Last updated: May 15th, 2024 | 9-patch segment

Last modified: Jan 24th, 2025 (ported to a document in the chromium contributors drive)

Deprecated, this document was ported to:

Fluent Scrollbars Technical Design Doc

Fluent Scrollbars Technical Design Doc.

Public document

Gastón Rodríguez
Rahul Arakeri
Yaroslav Shalivskyy

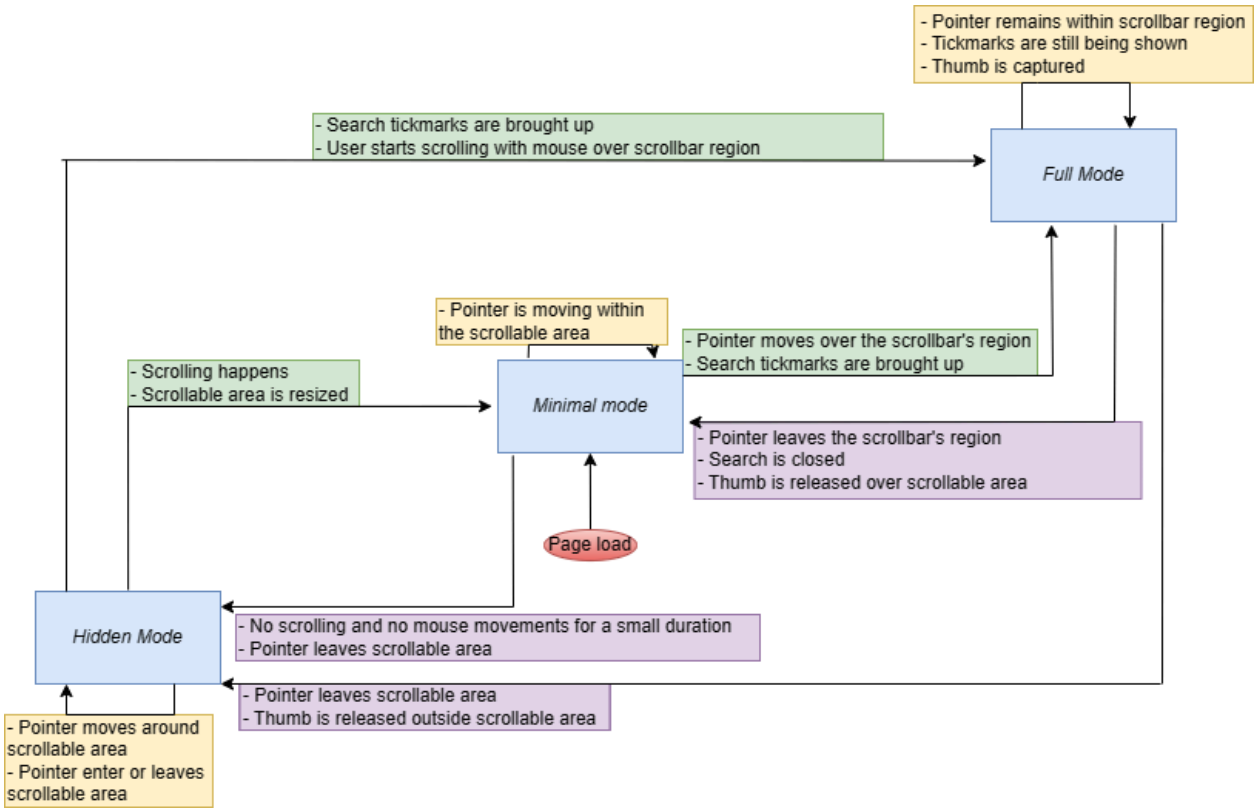
Visual specs: Fluent Scrollbars Visual Spec

State management (Overlay Scrollbars):

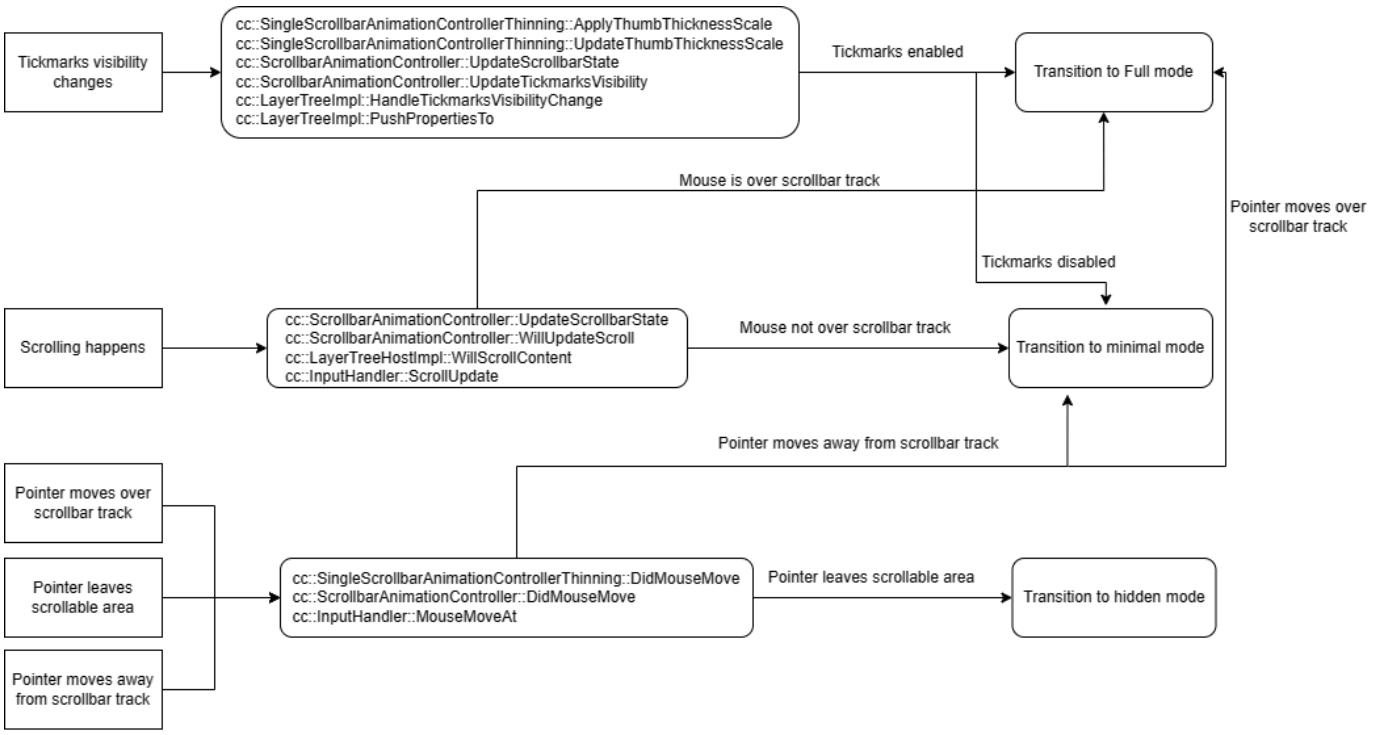
(Please see the visual spec (Section: State transitions) linked above to understand how the scrollbar looks in various states.)

cc thread

ScrollbarAnimationController and the SingleScrollbarAnimationControllerThinning handle the state transitions in cc. The following diagrams show stack traces for how the transitions will occur.



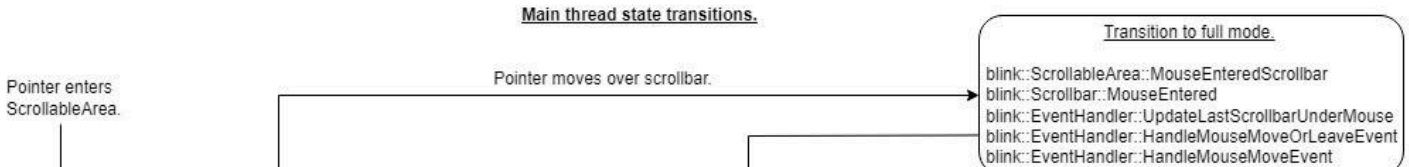
Transitions diagram



Stack traces of function calls for Transitions. This isn't an exhaustive diagram of all contemplated transitions, but it covers the main function calls involved in all transitions.

Main thread

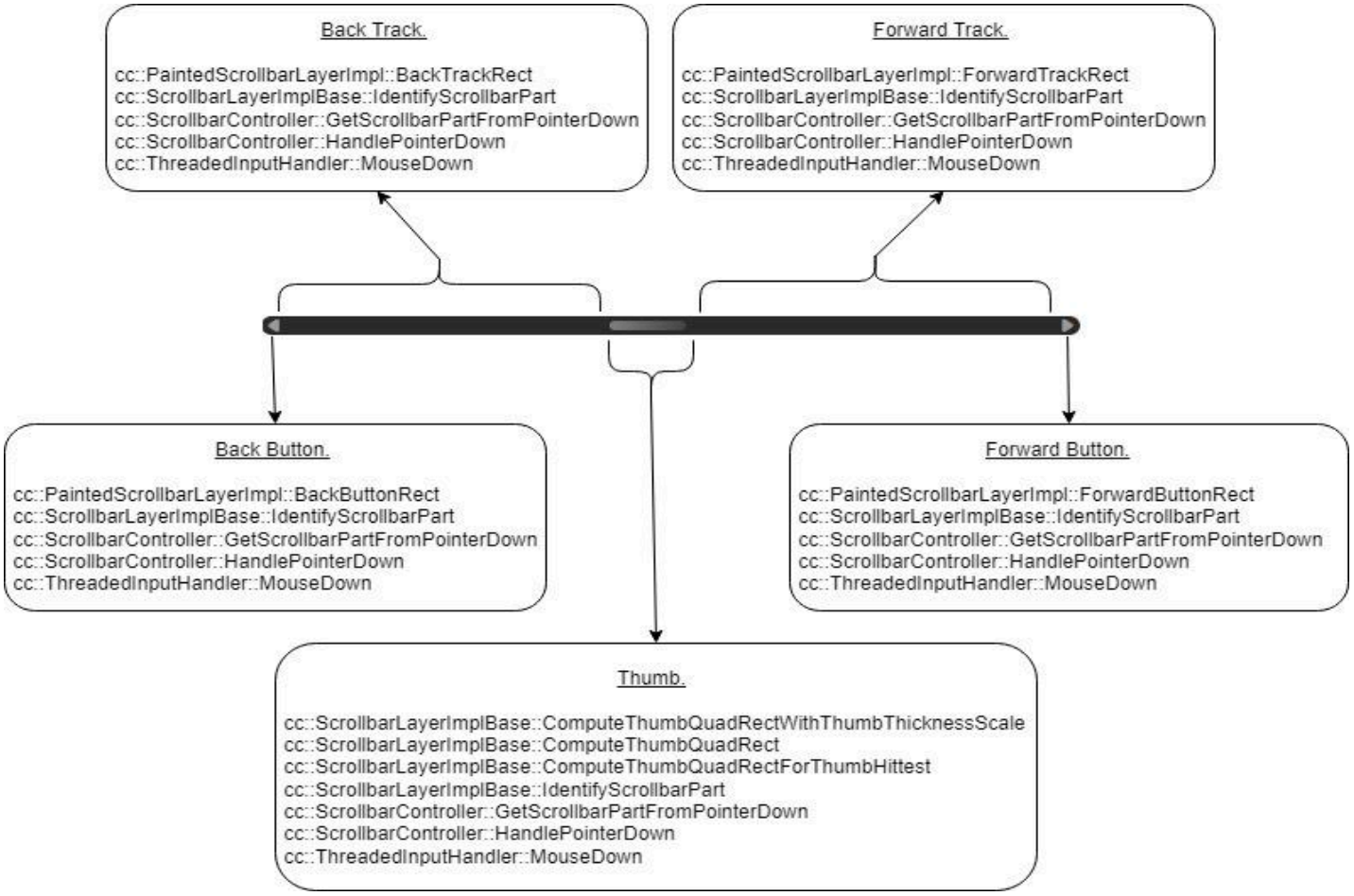
In Blink, state transitions are handled in ScrollableArea. This class will be modified to accommodate the new state transitions.



Hit testing:

cc thread

Non-overlay and Overlay scrollbars are backed by PaintedScrollbarLayer and they will be hit tested by cc::ScrollbarController. No new code will be needed here. The following diagram is only for reference.

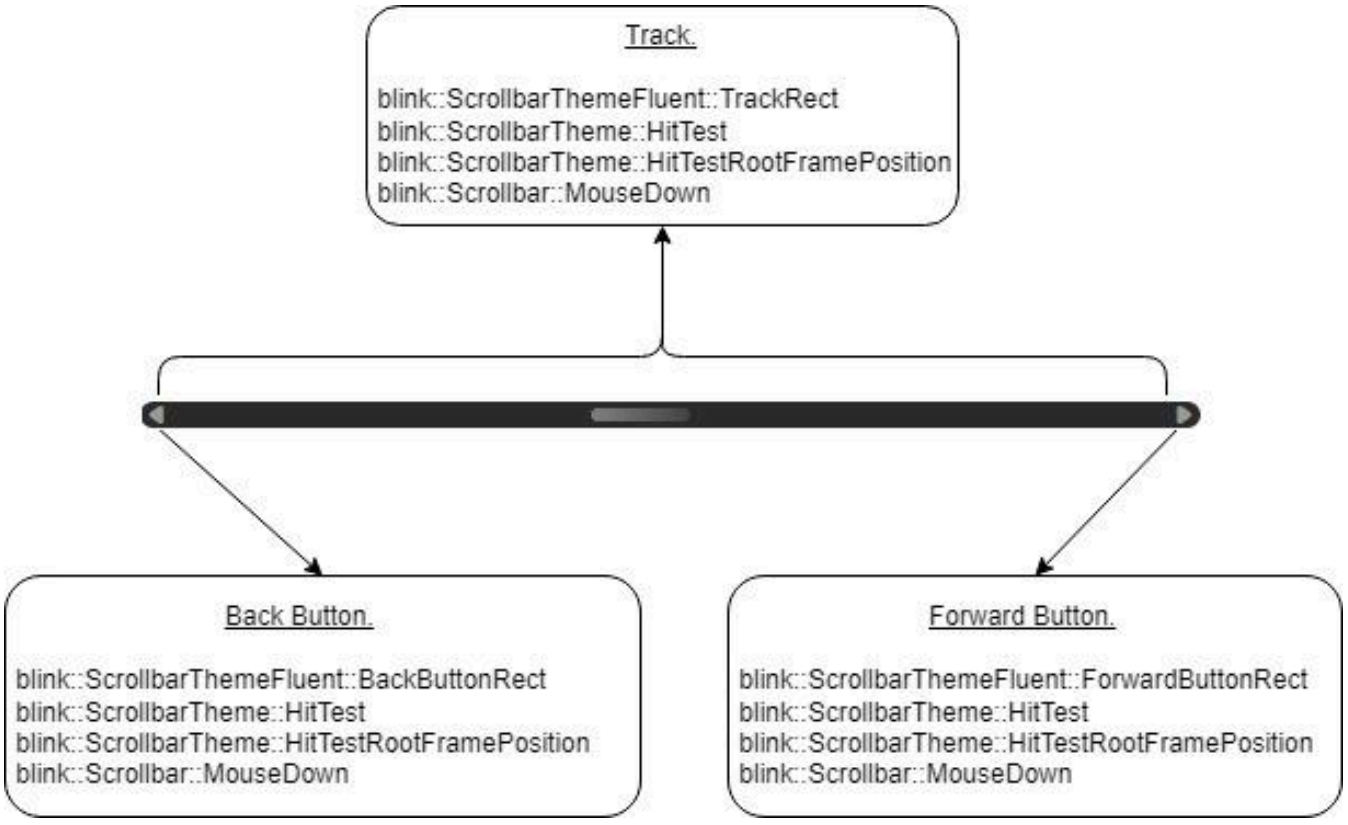


Fluent scrollbar parts (i.e rect dimensions) are decided by looking up the ScrollbarThemeFluent . Please see the class diagram in the "Painting" section to understand the hierarchy. Once it is added, an example of how scrollbar dimensions will be retrieved is shown below:

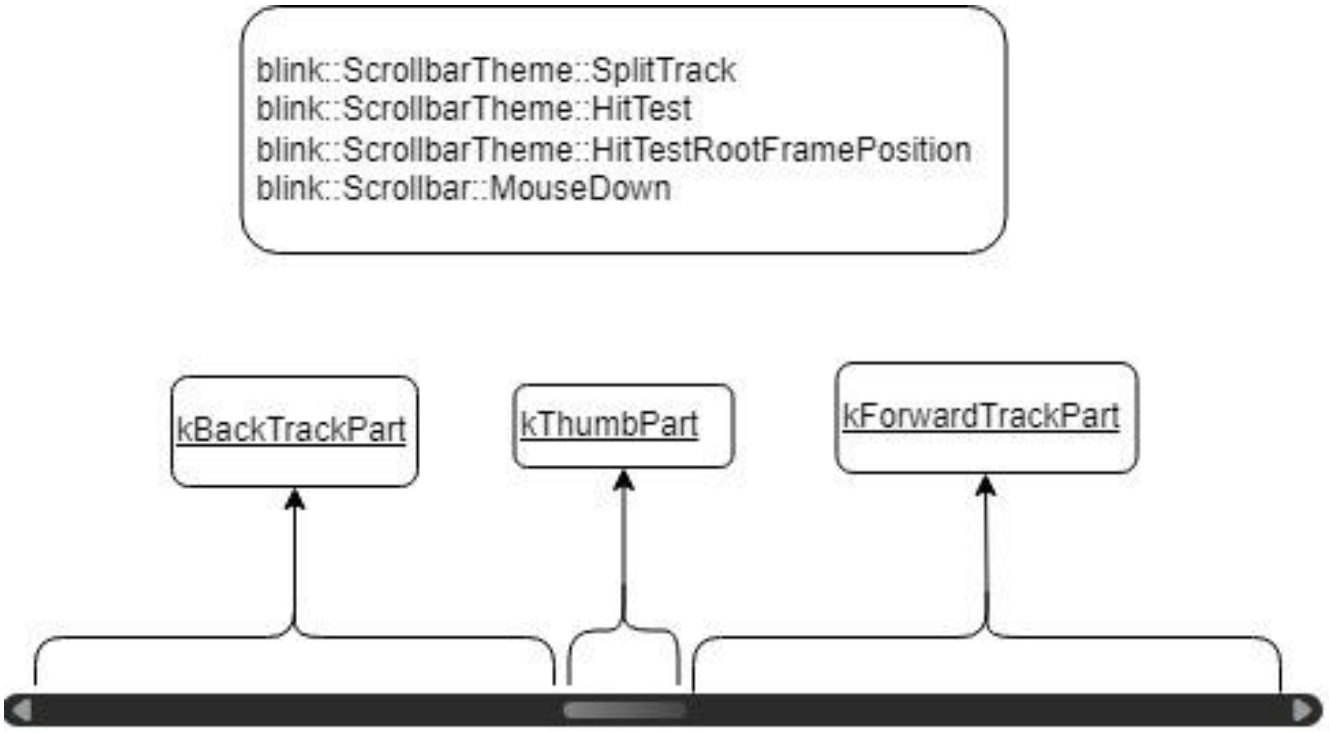
```
blink::ScrollbarThemeFluent::BackButtonRect
blink::ScrollbarLayerDelegate::BackButtonRect
cc::PaintedOverlayScrollbarLayer::UpdateWinStyleProperties
cc::PaintedOverlayScrollbarLayer::Update
```

Main thread

Blink scrollbars will be hit tested by ScrollbarTheme::HitTest. This is how the newly added ScrollbarThemeFluent matches for scrollbar parts..



For the remaining scrollbar parts, the track is split up as shown below (and ScrollbarTheme::HitTest will return the appropriate rect):



Painting:

This section describes the scrollbar painting pipeline across ui/, blink/, and cc/ modules. We provide the description for proposed changes in each module.

Regardless of the compositing being used, Fluent scrollbars painting is aided by *NativeThemeFluent* and *ScrollbarThemeFluent* classes, which help define dimensions and paint the bitmaps that are used by the scrollbars. The scrollbar parts paint invalidations happen in blink's Scrollbar class when appropriate.

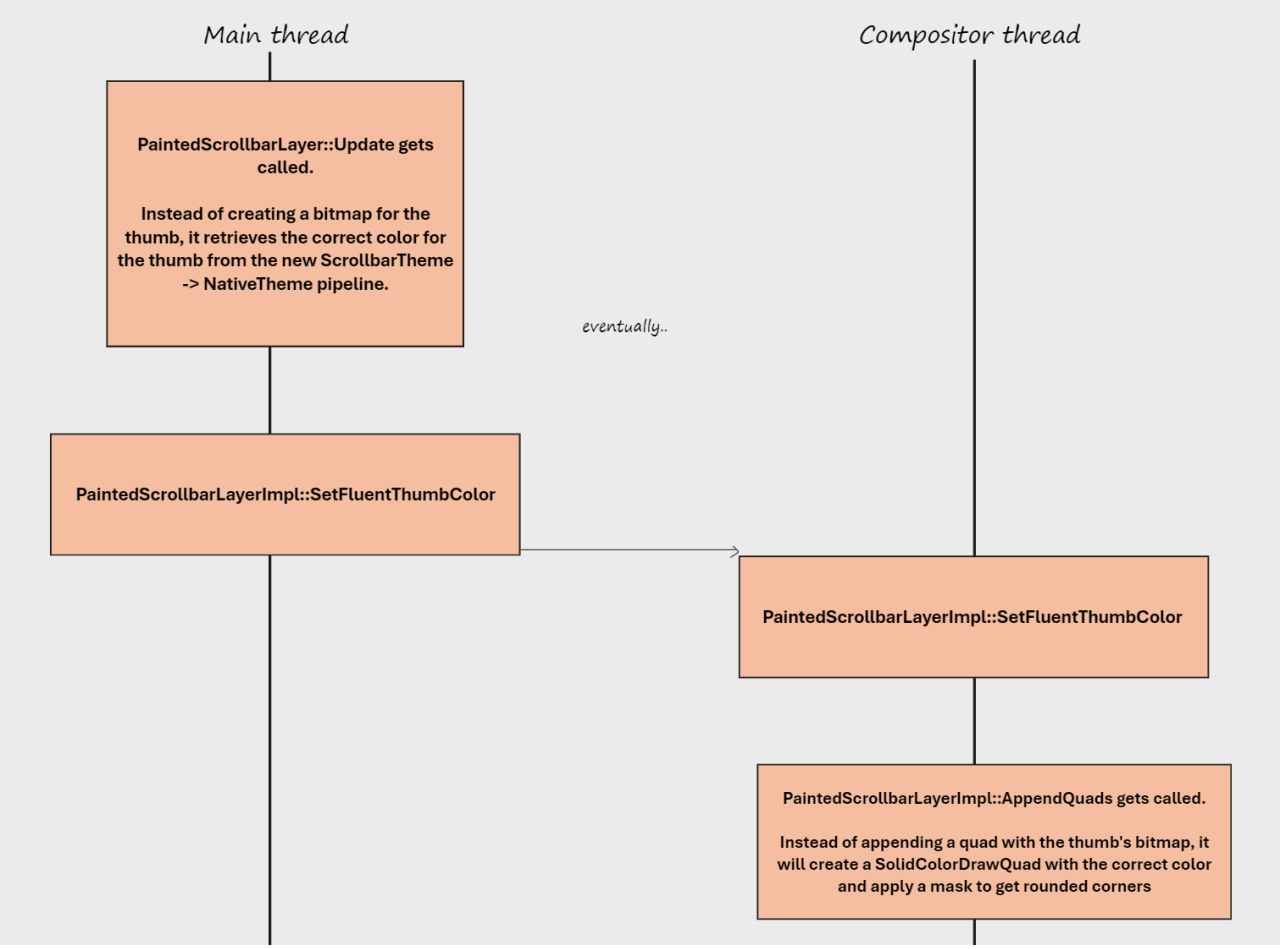
Main thread painting

Fluent scrollbars utilize the same architecture as regular scrollbars, painting via ScrollbarDisplayItem and the PaintArtifactCompositor.

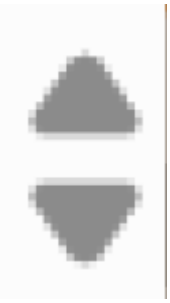
cc thread painting

Composited Fluent scrollbars are implemented on the *PaintedScrollbarLayer(Impl)* classes.

The scrollbar thumb will be painted entirely on the compositor thread, without a need for a bitmap to be painted and passed between threads. This is done by applying a mask to a solid color quad at composite time. The color for the thumb is piped from the main thread into the compositor thread by the layer classes.



The scrollbar track and buttons are painted and scaled using a 9-patch architecture. To understand the basics of 9-patch, reading this article is recommended. Using 9-patch scaling, the scrollbar parts would be painted the smallest possible bitmap, which would then be expanded to the appropriate size at composite time.

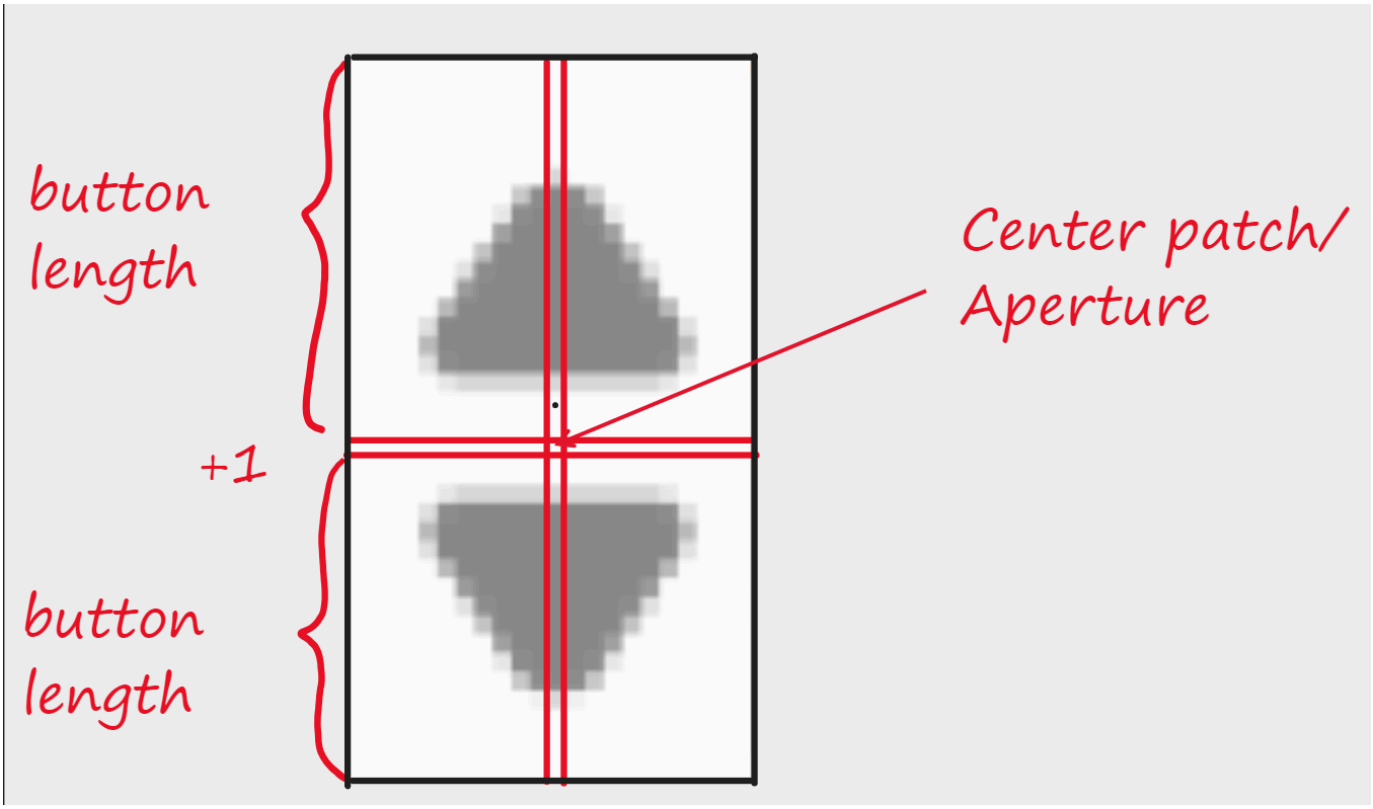


The smallest possible bitmap for the scrollbar buttons and track.

Following the existing Chromium implementation of 9-patch scaling, all images need two things defined: the *canvas* and the *aperture*. The canvas is the size of the smallest possible bitmap that would represent the image, and the aperture is a *Rect* which describes the center-patch for the image. In Chromium, buttons, track, and tick marks are all painted in the same bitmap.

- **Canvas:** The minimal possible dimensions a Fluent scrollbar can have is two times the length of a button, plus an extra pixel in between these two points for the center-patch. The width of the scrollbar would remain unmodified.

- **Aperture:** Scrollbars are only expanded in the direction of their scrolls, and not widened. To have the appropriate scaling, one pixel in the center of the bitmap is enough.
  When the scrollbars width is even, the button arrow will be painted with two “top” pixels, to enable proper scaling, the aperture will be two pixels wide instead of one.



Scrollbars smallest bitmap required to represent the scrollbar in its entirety

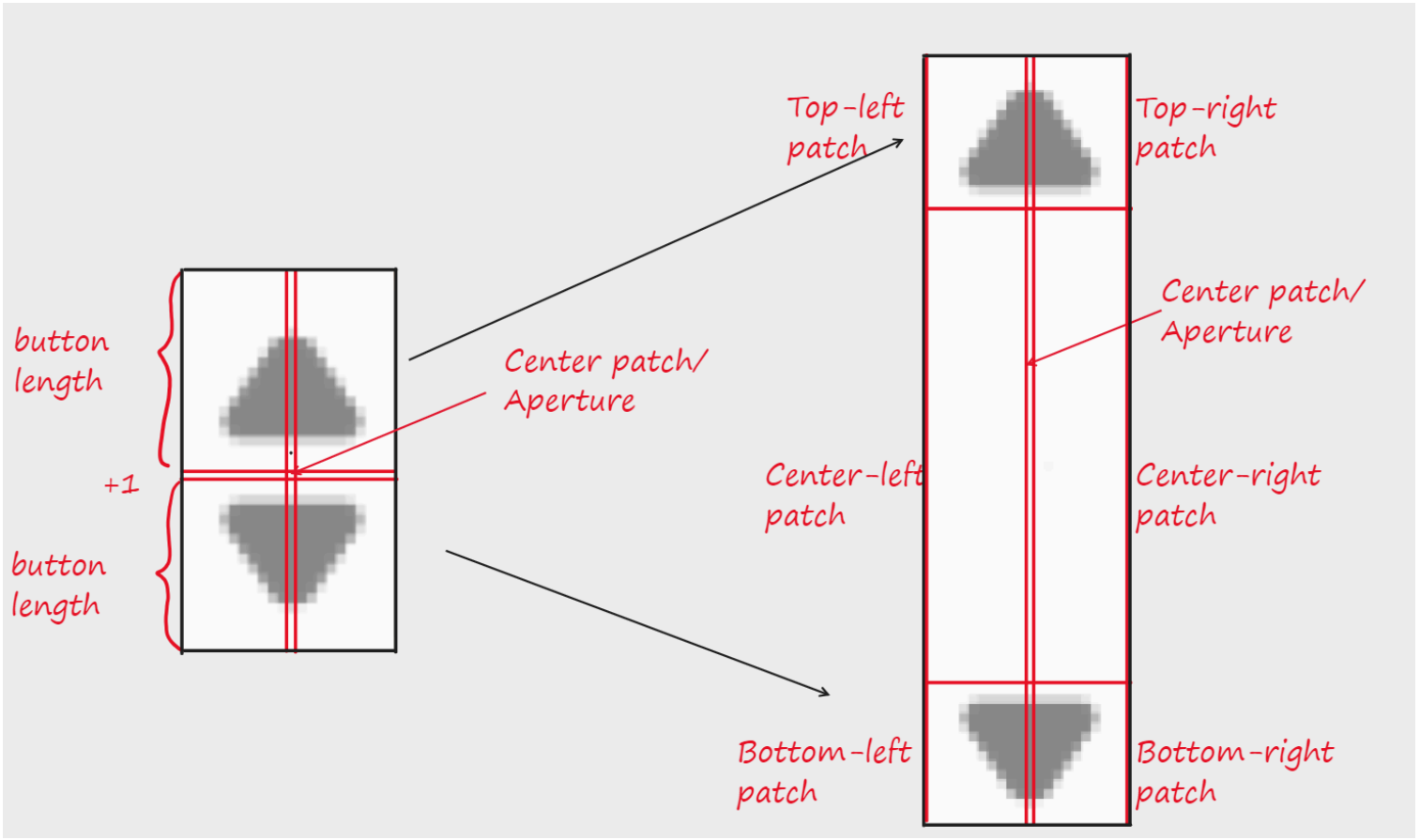


Diagram showing how the scrollbars would be expanded using 9-patch's architecture.

When find in page tick marks are present, nine patch scaling is not used and a full sized bitmap is used that contains all the tick marks.

Thinning animation

cc::ScrollbarAnimationController manages the scrollbars' animation state and supplies the thumb thickness to the *cc::ScrollbarLayerImplBase* on each animation tick.

```
cc::ScrollbarLayerImplBase::SetThumbThicknessScaleFactor
cc::SingleScrollbarAnimationControllerThinning::ApplyThumbThicknessScale
cc::SingleScrollbarAnimationControllerThinning::RunAnimationFrame
cc::SingleScrollbarAnimationControllerThinning::Animate
cc::ScrollbarAnimationController::Animate
cc::LayerTreeHostImpl::AnimateScrollbars
...
cc::Scheduler::BeginImplFrame
```

The thumb thickness value is used to calculate the correct size and location for the thumb rect and append *TextureDrawQuads* to the GPU process (link). Rasterized thumb resource (bitmap) is being scaled to the specified thumb rect geometry on the GPU process to be drawn on each frame. We are reusing the implementation that already exists in Chromium.

The main difference between Fluent overlay scrollbars and Chromium overlay scrollbars is the usage of a nine-patch architecture. Existing Chromium overlay scrollbars that are enabled by default on Android, ChromeOS, Fuchsia, and are available under the feature flag on other non-mac platforms use the nine-patch generator for creating draw quads for the thumb. As was mentioned in CRBug, nine-patch architecture prevents the stroke around the thumb from scaling during thinning animation. Fluent overlay scrollbar thumb UI design doesn't contain any strokes, thus, doesn't require such functionality.

The thinning animation is not supported on the main thread since there's no equivalent to *cc::ScrollbarAnimationController* in Blink.

Animations:

State transitions are managed with two types of animations: thinning and opacity animations. This section aims to explain the relationship between a layer's opacity, the thumb's thickness and the thumb and track's opacity.
Every Fluent scrollbar is represented by a *PaintedScrollbarLayerImpl* layer object. For overlay animations, we care mainly about two properties of the layer object: its opacity and its scrollbar thumb thickness factor. Opacity is a value that goes from zero to one, and the thickness factor is a value that goes from *kIdleThickness* to one.

$$Opacity \in [0,\ 1]$$
$$thickness\_scale\_factor\_ \in [kIdleThickness,\ 1]$$

If the thumb's thickness scale factor value is *kIdleThickness*, that means the scrollbars are in *Minimal Mode* and the track and buttons should not be visible. If the scale factor is greater than *kIdleThickness*, then the scrollbar is either in Full Mode ($thumb\_thickness\_scale\_factor$ = 1) and the track and buttons should be fully opaque, or the thickness factor is somewhere in between its max and min value and should be drawn with a decreased opacity.

Thumb scale factor and track opacity:

Transitions between Minimal mode and Full mode are managed by an instance of *SingleScrollbarAnimationControllerThinning*. When a thinning animation is triggered, the Controller will interpolate the scrollbar's thumb thickness scale factor towards the desired value (depending on which type of animation is queued)[1]. When the scale factor is greater than its minimum value, then the layer object will interpolate the track's and button's opacity to create a fade in effect. The relation between the thumb_thickness_scale_factor_ and the track's opacity is as follows:
$(thickness\_scale\_factor\_ == kIdleThicknessScale) => (tracks\_opacity == 0)$
$(thickness\_scale\_factor\_ == 1) => (track\_opacity == 1)$
$tracks\_opacity = (thickness\_scale\_factor\_ - kIdleThicknessScale) / (1 - kIdleThicknessScale)$

Layer opacity:

The layer object's opacity is managed by the *ScrollbarAnimationController* which only manages transitions between *Invisible Mode* and the other two.
When the layer's opacity is set to one, that guarantees that the thumb is being shown, but that doesn't imply that the whole scrollbar is visible since its visibility depends on the thickness scale factor.

Testing:

Both overlay and non-overlay Fluent scrollbars are tested extensively by unit, browser and web tests.

Since this feature will be on by default for Windows, we anticipate a significant test churn due to the visual and layout differences*. There are a few options we can explore to deal with this issue. (*Please note that these are just suggestions/thoughts at this point and we're open to other ways of dealing with it. We've not fully thought through this yet.*)

- Use VirtualTestSuites: Start deep in the directory structure and then keep moving up. For e.g: start with fast/scrolling/scrollbars, update them to work with fluent scrollbars. The regular tests will likely fail. Add them to TestExpectations. Repeat this as you move up to fast/scrolling and then to fast/ and so on. Once all tests have been adapted, the flag can be turned on by default and the TestExpectations can be cleaned.
- Follow the Mac route of using mock scrollbars (window.internals.useMockOverlayScrollbars). The downside of doing this would be that the tests would not be testing the real thing.

To preserve test coverage for non-fluent scrollbars we can fork fast/scrolling/scrollbars and use VirtualTestSuites (with fluent scrollbars disabled).

TBD - Feature exposure:

How Fluent Overlay scrollbars will be exposed after the implementation is complete is yet to be decided.

The intention right now is to make the feature flag expose a new option in *chrome://settings* that will control the Fluent scrollbars mode.

Adding a new setting requires more overhead because of the increased exposure of the feature, as said in the docs:

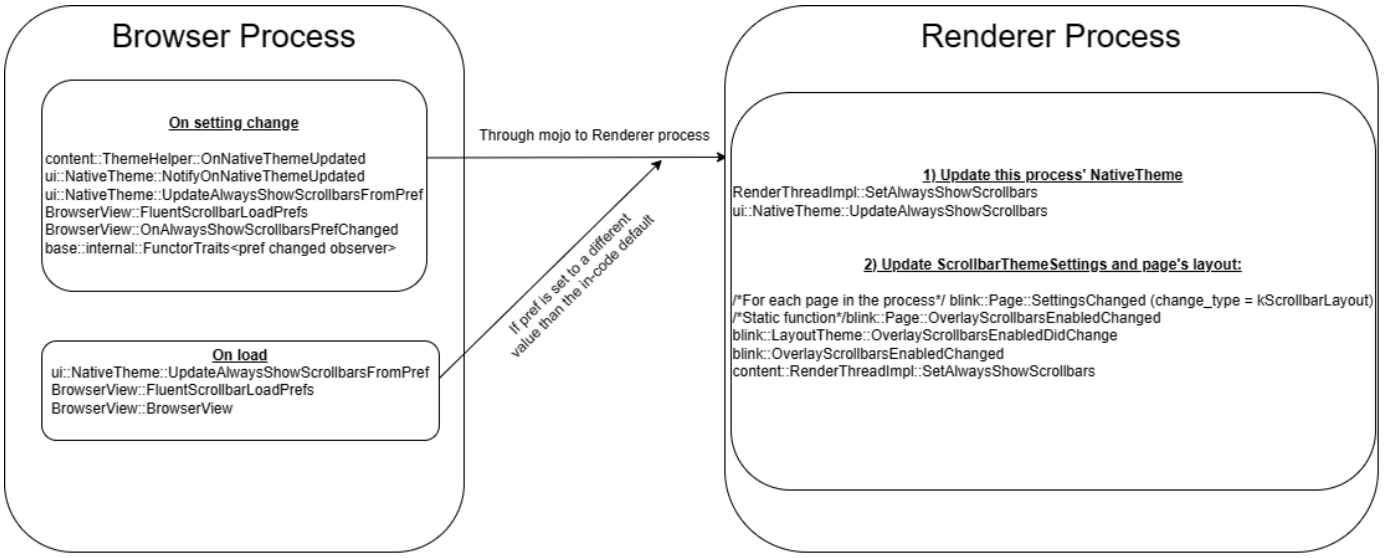
> ## *Settings*
>
> *Example: "Show home button"*
> *Settings are implemented in WebUI, and show up in chrome://settings or one of its subpages. They generally are bound to a pref which stores the value of that setting.* ***These are comparatively expensive to add, since they require localization and some amount of UX involvement to figure out how to fit them into chrome://settings, plus documentation and support material****. Many settings are implemented via prefs, but not all prefs correspond to settings; some are used for tracking internal browser state across restarts.*

Intention - Always show scrollbars:

The feature introduces a new setting in *chrome://settings/appearance* that allows a user to control the type of scrollbars and switch between overlay and non-overlay Fluent scrollbars.To see the differences in design, please visit the visual spec document linked above.

The setting will register a new *preference* which will determine the mode of the scrollbars. On startup, BrowserView will register an observer to oversee the changes in the pref, and set the correct overlay mode in the browser process' *NativeTheme*. *NativeTheme* will update its new flag that determines the status of this setting, and if it is different will call *NotifyOnNativeThemeUpdated(),* which will notify the renderer processes of the change in scrollbar mode. Through *ThemeHelper* and *RenderThreadImpl* the change will reach *Page,* which will update all the page's *ScrollbarTheme* and update the *ScrollbarThemeSettings::OverlayScrollbarsEnabled().*

On pref change, the observer will notify the browser process' *NativeTheme*, and through *NotifyOnNativeThemeUpdated()* the new setting will be spread to existing renderer processes



Outstanding issues:

1. Non-layered scrollbars don't animate since there's no equivalent to cc::ScrollbarAnimationController in Blink.

Potential follow-up work:

- **Implement similar behavior for regular aura scrollbars:** Fluent scrollbars code is mainly based on *Aura classes (ScrollbarThemeAura, NativeThemeAura, PaintedScrollbarLayer, etc). If upstream expresses interest in reworking the regular chromium scrollbars for them to use 9-patch and or paint on impl thread, we could see to implement it.