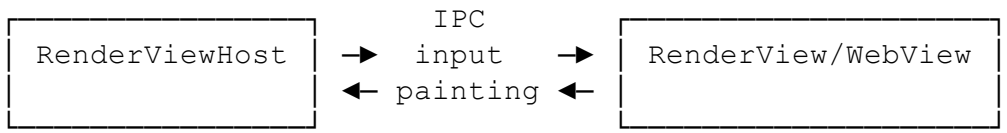
Background

This document covers the changes in input/painting plumbing to support out-of-process <iframe>s (OOPI).

In Chrome/Blink, input, layout, and painting are coordinated through the Widget hierarchy. In the browser process, this is represented by the RenderWidgetHost interface. In the renderer process, this is represented by the RenderWidget / WebWidget interfaces.

Pre-OOPI Architecture

Each WebContents has a RenderViewHost in the browser, and a corresponding RenderView / WebView in the renderer. RenderViewHost inherits from RenderWidgetHost and RenderView / WebView inherit from RenderWidget / WebWidget respectively. Since there is only one[1] RenderWidget/WebWidget object per WebContents, IPC messages for these functions are simply routed at the View-level:



In the renderer, since all frames in the tab are in the same process:

- Input events can be dispatched to the correct target, since all the information required to track focused frames and perform hit testing across multiple subframes is in the same address space.
- Layout/painting can be done for the tab and all its subframes, since the layout tree for all subframes is in the same process. When the renderer needs to generate a compositor frame, it can walk through the entire frame tree and send the final result back to the compositor.

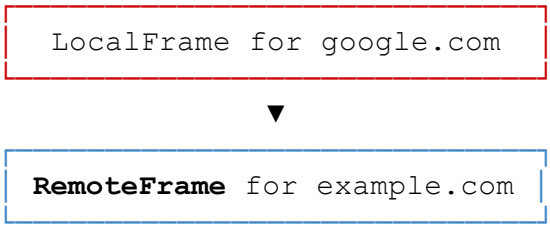
[1] This isn't completely true, but close enough for the purposes of this document. See the yet-to-be-written section on RenderWidget for more information.

Problems for OOPI

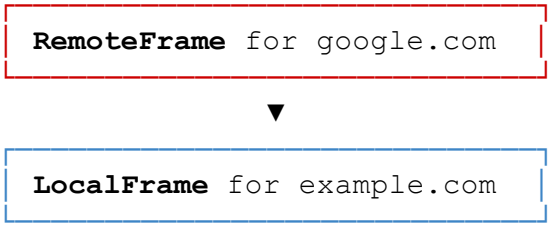
With OOPI, it's no longer sufficient to just route IPCs at the view-level. Consider a simple page:

With OOPI, it's possible for the main frame and subframe to be rendered in separate processes. Each renderer contains the full frame tree, with placeholder frame objects for frames that are in another process.

Frame tree in google.com renderer:



Frame tree in example.com renderer:



LocalFrame is a renderer concept, and is used for frames that are rendered in the same process. It has a LocalDOMWindow and Document; the corresponding DOM nodes are present, it has a v8 context for script execution, etc.

> **Important**! Frames that are same-origin will always be rendered in the same process. The same-origin policy means that same-origin frames can reach into other same-origin frames and synchronously mutate their DOM, so that state must always be maintained in process.

RemoteFrame is a placeholder in the renderer process for frames that are rendered in another process. It has a RemoteDOMWindow but it has no Document and no DOM nodes. It is simply a proxy for operations that can cross origin boundaries, like window.postMessage()[2].

This poses several challenges for the View-focused model of input / layout / painting:

- Some input events are routed to the focused frame. Suppose the subframe is focused: simply routing it to the corresponding RenderView / WebView means the input event will end up in the main frame renderer. This renderer doesn't know anything about the subframe's DOM nodes, v8 context, etc., since all that state is in the subframe renderer.
- Other input events require hit testing. If the target is inside the subframe, the main frame renderer can determine that the target was *something* in the subframe, but nothing more than that. Even if it could determine which DOM node the event should target, all that state is still in another process.
- Layout and painting are broken, since the main frame renderer won't know what to paint for the content for the <iframe> element holding the subframe.
- In addition, the subframe doesn't know where it's positioned in its parent frame, since the layout tree of the main frame is inaccessible.

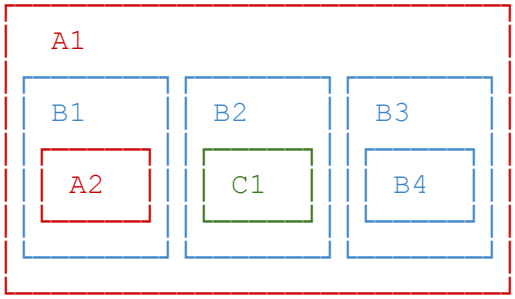
Some of these problems can be mitigated somewhat: for example, the main frame renderer can pass the problematic input events back to the browser to be re-dispatched to the subframe renderer. However, this introduce additional problems: deeply nested frames could cause a lot of ping-ponging between the browser and renderer processes, with the associated IPC latency, finding the focused frame might involve chasing the focus chain through multiple renderer processes, etc. Finally, this doesn't help at all with layout and painting.

[2] https://html.spec.whatwg.org/multipage/browsers.html#security-window is a list of all properties that are accessible cross-origin.

Architecture Changes for OOPI

In OOPI, input / layout / painting are still coordinated through the Widget hierarchy, but a tab may now have one or more Widgets. Each local root in a tab has a widget associated with it to manage input / layout / painting. In Blink, this widget is represented by the new WebFrameWidget class, while the Chrome side reuses the current RenderWidgetHost / RenderWidget classes.

A local root is a **LocalFrame** that does not have a **LocalFrame** parent. For instance, in a more complicated page:



There are a total of 6 local roots in this example:
Renderer A has 2:
A1 is a **LocalFrame** in renderer A and has no parent.
A2 is a **LocalFrame** in renderer A, and its parent frame (B1) is a **RemoteFrame**.

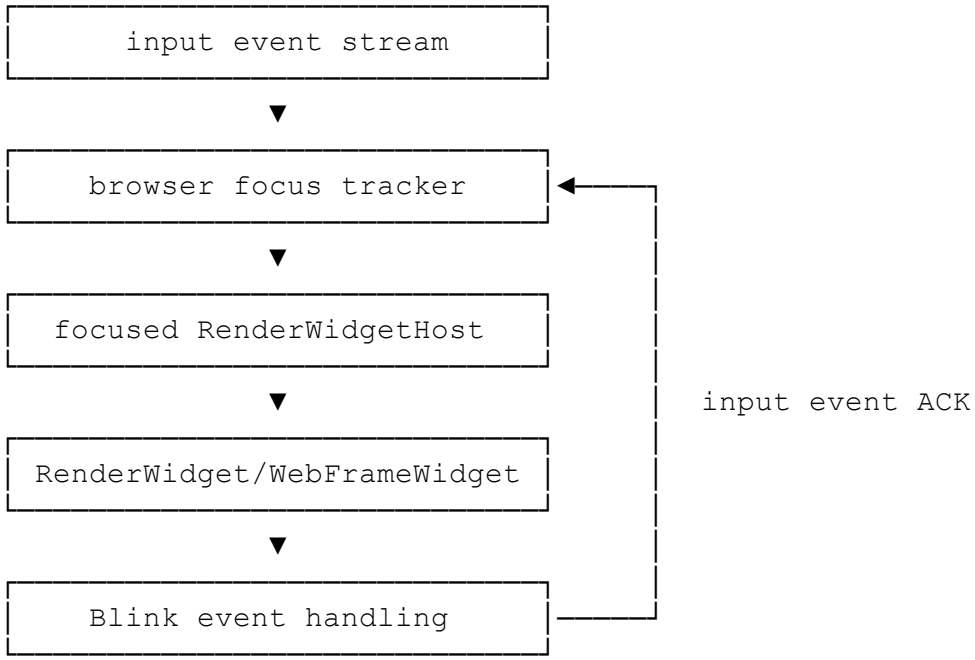
Renderer B has 3:
B1, B2, and B3: each one is a **LocalFrame** in renderer B with A1 as the **RemoteFrame** parent.

Renderer C has 1:
C1 is a **LocalFrame** in renderer C, and its parent frame (B2) is a **RemoteFrame**.

B4 is not a local root: while it is a **LocalFrame** in renderer B, its parent frame (B3) is a **LocalFrame** in the same process.

Input / layout / painting operations now have a browser-side component in addition to the work in the renderer.

For input events that are routed to the focused frame, the browser tracks the RenderWidgetHost that is focused. An incoming input event is first directed to the focused RenderWidgetHost:



Once the input event is sent to the RenderWidget / WebFrameWidget, Blink uses the existing plumbing for tracking focused frame / focus node to dispatch the input event as it did in the pre-OOPI world. Once Blink finishes handling the event, it sends an input event ACK back to the browser.

Since event handling can cause the focused frame to change, this ACK can cause the browser process to update the focused widget if needed. Thus, the browser process must wait for each input ACK before dispatching the next input event in the queue.

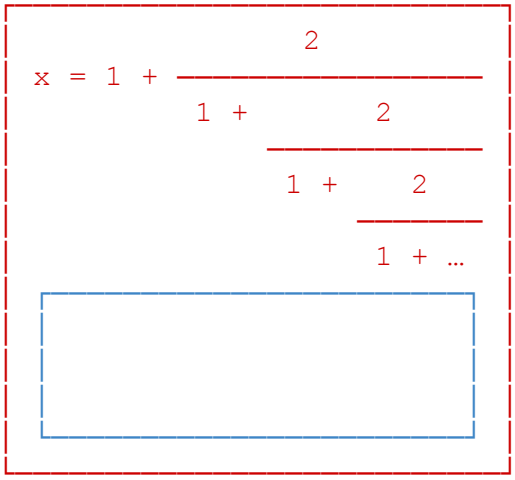
Input events that require hit testing follow a similar process. The browser process hit tests[3] using surfaces. This hit testing understands transforms on subframes and CSS properties like pointer-events: none[4], so it can find the target RenderWidgetHost. This event gets passed to the corresponding RenderWidget / WebFrameWidget. Since the definition of local root guarantees a contiguous chain of **LocalFrame** objects between the local root and the actual target in that renderer, Blink's event handling can all be done in process: events[5] do not bubble across frame boundaries.

---

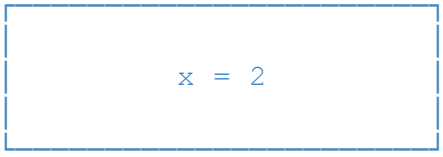
[3] This is currently being implemented in https://codereview.chromium.org/1265013003.
[4] This is not actually implemented yet, but will be.
[5] With the exception of scrolling.

Finally, layout and painting rely on browser-side compositing. A WebFrameWidget knows its viewport size, so it can layout and paint its contents without relying on other renderers. If the WebFrameWidget needs to paint a subframe that is a **RemoteFrame**, it reserves space for it during layout but doesn't paint anything.

For example, given a main frame with size 800x600) with a remote child with size 700x200), the compositor frame generated for the main frame would be an 800x600 texture that might look like this:



The placeholder for the 700x200 child is simply left blank in the main frame's compositor frame. The compositor frame generated for the child frame would be a 700x200 texture that might look like this:



The browser process receives these two compositor frames, combining the compositor frame from the child into the reserved space (applying any transforms, etc.) to produce the final result displayed: