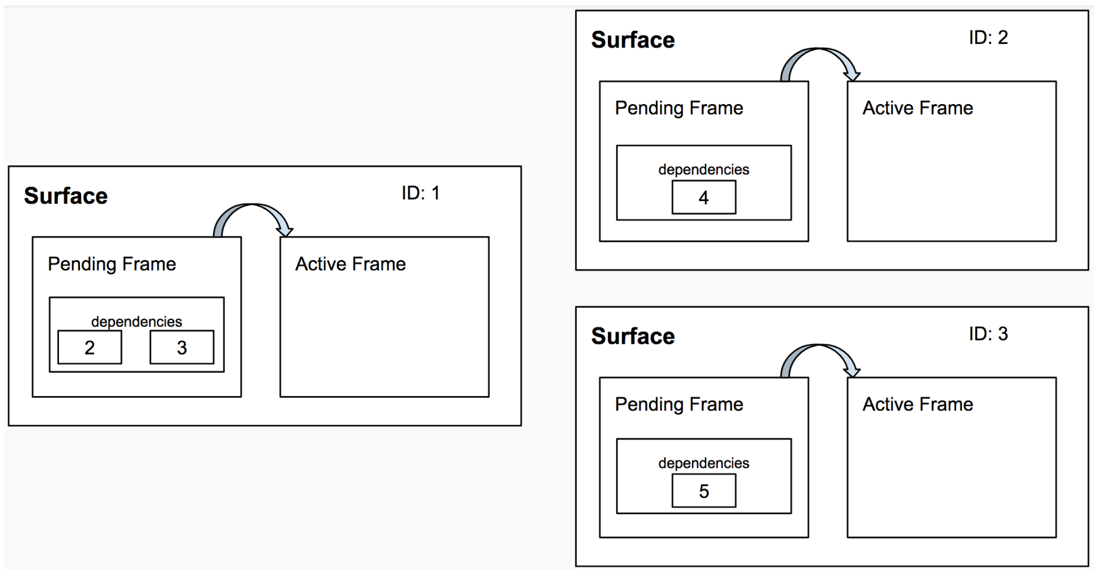
This diagram demonstrates a parent-initiated surface synchronization event (window resize). In this case, the LocalSurfaceId is allocated on the browser UI thread, is embedded in a SurfaceLayer and is passed down to the Renderer (child) mian thread which then commits to its impl (compositor) thread. Ultimately, the child submits a CompositorFrame to the newly allocated LocalSurfaceId. This whole operation completes in just 15.393ms. This demonstrates some of the additional parallelism achieved with surface synchronization. Prior to surface synchronization, the parent would not embed the new surface until AFTER the child has submitted a CompositorFrame to it. Thus, it would not be possible to submit a new frame, and embed it in the same frame interval.

Frame activation

A CompositorFrame is called ***active*** if it is a candidate for display: it can be used at display frame generation time.

A CompositorFrame submitted to a surface is said to be ***pending*** if it references other surfaces (called ***activation dependencies***) that either are not yet known to Viz or do not yet have CompositorFrames that are themselves active. A CompositorFrame is ***activated*** once either all its activation dependencies are active themselves or a deadline passes and Viz forcibly makes the frame available for display despite missing dependencies.

SurfaceId is a monotonically increasing identifier, and so Viz will assume that if a CompositorFrame for SurfaceId 2 arrives before SurfaceId 1 (both surfaces sharing the same FrameSinkId), then SurfaceId 1 is never going to arrive. Any CompositorFrame that depends on SurfaceId 1 will drop its dependency on that surface. This may then trigger an activation.



In this diagram, Surface 1 has a pending frame that depends on Surface 2 and 3. Surface 1 is not active because the CompositorFrames in Surface 2 and 3 are not active. Surface 2 depends on Surface 4, and Surface 3 depends on Surface 5, respectively, neither of which yet exist.

This case can happen if a client implements a throttling system for visual updates such as resize. For example, a client may respond to Resize 1 and receive a Resize 2 and Resize 3 while Resize 1 is being performed. It may choose to drop Resize 2 and skip straight to Resize 3 in order to catch up to the resize requests. The parent client may submit a CompositorFrame that depends on Resize 2 in its child. Once Viz sees a CompositorFrame for Resize 3, it will drop the dependency on Resize 2 and activate the parent client's CompositorFrame. By dropping activation dependencies that never arrive, surface synchronization is resilient to client-side throttling.

The SurfaceLayer

In the Layer Compositor (cc), a SurfaceLayer is a type of layer that specifies references to globally unique SurfaceIds. A SurfaceLayer is said to ***embed*** a surface of a given SurfaceId if it refers to that SurfaceId.

```
class SurfaceLayer : public Layer {
 public:
  void SetPrimarySurfaceId(const viz::SurfaceId& surface_id,
                           const DeadlinePolicy& deadline_policy);
  void SetFallbackSurfaceId(const viz::SurfaceId& surface_id);
  ...
};
```

A SurfaceLayer can be thought to embed a range of surfaces, as opposed to a single surface.

The primary SurfaceId is an activation dependency, or in other words, the CompositorFrame submitted that refers to this SurfaceId cannot be used until the client that submits to the activation dependency itself has an active CompositorFrame (where its dependencies are available) or a deadline hits as specified by DeadlinePolicy.

The fallback SurfaceId is the last SurfaceId known to the parent client to have been submitted by the child and activated. That way, in the event of a deadline passing, ***the fallback SurfaceId is guaranteed to be available for display***. viz::SurfaceRange tracks a set of related submitted surfaces, automatically providing a fallback Surface if it is required.

At surface aggregation time (the time to generate a single CompositorFrame for the entire display), the surface synchronization system will treat the fallback and primary surface IDs as an inclusive range of surfaces. The system will pick the latest surface known in the specified range. If the primary surface ID and fallback surface ID have differing FrameSinkIds then the system has no way of knowing the ordering relationship between surface IDs and so it will always embed the fallback SurfaceId in the event the primary is unavailable.

Background Color, Gutter or Stretching Content

The expected size of an embedded surface is the size of the primary surface. If the primary surface is unavailable at aggregation time, the chosen fallback may not match the size of the primary surface. If the fallback is too large, Viz will clip the surface content at aggregation time. If the fallback content is too small, Viz will fill in the gaps (gutters) with the background color specified on the SurfaceLayer.

Alternatively, the surface synchronization system provides the option to stretch content to fit the bounds of a SurfaceLayer. cc::SufaceLayer::SetStretchContentToFillBounds can be set. In this mode, gutter and clipping is not possible. Instead, fallback content (and primary content for that matter) will always be scaled to fit the bounds specified by the SurfaceLayer.

Device Scale Factor Scaling

Moving a window with multiple asynchronous clients distributed across a number of processes to another display with a different device scale factor presents a subtle synchronization problem. Clients must synchronize their scale factors atomically or else a very jarring effect will occur where content jumps back and forth until it settles on a device scale factor. Note that surface synchronization does not improve the situation where a single client is displayed concurrently across multiple displays with different device scale factors.

Surface synchronization can be used here as well, but the fallback behavior discussed above is inappropriate in this case. In the case of a mismatch of device scale factor between active parent and child surface content at aggregation time, the surface synchronization system will automatically scale the child content by the parent/child scale ratio guaranteeing stable device scale factor changes across clients.

Deadline API

Surface synchronization provides a rich deadline API for parent clients.

Ultimately, from Viz's perspective, each CompositorFrame has a single FrameDeadline. The FrameDeadline is effectively the maximum of all the specified SurfaceLayer deadlines.

Deadlines are always aligned to BeginFrames and so a FrameDeadline consists of a *start time*, a *deadline in frames*, a *frame interval* and a "*use_default_lower_bound_deadline*" bit. If the lower bound deadline bit is specified, then the deadline in frames used to wait for activation dependendencies is the std::max(system_default_deadline, FrameDeadline::deadline_in_frames).

A deadline specified by SurfaceLayer is ***one shot***, meaning that only one CompositorFrame will block on an activation dependency for a call to SetPrimarySurfaceId. Per-SurfaceLayer deadline policy is specified by the cc::DeadlinePolicy object. There are currently four different policies for setting a deadline on a SurfaceLayer:

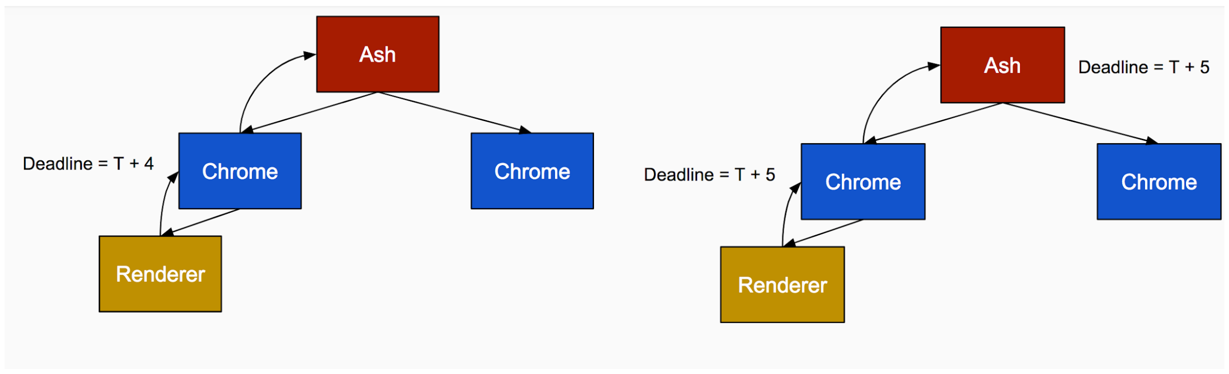
- **UseExistingDeadline**
  - Using the same policy as specified previously. SetPrimarySurfaceId may be called multiple times before commit. Following commit, the deadline is reset to zero. The UseExistingDeadlne policy indicates that the call to SetPrimarySurfaceId should not modify the current deadline being tracked by SurfaceLayer.
- **UseDefaultDeadline**
  - This policy indicates that the SurfaceLayer should use the system default deadline for the call to SetPrimarySurfaceId. This is typically 4 frames but may vary depending on the system.
- **UseSpecifiedDeadline**
  - This policy indicates that the SurfaceLayer should use a deadline specified in frames.
- **UseInfiniteDeadline**
  - This policy indicates that SurfaceLayer should block on the primary surface ID until it activates.

Deadline Inheritance

When a parent client specifies a deadline, it is making a statement to the surface synchronization system that it is willing to *defer its CompositorFrame for some number of BeginFrames without affecting the user experience*.

A child client may have a shorter (or longer) deadline than the parent. In either case, that is not ideal. In the case of a longer deadline, the child may have content that fills the bounds specified by the parent but is not used at aggregation time. In the case of a shorter deadline, the child client may show fallback content even though it had time (as specified by an ancestor) to wait for subcontent.

The solution to both of these problems is *deadline inheritance*. With deadline inheritance, a pending CompositorFrame inherits the deadline of its pending ancestors. Deadline inheritance occurs automatically within the surface synchronization system and is not currently configurable.



In this example, Chrome submits a CompositorFrame with a deadline of 4 frames. A BeginFrame later, Ash, the window manager submits a CompositorFrame with a deadline 4 frames AFTER its frame. As Ash embeds Chrome, the Chrome CompositorFrame inherits the deadline specified by Ash giving the system more time to produce the right frame.

System Default Deadline

The default system deadline is 4 frames with the frame interval determined from BeginFrameArgs supplied to the client by a BeginFrame message. A frame interval is determined dynamically at runtime based on the BeginFrame source. This parameter can be customized via command line.

--run-all-compositor-stages-before-draw will ignore all client specified deadlines and always wait until all activation dependencies are available in Viz. Note that this mode continues to be resilient to client-side throttling despite providing unlimited deadlines. See Frame Activation for more information on client-side throttling. This mode is used by Headless Chrome and Layout Tests.

Alternatively, --deadline-to-synchronize-surfaces=# can specify the number of frames to wait when ***cc::DeadlinePolicy::UseDefaultDeadline()*** is specified by a client.

Late Arriving Dependencies

If a particular CompositorFrame is forcibly activated after hitting a deadline, then all its dependencies are considered late. A forfeited dependency may arrive at a later point in time and may have its own set of dependencies. Blocking a late CompositorFrame on other, even later CompositorFrames compounds the problem of client tardiness.

To address the problem above, the surface synchronization system tracks late dependencies UNTIL the parent client submits a new CompositorFrame that then activates. Tardiness is a recursive property that propagates downward in the surface hierarchy. If an ancestor client arrives and is considered late, then any immediate children and their subtrees are also considered late and will activate immediately, if they haven't already.

Maintaining Surface Synchronization Invariants

In order to provide any sort of guarantee of synchronization, the surface synchronization system actively maintains a set of hard invariants. Any misuse or abuse of the surface synchronization API will result in either dropped frames or a terminated connection to Viz and report a "Surface Invariants Violation". Additional invariants may be introduced into the system in the future as needed to strengthen synchronization guarantees.

A FrameSink:

- Has a single fixed client for the entirety of its lifetime.
- Will never activate Surfaces out-of-order: two surfaces will never have pending CompositorFrames in flight at the same time.
  NOTE: In order to preserve this invariant clients MUST wait for a DidReceiveCompositorFrameAck prior to submitting the next CompositorFrame.
- Will never accept CompositorFrames submitted to surfaces older than the current surface.

A Surface:

- Has a fixed non-zero size.
- Has a fixed device scale factor > 0.
- Has a valid SurfaceId
  - FrameSinkId is valid if at least one of its two components (client_id, sink_id) are non-zero.
  - LocalSurfaceId is valid if all components are non-zero.

A CompositorFrame:

- Has the same size as the surface receiving it.
- Has the same device scale factor as the surface receiving it.
- Has at least one RenderPass.

The implication in the above invariants is if a client intends to change its size or device scale factor, it **MUST** allocate a new LocalSurfaceId.

Multi-client Visual Snapshotting

The surface synchronization system in conjunction with the CopyRequest system is capable of taking a consistent visual snapshot across all clients beneath a particular root. Suppose a privileged client, such as the browser, wishes to take a visual snapshot of all clients within the browser after a certain internal state change in some of the clients.

This can be accomplished via two steps:

1. The privileged client allocates a new LocalSurfaceId for the topmost surface to be captured, which, in turn, allocates new LocalSurfaceIds for all its children and so on recursively.
2. The privileged client can use the HostFrameSinkManager::RequestCopyOfOutput API specifying the newly allocated topmost LocalSurfaceId as the first surface that the client would like to capture from. Other clients can request snapshots of a surface through the privileged client.

The two steps above guarantee that the captured snapshot will incorporate all changes to all clients up until the moment the capture process was initiated.