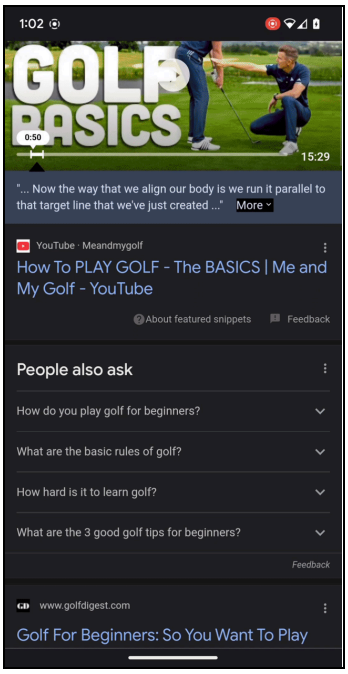
Default Navigation Transitions

Attention - This doc is public!

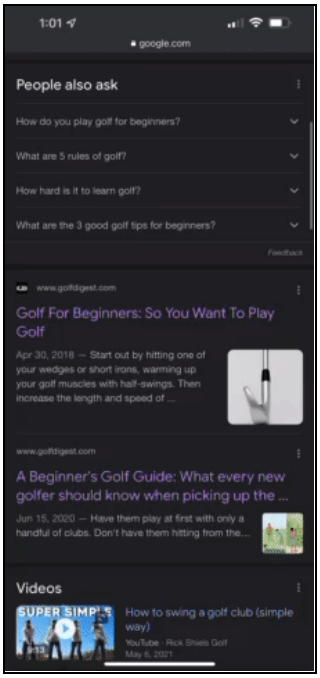
Contact: khushalsagar
Contributors: hvanopstal, jakearchibald
Status: Inception | Draft | **Accepted** | Done

Introduction

This design covers the technical details to support visual transitions when navigating across web pages in Chromium. See this PRD for motivation and detailed use-cases. The following is an TLDR example of the targeted UX : a swipe gesture lets you preview the back page.



Current Experience
(Chrome on Android today)



Targeted Experience
(Chrome on iphone today)

Note: The rest of the document makes comparisons with iOS since the targeted UX has already been shipped on iOS. The aim of these comparisons is to identify how problematic cases are being handled, not necessarily to copy the pattern. This helps to understand if Webkit already has an acceptable solution for these problems, Or we could work with Webkit folks to get consistent behaviour across the platform and make a case for new platform APIs, as needed.

Background

For the purpose of visual transitions, we've divided the different types of web navigations[1] into following categories:

Navigation History Interaction

The browser maintains a navigation history for each top level browsing context, commonly seen by users as 1:1 with a tab. Each navigation interacts with this history to:

- Push: Add a new entry to the history. The navigation could be to a pre-fetched or pre-rendered page, or require loading a new page.

- Traverse: Move the current entry to an existing entry in the navigation history, i.e., back/forward navigations.

  Note: This doesn't guarantee that the post navigation content aligns with what the user saw earlier. For example, logging out from a site in one tab could cause a history navigation in another tab to redirect the user to the login page.

- Replace: The current entry is replaced by a new entry. The navigation could be to a pre-fetched or pre-rendered page, or require loading a new page.

- *Reload*: The currently visible web page is reloaded.

Predictive vs Atomic

A user initiated navigation can be:

- Atomic
  Clicking a button which immediately **initiates** a navigation, i.e., the user action immediately conveys the user intent to start a navigation.

- *Predictive*
  A swipe gesture which shows a preview of the potential next active entry and initiates or aborts the navigation depending on the user gesture.

The key difference between atomic and predictive is whether initiating the navigation is done synchronously with the user gesture. Committing the navigation is asynchronous in both cases

[1] This documentation provides an excellent summary. Also the slides here for a user-centric summary.

and must wait until a server response from the destination (which could be several seconds after the navigation is initiated).

Same vs Cross Document

While each navigation will be associated with a distinct entry in the navigation history, it may be same or cross-document. Examples of same-document navigations include:

- Fragment Change: The navigation is to an anchor in the document while the browser will scroll into the viewport. For example: https://en.wikipedia.org/wiki/Cat to https://en.wikipedia.org/wiki/Cat#Senses.

- *Single Page Apps*: Script can make any change to Document state and add an entry to the navigation history, for example with `history.pushState()`.

In V1, we're targeting same and cross-document navigation traversals for predictive navigations [2]. While the design is focused on this use-case, the goal is to build a system which can support transitions for all use-cases.

Design

Overview

The basic control flow for the transition UX is as follows:

1. When a navigation is ready to commit, the browser preserves a visual representation of the outgoing DOM state to use in previews. This is similar to thumbnails stored for the tab switcher.

2. When the user starts interacting with the web content, the browser (or the OS) decides whether the gesture should be processed as a predictive navigation. The gesture progress feeds into the animation below.

3. The predictive animation composes a static preview of the navigation entry which will be made current (if the navigation is committed) with the live view of the current DOM state. Based on the user gesture, the navigation is committed (replacing the preview with live rendering for the new current navigation entry) or aborted (restoring live view of the current navigation entry).

---

[2] The rest of the doc refers to this as "predictive back navigation". This lines up with the Android terminology of this feature, since swipe from either screen edge triggers a back navigation. Chromium may eventually enable gestures to trigger forward navigations as well.

The feature involves the following broad components:

Thumbnail Cache

This cache manages capturing and persisting previews which are 1:1 with a navigation entry. There are multiple options for the exact visual state cached to trade-off between rendering latency (to display the preview) and memory:

- Pixel data
  This can be stored/compressed dependent on whether it's likely to be used:

  - GPU memory as texture (using a UIResource). This can also be compressed.
  - Bitmap on the CPU. Possibly encoded.
  - Persisted to disk.

  Note that this strategy is used by the Clank UI for snapshots displayed in the tab switcher, allowing reuse of optimizations. However the number of entries required for this feature will be significantly more: number of navigation entries across all tabs.

  Note: This is the only way to support native pages like NewTabPage.

  Capture Timing
  This capture can be executed asynchronously to avoid adding any delay to the navigation's critical path. This is because Viz already has the last rendered frame associated with the outgoing entry when a navigation is initiated. The capture requires flattening that frame into a single texture and doing a readback. This can be done async by generating a new SurfaceId when the navigation is initiated, and issuing a request to copy the previous Surface. This will likely need a synchronization primitive to ensure that the copy request reaches the GPU process before the renderer submits a new frame (and causes eviction of the previous Surface). This is particularly tricky for same-document navigations which can be committed in the renderer. The copy request will need to be issued in the renderer but the bitmap is sent to the browser process.

- Freeze Dried Tabs (FDT)
  This allows persisting the page as a vector graphic (SkPicture) which is memory efficient. Since rendering the picture requires parsing untrusted input from the page (web fonts, images), a sandboxed utility process is required to convert the FDT into a GPU texture.

  A benefit of this approach is that we can support minimal interaction (like scrolling) with the preview to cover up the loading latency. This however poses a challenge of synchronizing the scroll offset between the preview and live page, when the flip occurs. Since interaction with the preview is not a requirement, this benefit is not being

considered.

A caveat with this approach is that it requires going via the renderer process (or processes with OOPIFs) to capture an SkPicture representation of the Document. This will be difficult for same-document navigations where the FDT needs to be captured before the DOM is updated.

Capture Timing
The capture is more complicated than the Pixel data option. That's because it requires the renderer to execute a special paint operation to generate the FDT. It can be done asynchronously for the outgoing Document in the case of cross-document navigations. But same-document navigations would require the capture to be done synchronously when `pushState` is executed, since the developer could change the DOM after this call.

- BFCache
  BFCache persists the DOM and V8 state for a navigation entry. While this is core data that must be persisted, visual state derived from it could also be cached. This visual state can be anything after and including the Layout tree. Currently a BFCached renderer persists the data until paint. All gfx state from rasterized textures and beyond is discarded by marking the renderer invisible.

  We could avoid caching a snapshot if a page is in BFCache provided it can be made visible during the animation without triggering any script events. We'll also need to cache a snapshot if an entry is discarded from the cache.

  BFCache is limited to cross-document transitions since it persists the Document corresponding to an entry. Same-document navigations which change the DOM state for the same Document persist no state in BFCache.

The MVP will persist bitmaps on the CPU. These are uploaded to GPU memory (as UIResources) only for the duration of the transition.

Caching Strategy

There will be a global memory based cache to manage entries across all tabs associated with a browser instance. The cache eviction strategy can be optimized based on common user patterns. For example:

1. Users commonly go back to previous tabs and go back/fwd only 1 entry. Prioritize keeping only back/fwd entry for each tab.

2. Users rarely go back to previous tabs and navigate back/fwd. More common is multiple traversals on recently used tabs. Prioritize keeping more nav entries for recently used tabs.

Gesture Processor

The GestureProcessor is responsible for converting user gestures into inputs needed to drive the animation. For instance:

- Progress: 0-100%, based on where the swipe started and distance from the end of view/screen.
- Direction for swipe.
- Pointer location to track the gesture curve.

Note that there is a priority order for which system handles a user gesture:

1. Touch events will always be delivered to the web page first to allow the page to consume them directly. Or start a scroll gesture if the hit test target is scrollable.

2. Only when the scroll target has hit the overscroll limit when the gesture started (or the hit test didn't resolve to a scrollable element), the browser takes over the gesture to trigger predictive back nav. This implies that if any scrolling occurred in response to a gesture (which caused it to hit the overscroll limit), then a new gesture will be needed to trigger a back nav.

Clank Considerations

On Android, the OS establishes a fixed sized boundary along the edge of the screen. User gestures starting at this boundary are consumed by the OS instead of forwarding the events to the app. A GestureProcessor internal to the OS converts them into a predictive back gesture with high level platform events sent to the app. So this component will be a passthrough for Android.

WebView Considerations

Whether a predictive back gesture should be consumed by a WebView or another component is decided by the hosting app. This is similar to the hosting app consuming OnBackInvoked and conditionally forwarding the event to WebView using WebView.canGoBack() and WebView.goBack().

The WebView API should provide a hook to enable embedders to forward OS events to the WebView for predictive back nav. For instance, `WebView.getOnBackAnimationCallback()`.

Question: Can this be finch controlled? What happens if the feature is disabled, is there a pattern for APIs that provide this info to the app and expect graceful fallback.

UI Integration

Rendering the cached snapshot with a live view of the current page requires integration with the browser compositing system. The browser compositor goes through multiple layers of abstractions which are different across platforms.

Clank

Clank UI code builds a tree of CC layers. The content module provides Chrome with a CC layer which encapsulates the content to render into the tab's web content area via ViewAndroid::GetLayer. The layer is then added to the CC layer tree built by the UI.

This View can return a subtree of CC layers which cross-blend the cached snapshot and live web content (rendered as a SurfaceLayer). This would allow trivially plugging in the animation UI into the UI compositor.

Desktop

Desktop UI builds a tree of UI layers. These layers are internally backed by CC layers, resulting in a CC layer tree similar to Clank, but the CC layer backing is kept as an internal detail of the UI compositor.

The UI layer for live web content is integrated with the browser UI via DelegatedFrameHostClient (mac and aura). These code points can be updated to build a tree of UI layers to cross-blend the cached snapshot and live web content (provided by DelegatedFrameHost).

WebView

WebView doesn't have any browser-side UI. Frames submitted by the renderer are directly displayed by the display compositor. There is a setup to route these frames via the browser for integration with HWUI. But there are no browser driven animations or content other than renderer frames. Since Clank/Desktop have additional UI, there is already a setup to add animations on top of renderer's frame which can be easily leveraged.

For WebView, we'll need a new component which does the job of CC/UI:

- Trigger BeginFrames/invalidations in the browser to request HWUI to redraw.

- Pass parameters to generate a CompositorFrame which blends a texture referencing the cached snapshot with live web content. The WebView pipeline already generates a CompositorFrame to wrap renderer's frame here.

Note: This step is effectively done by CC for Clank/Desktop.

Common Solution

With the differences above, we'd end up expressing the transition animations across 3 different gfx primitives:

Platform	Gfx Primitive
Clank	CC layers
Desktop	UI layers
WebView	CompositorFrames

Ideally we'd try to find a common integration point to avoid maintaining 3 different UI backends. We could set up a CC instance for WebView but that will likely add unnecessary overhead, since the requisite animation is a lot simpler than what CC is used for. Luckily, there is ongoing work to switch Clank from CC to CC Slim. The latter is a lightweight version of CC to reduce the computational overhead on Clank UI which uses a subset of CC's features. We'd likely be able to initialize an instance of this compositor on WebView.

I propose we start with CC layer integration to ship on Clank. When CC slim ships on Clank, we can initialize a CC slim instance on WebView and use the same UI code for the animation on both Clank and WebView. This can be swapped with one of the common solutions above as a part of building out the feature for WebView.

Compositor Frame Alternatives

Without CC slim, CompositorFrames is the deepest step in the rendering pipeline common to all platforms. So the following alternatives were considered:

- Creating a new CompositorFrameSink for the duration of the transition animation. This frame sink produces a CompositorFrame which blends the snapshot and viz::Surface for the live web content.

  All code which was embedding a viz::SurfaceId for live web content earlier would embed a viz::Surface corresponding to this frame sink instead. It's unclear how difficult changing this would be.

- Mutating renderer's CompositorFrame before forwarding it to the display compositor. This is the approach taken by ViewTransitions by changing the rendered frame corresponding to a Surface here. The Viz side code currently assumes only one switched frame at a time. So this would interact badly if ViewTransitions also switched the frame but it's fixable.

Layering

There are 2 options for layering of this code in the Chromium stack. The proposed solution is the Content option.

Content

The feature is an internal detail for the content/ module. There is a minimal public API for embedders to route OS events (see Clank Considerations) to content/ internal code. This API will also be optionally used if the events are driven by the OS. Desktop platforms (unsure about mac) will involve a content internal Gesture Processor, similar to how overscroll works.

The arguments for this approach are as follows:

- While the feature is currently limited to browser UX, it will eventually be tightly coupled with View Transitions (see Developer Customization). The Thumbnail Cache will persist visual state as spec'd in View Transitions API. Since the feature components will be required to support a Web Platform feature, it makes sense for them to exist in content/ (and eventually some parts in third_party/blink).

- The logic for how a gesture is processed and reflected in the UI, if not handled by web content, is currently a content/ internal detail. It makes sense to have a unified module which does this for overscroll or predictive back nav.

- The feature doesn't need to be supported on iOS. WkWebView supports this natively, which is used by Chrome on iOS today. If there is a Blink based Chromium browser on iOS, it will build on top of content/.

Component

The feature is a new component/ which layers on top of the content/ public API. Code in chrome/ provides the glue for integrating with requisite content/ objects. The only argument for doing this would be to follow the design principle of limiting content/ code to the minimal feature set required for a secure and web compliant browser. Given that parts of this feature will be required to support a core web platform feature, it makes sense for this to live in content/.

Alternatives Considered

The design above proposes a system which lives completely in the browser process. The interaction with the GPU process is to animate a texture similar to the omnibox animation on Clank.

An alternate way to do this would be to build Web Platform primitives (see Developer Customization) and implement the default browser UX on top of that. For example, launch chrome, enable chrome://flags/#view-transition-on-navigation[3] and you'll see all same-origin navigations doing a cross-fade animation. This is done using the ViewTransition Web API being prototyped for same-origin navigations, with the animation defined using UA CSS. Developers can then trivially customize the transition by keeping some UA behaviour and overriding others.

An approach like this helps invest engineering resources into building and maintaining a single system which works out-of-the-box for Web developers. Unfortunately this approach can't be taken for security reasons:

- The feature works by generating pseudo-elements in the context of the old or new Document. This allows the renderer process to access texture handles from another Document. This can not be permitted for cross-origin navigations.

- We could've explored design options to limit/eliminate the attack surface if ViewTransitions were used by cross-origin transitions by:

    - Only supporting root transitions, so there is no geometry state from the previous page. If the restored Document will already receive events exposing that this is a back navigation no new information would leak.

    - Limit the duration for which the renderer process has access to the texture handle associated with the cached snapshot. Even if the process is compromised, and changes how the snapshot is rendered, it won't be able to embed the snapshot after the animation duration (< 0.25s).

    - No input is forwarded to the renderer process during the animation, so a compromised renderer can't cause the user to interact with any part of the snapshot.

However, gesture based transitions cross-blend the 2 Documents for the duration of the gesture. The feature is also meant to give users a preview of where the navigation will take

[3] Docs doesn't allow linking to chrome:// URLs, see b/265215274.

them. Driving this in the renderer implies the user can be spoofed about where the navigation will commit.

For these reasons, this alternative was not explored further.

Web Platform Edge Cases

Capture Timing

Same-document navigations have additional complexity since the old DOM state, which needs to be captured for a preview, is not explicitly known to the browser. The following are example scripts which an author could use for a same-document navigation:

```
function switchToBAndNavigate() {
   updateDOMtoB();
   window.history.pushState({}, '', newURL);
}

function navigateAndSwitchToB() {
   window.history.pushState({}, '', newURL);
   updateDOMtoB();
}
```

`navigateAndSwitchToB()` adds a navigation entry before updating the DOM to the new state. So the visual state can be captured when this call is invoked. However, script could do the same with `switchToBAndNavigate()` which adds the entry after the DOM update, in which case the browser has no opportunity to capture the visual state.

Solution: The browser caches the last rendered frame before `pushState`, or any other API which commits a same-document navigation.

Interaction with Developer Transitions

A back navigation involves traversing the navigation history to restore all state associated with the previous entry. If this entry was added by the developer by updating the DOM (as described above), restoring it involves the developer restoring the prior DOM state. This is done by listening to the `popstate` event:

```
window.addEventListener('popstate', (event) => {
  updateDOMAtoB();
});
```

```
window.addEventListener('popstate', (event) => {
  document.startViewTransition(updateDOMAtoB);
});
```

The transition animation is a browser only visual change. There is no interaction with the navigation history (other than using the cached snapshot tied to the previous entry) when the gesture is initiated and ongoing. Once the gesture has resolved to commit the navigation, and the animation has finished to show a preview of state B, we ask the navigation stack to start navigating to the old state.

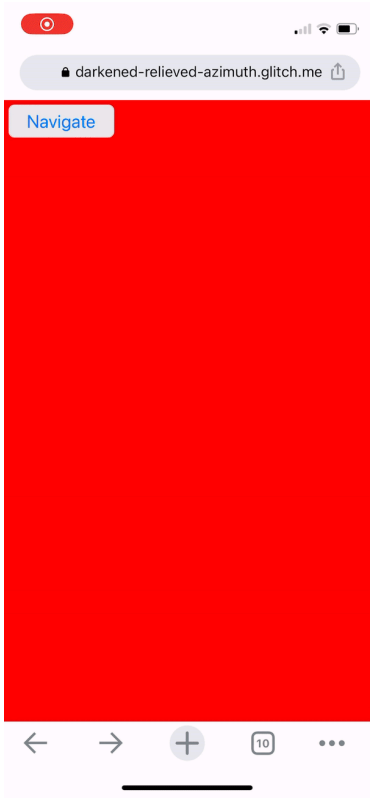
This implies that all script observable events associated with this navigation are dispatched when the user is seeing a preview of the entry being restored. This can be an issue depending on how the developer handles the `popstate` event:

- In the first version, the DOM is directly switched to the state B. This would not be noticeable to the user if the preview is consistent with this DOM state.

- In the second version, the developer sets up a transition to animate to the state B (either via native ViewTransition or an internal framework). This would cause the tab to go through the following sequence:

    - Browser animation from live A to preview B.
    - Script changes from preview B to live A.
    - Script animates from live A to live B.

iOS has already shipped this UX and the erroneous sequence in the second version occurs on a sample SPA site. The site changes the background on each navigation with a cross-fade, demo below:



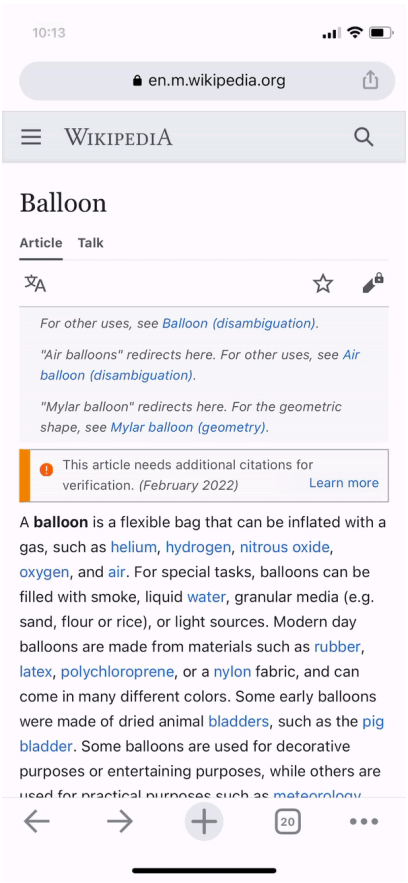
Chrome on iOS for the sample site

This pattern can also be seen on Google search here.

Solution: A hasUAVisualTransition bit has been added to navigation APIs to allow authors to detect when a UA transition was done. This can be used to suppress the author transition.

Transient State on Old DOM

The old DOM or old Document (this case is relevant for both same and cross-Document navigations) could have a transient UI which is only displayed when the navigation is triggered. Capturing the DOM's visual state will cause the preview to include the transient UI which will be inconsistent with the state seen by the user after navigation. This pattern can be seen currently on Chrome on iOS.



The options to handle this UX better are as follows:

- Encourage developers to dismiss such transient UI when a navigation is initiated. This is very difficult to do deterministically since authors would likely want this dismissal to be done using an animation. The browser's capture must happen after this animation finishes, which delays the navigation.

- Provide an event that informs the author when the DOM is being captured for a preview. This would allow the author to restore the DOM to a state which will be displayed when the entry is restored. This DOM state doesn't need to be displayed to the user, the browser could render a frame to capture this after the active entry on the tab has changed.

  This option isn't feasible for same-document navigations where switching the navigation entry mutates DOM state in the same Document.

- Do a cross-fade between the preview and the live web content for the restored entry to paper over the visual differences.

- beforeTransition API which lets the developer disable the browser transition and eventually add their own custom transitions for such cases.

Solution: The browser crossfades the screenshot with live new content after the navigation commits.

beforeUnload

The web platform provides APIs which allow authors to prompt users to cancel or confirm a navigation like beforeUnload or the navigation API. The aim is to prevent navigations where the user would lose state on the current Document (like a partially filled form). The main difference between these 2 APIs is that the prompt shown via beforeUnload is browser UI. While the navigation API would allow the web page to display its own prompt.

So we have 2 points in the timeline of the navigation when it can be canceled:

1. User starts a gesture which shows the preview. Depending on how the gesture ends, the navigation may be triggered or abort.

2. The author receives an event when a navigation is initiated which can trigger a confirmation dialog for the user to continue the navigation or abort.

It makes sense to sequentially do the above, let the user gesture end in a state where the navigation is initiated which will then dispatch beforeUnload as needed. This also lines up with the current behaviour where the navigation is initiated once the gesture ends. See Cancellable Navigations for how the UX interaction in this case.

Note: No comparison with iOS/Mac, beforeUnload is not dispatched for back navigations.

Developer Customization

The discussion for a web API to allow authors to customize gesture based navigations has been moved here.

UX Considerations

Preview to Live Switch

The transition involves animating a preview of the restored navigation entry and switching from the preview to a live version of the restored entry. The web content won't be interactive until the tab switches to the live version.

Solution: The screenshot has a scrim which stays until its cross-faded to the live content.

We also considered a small chip that says "preview" in the omnibox which disappears on the live version.

URL Bar/Omnibox

The transition animation is limited to the web content area. The URL bar always corresponds to the currently active navigation entry, which is the Document that receives events for that tab and can access sensitive permissions like location, microphone etc.

Currently, the URL bar is shown once the navigation is initiated. This feature also forces the URL bar to show when the user starts swiping. Since cross-document navigations force the URL bar to show when a navigation is started, it's almost always showing when the navigation commits. This means screenshots are in a state with the URL bar showing and the screenshot only includes the visible part of the page.

Displaying this screenshot when the URL bar is hidden causes guttering since it doesn't have content for the area covered by the URL bar when it was captured. Showing the URL when the swipe starts avoids this.

Thumbnail Cache Fallback

Always showing a preview on predictive back nav will require persisting state in the Thumbnail Cache for each navigation entry across all tabs in the browser. This can add significant memory overhead depending on the size of these entries.

Solution: A fallback using a theme based background color along with the favicon is used when a screenshot is not available.

Animation Types

Navigation History Interaction lists the different types of navigations based on how the history is mutated. While the first phase of this feature is scoped to traversals, we will eventually target all navigation types. These would need different UX animations to provide feedback to users about the type of navigation.

Swipe Direction

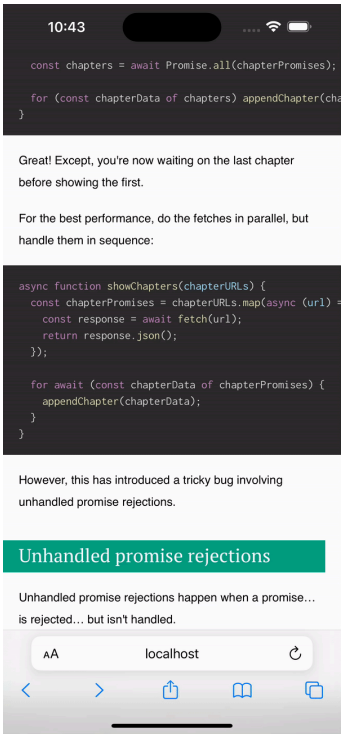
Swipes from either edge of the screen trigger a back navigation on Android. This may be confusing since a user could expect left to right swipes to trigger a back navigation and right to

left swipes to trigger a forward navigation.[4] This is also inconsistent with the behaviour on mac (2/3 finger swipes trigger a navigation) and iOS.

Solution: For swipes initiated from the edge which is logically not the “back” edge, Chrome will show no animation.

Scroll Inducing Fragment Navigations

Fragment Change navigations include same-document navigations which change the Document’s scroll-offset. The browser executes and is aware of the scroll offset change. There is also an existing primitive that allows the developers to transition this using a scroll animation. It’s unclear whether a full page swipe makes sense for these transitions. It could give the user a false perception that they’re switching to a new page while they are being scrolled to a different position on the same page. The following shows the current experience on iOS which does this:

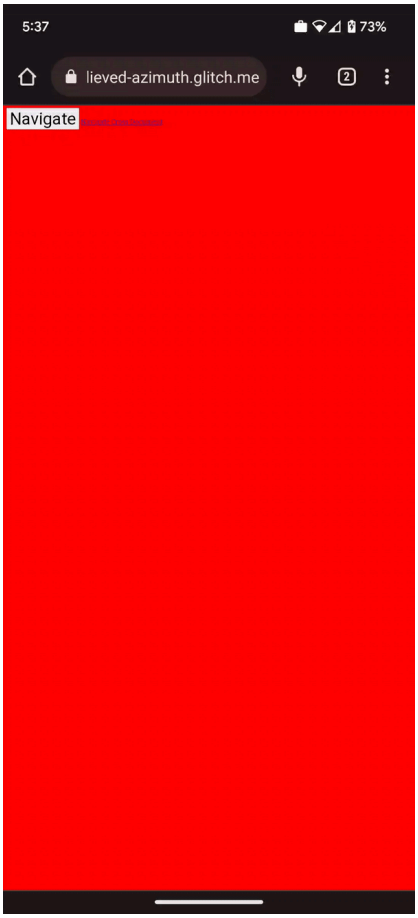


Solution: For consistency, this will use a full page swipe similar to all navigations.

[4] This could be reversed if the user’s default language is right-to-left (for example Hebrew).

Cancellable Navigations

For navigations which are canceled via beforeUnload, the current UX displays browser UI on top of the web content to let the user decide whether the navigation should proceed. An example is shown below.



Solution: If the web page decides to show a dialog, a cancel animation is played to bring the live page back to view while the user is interacting with the dialog.

Security/Privacy considerations

The following design decisions have been made considering potential security/privacy attack vectors:

- The user must be able to distinguish which is the page currently active in the tab. Active here implies the page which is able to execute code, create permission prompts or access security/privacy critical data like location, microphone. The following UX patterns

ensure this:

    - URL Bar/Omnibox displays the URL corresponding to the currently committed navigation entry.

    - The URL bar is shown when the user starts swiping, as opposed to the current behaviour which shows it when the user lets go and a new navigation is initiated. This ensures the URL bar is visible, and shows the currently committed page's URL, as the user is swiping. So the user understands which is the active page while the web content area is showing a screenshot of the previous page.

    - The visuals for the currently committed entry cover at least ~15% of the tab's content area at all times of the transition.
    - An indeterminate progress bar is anchored to the top of destination page's screenshot to clearly indicate that this page is still loading (example video).

- All input processing and updates to the UI to cross-blend live web content with cached snapshots (that could be cross-origin) is executed in the browser process. The renderer process is unaware of the animation.

- Snapshots tied to a navigation entry are also persisted in the browser process or GPU process. The isolation follows the same pattern used by omnibox screenshot or cross-origin textures across iframes/tabs in the GPU process.

- The previews are persisted until the user can go back to that navigation entry and the tab is in memory, i.e, the tab state hasn't been pushed to disk (either to save memory or because Chrome was backgrounded). We do want to eventually persist them to disk so if a user relaunches Chrome and navigates, the previews are still there. But that won't be done in this launch.

  If the user clears their history, the navigation list for all tabs is also cleared. Since the preview's lifetime is tied to the navigation entry, the previews would also be deleted in this case.

- If a page uses cache-control: no store, no screenshot is preserved and fallback UI is used instead. This is to avoid caching potentially privacy sensitive information the user wouldn't expect to be persisted.

- When going from A back to B, the live content from committed page B is overlaid with B's screenshot until B paints its first frame. This is to avoid fading from a screenshot of B to no content.

  However, if there is any case where the committed origin changes : server redirect or the

browser navigates to C before B paints its first frame, the browser immediately fades out the screenshot to reveal the live content from the currently committed origin.