Fullscreen with out-of-process iframes

Alex Moshchuk <alexmos@chromium.org>
3-4-2016

Links

HTML5 spec: https://fullscreen.spec.whatwg.org/
Tracking bug: https://crbug.com/550497

Objective

Currently, fullscreen does not work correctly with out-of-process iframes. For example, when pressing the fullscreen button on a Youtube video embedded in an OOPIF, the whole tab does fullscreen instead of just the video. The goal of this document is to explain what's broken today and propose/discuss solutions.

Background: browser side

There are two separate fullscreen codepaths in Chromium: **Flash fullscreen** and **HTML5 fullscreen**. Some more detailed information about how they work can be found in this Chromium design doc (a bit outdated); below is a quick overview.

Flash fullscreen uses `ViewHostMsg_CreateFullscreenWidget` and `ViewHostMsg_ShowFullscreenWidget` IPCs to

- create a new widget (`RenderWidgetHost` and `RenderWidgetHostView`)
- save it in the current `WebContents` using `fullscreen_widget_routing_id_`
- call into `FullscreenController::EnterFullscreenModeForTab()` in the `chrome/` layer, which will tell the `BrowserWindow` to expand to fill the screen.

`views::WebView`, the child view within `BrowserWindow` for the actual tab contents, will receive a `WebContentsObserver::DidShowFullscreenWidget()`, making it:

- look up the fullscreen widget via `WebContents::GetFullscreenRenderWidget()`, which uses `fullscreen_widget_routing_id_` and the main frame's process ID.
- swap its normal native view (for current WebContents) with the fullscreen native view (for the Flash widget).

Once the fullscreened view is resized and ready, a `ViewMsg_OnResize` is sent to renderer, which will tell the renderer that fullscreen was entered (`is_fullscreen_granted` in `ResizeParams` will become true) and pass fullscreen dimensions.

HTML5 fullscreen allows arbitrary DOM elements to be fullscreened. It reuses some, but not all of Flash fullscreen plumbing. It is initiated differently: the frame containing the DOM element to be fullscreened, sends a `FrameHostMsg_ToggleFullscreen` to enter fullscreen mode, passing an `enter_fullscreen` bool. This bubbles up to `WebContentsImpl::EnterFullscreenMode`, which then follows the Flash path above (`Browser::EnterFullscreenModeForTab` and `FullscreenController::EnterFullscreenModeForTab`) to put the whole current tab into fullscreen mode, but this time, `views::WebView` just fullscreens its normal native view (for current WebContents). Note that `fullscreen_widget_routing_id_` is *not used at all* here. Once that's done, as before, `ViewMsg_OnResize` is sent to the main frame renderer, telling it that fullscreen mode was entered. Blink then performs some layout magic to put the fullscreened element in front and hide everything else within the current page's bounds.

Background: Blink side

On the Blink side, there are two relevant classes: `Fullscreen` in core/ (one per `Document`), and `FullscreenController` in web/ (one per page, lives on `WebViewImpl`, not to be confused with the browser-side class of the same name).

Suppose there is a DIV which wants to go fullscreen. The page will execute something like:

```
document.querySelector("div").webkitRequestFullscreen();
```

This will bubble up to current document's `Fullscreen::requestFullscreen`, which will:

- Walk up the iframe chain to see if all ancestor iframes have the `allowFullscreen` attribute set to true.
- constructs a `fullscreenchange` event to be fired on the fullscreened element
- walks up the frame ancestor chain and constructs a bunch of `fullscreenchange` events in the embedding documents with e.target pointing at appropriate <iframe> elements.

All these events are saved in a queue but not fired yet.[1]

Next, things move to the web/ layer to `FullscreenController::enterFullScreenForElement`, which will save the provisional fullscreened element and some page sizing stuff, and then ask the embedder (`RenderFrameImpl`) to send the `FrameHostMsg_ToggleFullscreen`.

---

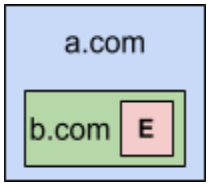
[1] The reason for saving them now is to preserve the fullscreen request type (prefixed vs. unprefixed), which have slightly different event semantics. When fullscreen is entered later, the event of appropriate type is fired.

Once the renderer receives a `ViewMsg_OnResize` with the fullscreen bits turned on, `RenderWidget::Resize` detects that it entered fullscreen and calls `RenderWidget::DidToggleFullscreen`, which forwards to `WebWidget::didEnterFullScreen`. `WebViewImpl` then calls `FullscreenController::didEnterFullScreen`. This updates various page sizing stuff and call `Fullscreen::didEnterFullScreenForElement`, passing the provisional fullscreen element it had saved earlier.

`Fullscreen::didEnterFullScreenForElement` is where the most interesting things happen:

1. It triggers the magic layout stuff, which fill the whole page with the fullscreen element and hide everything else, **"by wrapping the fullscreen element in an anonymous flexbox with a large z-index and solid black background"** [source]. This specific mechanism doesn't actually follow the mechanism described in the spec's rendering section, and there were some attempts to change this, but it doesn't look like anything shipped (see https://codereview.chromium.org/1363023005, which was reverted).
2. It walks up the fullscreen element's ancestor element chain, crossing frame boundaries, and on all those elements it sets a contains-fullscreen-element flag to true (see `Element::setContainsFullScreenElement`) and triggers style recalc -- this is used to apply CSS `:-webkit-full-screen` pseudoclass to the element and all <iframe> containers. Containers also get the `-webkit-full-screen-ancestor` class. Typically, an element detects that it's become fullscreened via `:-webkit-full-screen` and restyles itself accordingly (e.g., setting "`width: 100%; height: 100%`" to fill the whole screen). Example.
   Some elements, including <iframes>, have default UA CSS rules for the `-webkit-full-screen` pseudoclass, which are specified in Source/core/css/fullscreen.css. So, when an element is fullscreened in an iframe, the parent window will apply these styles to the `<iframe>` element to stretch it to fill the whole viewport, remove margin/borders, etc.
3. It triggers dispatch of the `fullscreenchange` events it had saved earlier in `Fullscreen::requestFullscreen`, for current frame and all the ancestor frames.

HTML5 fullscreen with OOPIFs: what happens today



Suppose a.com embeds a frame b.com, which requests an element E to go fullscreen.

First, b.com incorrectly calculates that it is always allowed to go fullscreen (see allowfullscreen). Next, it sends `FrameHostMsg_ToggleFullscreen` to the browser, which makes the current `WebContents` fullscreen. However, the browser then sends the confirmation of the fullscreen change (in `ViewMsg_Resize`) to the main frame

process, a.com, which bail outs early since it's not expecting any of its elements to go fullscreen. We end up leaving the whole tab fullscreen and leave the `FullscreenController` in b.com hung in provisional state.

Making HTML5 fullscreen work with OOPIFs

Overview

The following aspects of fullscreen need to be refactored to work with OOPIFs:

- `allowFullscreen` attribute needs to be replicated to proxies
- `document.fullscreenElement` needs to return the <iframe> element containing the fullscreened element in ancestor frames
- `-webkit-full-screen` and `-webkit-full-screen-ancestor` CSS styles need to be correctly applied to OOP <iframe> ancestors.
- `fullscreenchange` events need to fire on ancestor frames.

allowfullscreen

Elements in <iframes> can only go fullscreen if their iframe container elements have the `allowfullscreen` attribute set to true.

For example, in this page hierarchy,

```
foo.html: <iframe src="bar.html"></iframe>
bar.html:       <iframe src="baz.html" allowfullscreen=true></iframe>
baz.html:               document.querySelector("img").webkitRequestFullscreen()
```

the image is not allowed to go fullscreen because the top frame lacks the attribute.

This is exposed to JS via `document.fullscreenEnabled`, so that any frame can check whether it is allowed to go fullscreen.

Currently, this attribute is not replicated across processes, so a fullscreen request in an OOPIF will incorrectly always succeed. (We fail open because the ancestor frame iteration in `blink::fullscreenIsAllowedForAllOwners` thinks it reached the top-level frame once `Document::ownerElement` returns nullptr, but that also returns nullptr for remote parent frames.)

Proposal: add the `allowfullscreen` flag to `FrameReplicationState` and replicate it similarly to sandbox flags or strict mixed content flag. Note that although `WebFrameOwnerProperties` seems like a good fit for it, that is not replicated to proxies, so

we won't be able to look it up when walking the ancestor chain containing several `RemoteFrames`. Dynamic JS updates of the attribute take effect immediately and so will need to be sent immediately to all proxies.

The checks in `blink::fullscreenIsAllowedForAllOwners` will need to change to properly discover the flag for `RemoteFrameOwner`.

Alternatively, we could add the flag to `WebFrameOwnerProperties` and replicate that struct to proxies in addition to `RenderFrames`. This might make sense since updates to `WebFrameOwnerProperties` should be very rare and uncommon. As an optimization, we could replicate it only when it has non-default values.

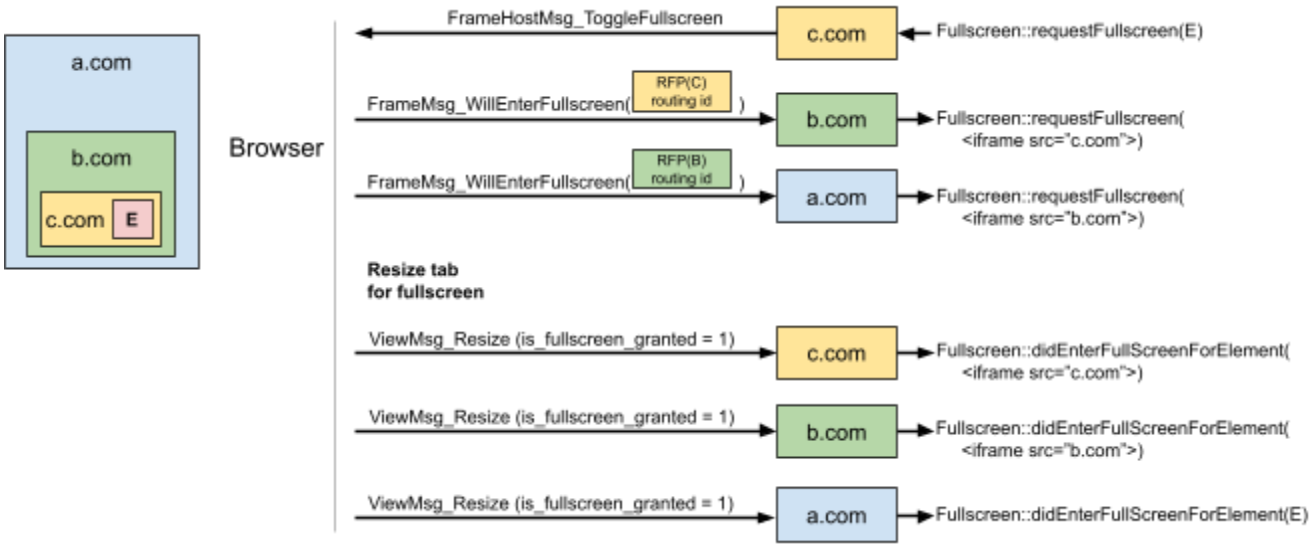
Browser->renderer IPCs

Currently, `ViewMsg_OnResize` carries the fullscreen bits and is only sent to main frames. However, if an element goes fullscreen in an OOPIF, we will need to send an IPC to that subframe's renderer, as well as all renderers on the ancestor chain, as all of these renderers will need to dispatch `fullscreenchange` events, flip `containsFullScreenElement` bits, and correctly support `document.fullscreenElement` (which needs to return the <iframe> element containing the fullscreened element in ancestor frames).

Proposal: add a `FrameMsg_WillEnterFullscreen`, which can be sent to a particular frame when it contains an element that is going to enter/exit fullscreen. Let `ViewMsg_Resize` trigger fullscreen enter/exit for subframe widgets in addition to main frame ones.

The reason for keeping `ViewMsg_Resize` as the trigger for entering fullscreen, rather than a separate IPC, is that it also carries the fullscreen resize params that will stretch the widget to fill the whole screen. This way, we (1) don't fire the `fullscreenchange` event too early, before fullscreen resize has happened, and (2) don't separate the work of resizing the widget and restyling content for fullscreen, which might help avoid flicker with the wrong intermediate layout.

The `FrameMsg_WillEnterFullscreen` IPC can include the routing ID of the child frame proxy containing the fullscreened element for OOPIF scenarios. For example, suppose a.com embeds a subframe b.com, which embeds another subframe c.com, which requests its element E to go fullscreen. In this case, the browser will send IPCs to b.com and a.com in addition to c.com:



Each renderer, upon receiving the IPC, can invoke `Fullscreen::requestFullscreen()` on either the actual element or the embedding <iframe> element. This will prepare `fullscreenchange` event and future document.fullscreenElement in all frames. Later, when the browser completes fullscreen resizing, it will send `ViewMsg_Resize` to all widgets, with the `is_fullscreen_granted` bit flipped. That can in turn forward to `Fullscreen::didEnterFullScreenElement`, which will take care of setting CSS `:fullscreen` everywhere, updating bits required for `document.fullscreenElement`, firing `fullscreenchange` events that are already prepared, etc. Note that it shouldn't be necessary to iterate through widgets to send a `ViewMsg_Resize`: sending it to the top widget should trigger `frameRectChanged` on subframe widgets, which will then themselves send a `ViewMsg_Resize`, and this can continue downward. We do want to change `WebContentsImpl::IsFullscreenForCurrentTab` (used by `RenderWidgetHostImpl::GetResizeParams`) to allow subframe widgets to know that they're in a fullscreen tab.

Question 1: should we send this IPC per frame, per widget, or per ancestor SiteInstance? E.g., in a A-B-B-C hierarchy, do we send one or two messages to B's process? What about A-B-A -- does the top A need an extra message? Per frame would be easier to reason about on the browser side. Per widget or SiteInstance may make it simpler to deal with the fullscreen layout stuff that Blink currently does (see next section).
Answer: For A-B-A, after calling element.requestFullscreen() in the bottom frame, JS in the top frame can synchronously query document.webkitCurrentFullScreenElement, which should return the <iframe> element for B (even before fullscreen is entered). This suggests that we should process all local ancestors in one shot in places like `Fullscreen::requestFullscreen`, and given that, it makes sense to send the IPC one for each ancestor SiteInstance to be consistent.

Another reason for this is that there's one `FullscreenController` per page, so sending the IPC per widget or per frame will need some refactoring of how `FullscreenController` works.

Open question 2: is it ok to send all the renderer IPCs in parallel or do they have to be done serially? The spec (section 6 in requestFullscreen) does specify a particular order (top-down) for firing `fullscreenchange` events, but I wonder if we can get away with that with the usual argument about inherently asynchronous nature of cross-origin frames.

Timing of resizing and onfullscreenchange events: It appears that one reason for tying fullscreen changes to `ViewMsg_OnResize` is so that `onfullscreenchange` event handlers can observe the new innerWidth/innerHeight after fullscreen resizing has occurred (see 100264). This is why the proposed design doesn't use a separate browser->renderer IPC for actually entering fullscreen, but rather reuses `ViewMsg_OnResize`. Unfortunately, it appears difficult to guarantee the final width/height even at the time of `ViewMsg_OnResize` (see 396576, 544269, and fix resize doc). For example, in default Chrome today, entering fullscreen with devtools open on the side, or with gold bars on top, appears to give a transient innerWidth/innerHeight inside the fullscreenchange event handler. So this problem might need to be addressed everywhere consistently, not just for OOPIFs, and so this won't be high priority initially. It appears that developers already know about this limitation and use the onresize event instead (example). Also see this related discussion of fullscreen and orientation.

Layout for a fullscreened element with multiple frame widgets

This is probably the trickiest part. As is, Blink's layout magic to fullscreen an element in a `WebFrameWidget` will only work within that widget's boundaries. So, additionally, we need to somehow stretch the subframe to cover the entire screen, covering all the ancestor frame widgets.

Fullscreen Flash codepaths contain the ability to fullscreen a particular widget within a WebContents (via `fullscreen_render_widget_routing_id_`, which could be expanded to also keep track of process ID). However, this can't be used because `RenderViewHostViewChildFrame` doesn't have its own native view, which `views::WebView::ReattachForFullscreenChange` requires.

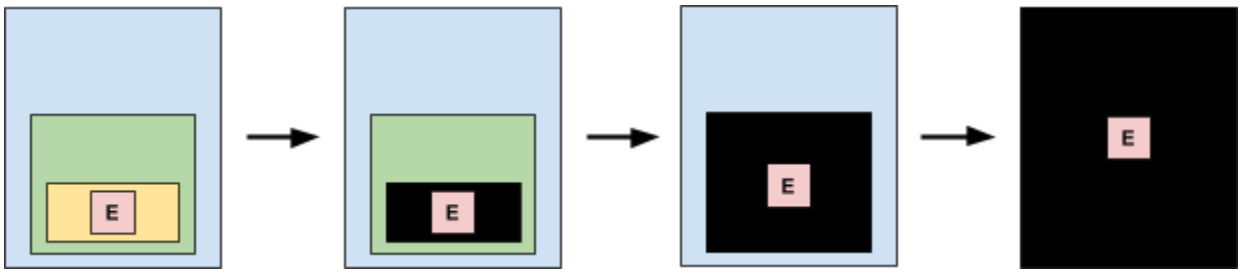
We could do something on the browser side to resize/reposition each widget on the ancestor chain to completely fill the space in its parent widget, but it seems easier to drive that from the renderer:

Proposal: when an element E in a subframe widget is fullscreened:

1. Take the entire tab fullscreen via `WebContentsImpl::EnterFullscreenMode`.
2. Run the Blink fullscreen layout algo to fullscreen E in the subframe widget's bounds.

3. For all the ancestor widgets, run the Blink algo to fullscreen the <iframe> element containing the next subframe widget.
    a. This can be facilitated as part of handling `FrameMsg_WillEnterFullscreen` and `ViewMsg_Resize` (is_fullscreen_granted=1) from the previous section.

For example, if the element to be fullscreened is in the bottom widget in a hierarchy of three frame widgets, the fullscreening layout would be adjusted roughly like this:



This is somewhat similar to how fullscreen is implemented in <webview>: when an element goes fullscreen inside a <webview>, it is first fullscreened within the guest's bounds, and then the embedder executes JS to make the <webview> element fullscreen (see https://codereview.chromium.org/984963004), which proves that the layout part of this should work.

Note that "within the widget's bounds" doesn't mean that E is cropped to the widget's bounds. If E is bigger than the size of the subframe's widget, E might still become fully visible in fullscreen mode. So really, E should be resized to the visual viewport size, and then gradually become more and more visible as its ancestor frames become fullscreened. See visual viewport below.

Visual viewport

The page's visual viewport is used by `LayoutFullscreen::updateStyle()` when setting the style of a fullscreened element to determine the container's width/height. When this is called in a subframe widget, the

```
document().page()->frameHost().visualViewport().size()
```

returns a null size. I.e., we don't track visual viewport correctly in subframe processes, and this needs to be fixed. Visual viewport is a page-level concept, so whenever it changes, the new size needs to be replicated to all subframe renderers.

To update this for OOPIF, we need to:

1. Pass correct visual viewport in `ViewMsg_Resize` messages to subframe widgets. Currently, the source of `ResizeParams::visible_viewport_size` is `RenderWidgetHostViewBase::GetVisibleViewportSize`, which doesn't return correct size for `RenderWidgetHostViewChildFrame`: it returns the widget's size rather than the page size.

2. Implement `WebFrameWidgetImpl::resizeVisualViewport()` for OOPIFs. Currently this has a FIXME.

Plumbing in WebFrameWidget

`WebFrameWidgetImpl::didEnterFullScreen` and `WebFrameWidgetImpl::didExitFullScreen` are both empty with FIXMEs to implement them for OOPIFs. Depending on whether browser->renderer fullscreen IPC targets frames vs. widgets, they might need to forward the call to the subframe renderer's `FullscreenController` or be NOTREACHED().

Return correct fullscreen status for subframe RWHs in browser process

`WebContentsImpl::IsFullscreenForCurrentTab` currently only returns true for main frame `RenderWidgetHosts`. It will probably need to work for subframe widgets as well (e.g., this is what gets used to populate `ViewMsg_Resize` params).

Exiting fullscreen

The exit flow changes for fullscreen will need to mirror the enter flow changes. I.e., IPCs will be sent to all renderers on the ancestor chain of an element that's exiting fullscreen. We will need to make sure that this works for both renderer-initiated exit (e.g., via `document.exitFullscreen` or by removing the fullscreened element from the DOM) and browser-initiated exit (pressing ESC).

Currently, `Fullscreen::fullyExitFullscreen()` uses `Document::topDocument()` and assumes the result is local; this will need to change.

blink::FullscreenController

`FullscreenController` assumes that the top frame is a `LocalFrame` in several places, e.g., when saving and restoring scroll offset:

```
m_exitFullscreenScrollOffset = m_webViewImpl->mainFrame()->scrollOffset();
```

This probably will matter only in main frame renderers, so we will need to bail out in subframe renderers.

`FullscreenController` is a page-level object which keeps track of the current fullscreened element and fullscreened frame (and assumes it's a `LocalFrame`). For example, this is used to handle transitions to a different fullscreen element when fullscreen is already active. This will

need to be adapted to work for renderers where the fullscreened element is in a descendant cross-process iframe.

Incorrect background when fullscreening a cross-site <iframe>

Currently,

```
document.querySelector("iframe").webkitRequestFullscreen()
```

for a frame that's rendered out of process will work mostly correctly, and the iframe's body will be correctly positioned and resized to fill the screen. However, in debug builds, with default styles, the iframe's background is incorrect (blue instead of white). In release builds, the old page "screenshot" is still visible alongside the fullscreened page.

Post-fullscreen resizing

While in fullscreen mode, the visible area might be resized: for example, by closing gold bars such as "default browser", or by devtools side-by-side. The plumbing for this also needs to work with OOPIFs.

Removing fullscreened element from DOM

Currently, `Element::removedFrom` checks for this and makes various OOPIF-unfriendly calls, including `setContainsFullScreenElementOnAncestorsCrossingFrameBoundaries`. We'll need to make this case work for OOPIF.

Making Flash fullscreen work with OOPIFs

There are two filed bugs:
Flash doesn't animate inside OOPIFs: https://crbug.com/593520
Renderer crashes when Flash fullscreen is activated from OOPIF: https://crbug.com/593522

The main fullscreen-related issues are:

- `RenderFrameImpl::CreatePepperFullscreenContainer` crashes because it tries to use main frame's document URL, incorrectly assuming that the main frame is local.
- Incorrect creator/opener routing ID is used when sending `ViewHostMsg_ShowFullscreenWidget` and `ViewHostMsg_CreateFullscreenWidget`. These pass in the subframe's widget routing ID from the renderer but are handled in `RenderViewHostImpl`. As a result,

`RenderWidgetHelper::OnCreateFullscreenWidgetOnUI` can't resolve the RVH from the passed in routing ID and aborts early.

- `RenderViewHostImpl::OnShowFullscreenWidget` aborts early if is_active_ is false. Similarly to `RenderViewHostImpl::OnShowWidget`, this can be legitimately called from a subframe for a swapped-out RVH, so this check is incorrect.
- WebContentsImpl tracks the fullscreen RenderWidgetHostView using a single routing id, fullscreen_widget_routing_id_. Various other places resolve the current fullscreen RenderWidgetHostView using `WebContentsImpl::GetFullscreenRenderWidgetHostView`, which couples that routing ID with the main frame's process ID. For OOPIFs, this is incorrect and needs to use the subframe's process instead.