scroll-blocks-on

Influencing the scrolling performance vs. richness tradeoff

rbyers@chromium.org - Updated May 28, 2015
[public comments disabled due to excessive spam, e-mail for access]

Problem

In any system where scrolling occurs on a thread other than the main javascript UI thread, there is a fundamental tradeoff between performance (the smoothness of scrolling) and richness of effects (eg. the ability to synchronize effects precisely with the scroll position, or to start and stop scrolling at specific points). In chromium we've changed over time between a variety of models, and each browser has its own particular model.

It's becoming increasingly clear that there's no one-size-fits-all solution to this problem. Heavy 'desktop' document-centric sites viewed on a phone should probably always default to preferring scroll smoothness over control (and may want to opt-in to giving up even more control for the sake of scroll responsiveness, as is the case when touch scrolling in IE). Mobile web apps with rich UI effects (such as pull to refresh) may prefer to depend on a jank-free main thread (with all the perf tuning that entails) in order to get full control over scrolling. Note that increasing the apps ability to block scrolling on JavaScript is highly contentious (with engineers from IE and Safari in particular arguing against it, see public debates in public-pointer-events, www-style, and www-dom). The rest of this document assumes such capability is desirable.

In some cases, it may not even be possible to make a good choice in a document-wide fashion. For example, an image carousel component may be willing to take some risk of jank in order to implement desired snapping behavior (as many such components do today), but the use of such a component on a website shouldn't change the performance of scrolling elsewhere. Over time, a component may want to opt-in to a different mode without breaking sites that happen to also use other components designed for different modes.

Also there are scenarios (such as analytics) where a component wants to passively monitor input events without ever suppressing or blocking scrolling, but touch and wheel event handlers always have the capability to block scrolling so we must wait for them to run.

Use cases

Enable effects that respond to scroll position, such as:

- Scroll Header Panels
- parallax scrolling
- position: sticky
- Large viewporting scrollers, eg. in GitHub's Atom editor

Enable passive event monitoring without affecting scroll performance, such as for analytics purposes.

Related to new APIs for scroll customization (eg. pull-to-refresh) such as "beforescroll". See our extensible scrolling BlinkOn talk (especially the speaker notes) for more details.

Proposal: EventListenerOptions

See official proposal here.

This idea could possibly be extended to address the larger issues in this doc with two extensions:

1) A mechanism to opt-in to blocking scroll events on JavaScript. This could be an additional "synchronization" option. Or it could just be a new type of synchronous event, eg "beforescroll".

2) A (non-throwing) mechanism to determine in advance whether cancelable touch events are supported. Maybe a static "supportsCancelation" property on TouchEvent?

See this public-pointer-events discussion.

There are other unrelated options that could make sense to add in the future. Eg. a 'rate' option to specify what frequency of callbacks are desired for continuous events like touchmove and mousemove (to opt-in to high-precision movement or enable power-saving optimizations for low-frequency updates). Similarly a 'precision' option could be used to enable optimizations that avoid dispatching events for fine grained movement.

Obsolete proposal: new CSS property 'scroll-blocks-on'

A key problem with the scroll-blocks-on proposal below is that it doesn't compose well between different components. Eg. say an analytics framework wants to install a passive touch event listener on the document ('html { scroll-blocks-on: none }'), while some other component in the page wants synchronous scrolling for a scroll header effect ('html {scroll-blocks-on: scroll-event}'). The two components may fight over the value of documentElement.style.scrollBlocksOn. This gets even more complicated if components are being enabled/disabled throughout the lifetime of the page. Effectively you need to rely on a common component to mediate all the different requests to change the scrollBlocksOnMode. This is brittle and error prone.

Since all components must adhere to the same protocol, mediation between components is always best done by the platform itself. Fundamentally the issue is that the scroll-blocking setting is not a property of the element, but a property of the event listener. The code installing

the event listener should indicate at that time what properties the handler wants. Also, it should be possible for the system to refuse to grant the requested properties.

One general way to address this is with a new CSS property 'scroll-blocks-on' which is a list of activities which must complete before any scroll involving that element is displayed to the user. I propose the following definition:

‘scroll-blocks-on’

Value: none | [start-touch || wheel-event || scroll-event]
Initial: none
Applies to: all elements
Inherited: no

User-agent stylesheet (which matches the behavior prior to introduction of scroll-delay):

html { scroll-blocks-on: start-touch wheel-event }

When a scroll is targeted (i.e. on GestureScrollBegin for touch, and on each wheel event for mouse), we compute the union of all scroll-blocks-on values on the containing block chain of the target element. This enables changing behavior for a tree of elements, without the performance overhead of inherited style propagation.

Values have the following meanings:

- none
  - none of the below - always safe to scroll without waiting on main
  - cannot be combined with any of the values below
- start-touch
  - any cancelable touch events at the start of a scroll gesture block scrolling
    - On chromium this is the touchstart and first touchmove, but it differs in other browsers
  - without this, all touchstart and touchmove events become uncancelable, so they can be delivered to JavaScript asynchronously but calls to preventDefault on them are ignored.
- wheel-event
  - trackpad / mouse scrolling is blocked on wheel / mousewheel handlers at each frame
  - without this mousewheel events are only dispatched if the browser knows scrolling isn't possible
    - we could consider dispatching async wheel events during a scroll, but that would be a breaking change to the 'wheel' event definition and it's hard to see how it would provide any value over 'scroll' events.

        - this is needed whenever a page wants to consume wheel events on a scrollable surface
- scroll-event
    - 'scroll' events are delivered synchronously just before the scroll update becomes visible
    - if the handler modifies the scroll offset it's guaranteed to take effect before the original scroll offset becomes visible to the user. This enables, for example, a scroll event handler to counter-scroll a bit to add a friction effect to the scroll without risk of the user seeing the scroll position jitter back and forth.
    - can be used to overlay motion onto a fling, for example to implement snap points in a mostly input-agnostic fashion. Note that everything here is just about responding to scrolling, not feeding back into the input/scroll system at all (see beforescroll for that)
    - without this, scroll events may be sent async
    - this may be desired when implementing scroll-linked effects (such as parallax scrolling) when the developer wants to minimize the chance of ever getting a frame where the scroll offset is out-of-sync with it's effect.
    - Any synchronous scrolling APIs/events (such as the proposed 'beforescroll') would be enabled only when scrolling in this mode.

A user agent may choose to block in additional scenarios than just those required. A simple implementation may choose to use the union of all scroll-delay values in a document to set a global mode (eg. in Chrome I think we'd want to track modes at layer granularity). Similarly an implementation may choose to always block on some types of events regardless of the scroll-blocks-on setting, as long as the semantics are correct so that pages should not become broken if they're changed to asynchronous in the future. For example, in chromium we'd probably first implement the absence 'start-touch' by only affecting the cancelability of the events, and later (when we have touch-action hit-testing on our compositor thread) we could enable scrolling without blocking on these events.

Of course if no handlers of the specified types exist at a point, the user agent isn't required to block on the main thread. So, for example, touch-scrolling an element with 'scroll-blocks-on: start-touch' but without any touch handlers is equivalent to one with 'scroll-delay: none'. This means our existing optimizations (such as compositor touch hit testing) continue to be useful.

To enable efficient threaded implementations (for 'scroll-blocks-on: none' in particular) we could restrict scroll-blocks-on to be applicable to only block-level elements (as is the case for touch-action).

Most (all?) browsers can trigger scrolling during pinch-zoom, so this API affects the behavior/performance of pinch-zoom as well as scrolling. That is the things affected by scroll-blocks-on are the same as those affected by touch-action.

Mitigating performance risk

If used inappropriately (eg. on sites with heavy JS during scroll) 'scroll-blocks-on: scroll-event' could lead to substantially increased scroll jank. For example, developers may test their use of this API only on fast devices. User agents want a fallback to ensure they can keep scrolling minimally responsive.

Many browsers already implement a timeout to ensure scrolling can't be delayed indefinitely (in the 'scroll-blocks-on: start-touch' scenario). We can extend and improve that model to mitigate the performance risk of this API. When a user agent decides the requested scroll blocking mode cannot be achieved (possibly part way through a scroll gesture), it fires a non-bubbling UIEvent named 'scroll-block-timeout' at the element being scrolled and (possibly in parallel) enables free scrolling. Since this may result in UI glitches, this should be reserved for serious failures only.

In Chromium I propose starting with a simple 150ms timeout per scroll gesture (similar to the existing touch ACK timeout). If at any time during a scroll gesture, event handling took >150ms, we'd immediately switch to scrolling freely on the compositor thread. To easily recover from transient issues (eg. during page load), any subsequent scroll gesture would be unaffected. We'd then study performance in practice and refine the model to strike a reasonable balance between a good user experience and developer rationality. For a well behaving page, 150ms is an eternity (to be responsive, the main thread can be blocked for no more than 16ms at a time). So if this timeout gets triggered, the page is suffering from a catastrophic performance failure and extreme measures are not unreasonable.

Open questions

- Is element-based control necessary? Or could we get by with a document-wide ‘single-threaded mode’?
    - The main justification for element-based control is that different scenarios/components have different tradeoffs of richness vs. performance.
    - For example consider a relatively long document with an image carousel on it (eg. maybe a product description in a mobile store app). Scrolling the document has no need for customization but rock-solid 60fps is critical to the experience. But if the user swipes the image carousel, snapping behavior is critical to the experience, and developers may be willing to live with a higher risk of jank as a result. These scenarios are common today using touch events to drive the carousel and we already work hard (cc touch hit testing) to ensure that the touch handlers on the carousel don’t impede scrolling performance on the rest of the document.
    - So element-based control enables more incremental adoption of main-thread-blocking scroll. This way developers can get a “pay for play” return on their investment in performance work, rather than have a single all-or-nothing switch.

- Is a within-gesture timeout worth the cost in implementation complexity?
    - Transfering a scroll gesture from main to impl may be tricky to get right.
    - Could we get away with a simpler timeout model that didn't require this? Eg. perhaps disabling scroll-blocking for new gesture scrolls in the next minute would be sufficient. Users often lift and try scrolling again.
- Should we consider a JavaScript API (eg. Element.setScrollDelay) instead of a CSS API?
    - The main benefits of using CSS are:
        - There's an existing pattern and tooling for reasoning about inheritance down the DOM tree
        - It's easier to find and track sites that depend on this capability when developers declare it statically
- Should we specify 'wheel' and 'start-touch' now or just focus on scroll-event as the only knob?
    - If we want to leave wheel/touch as a potential future optimization, we'd need a (probably confusing) syntax for disabling scroll-delay modes that are on by default.
    - There's general agreement between vendors that we should be letting developers opt-out of blocking the start of scroll on touch events (this is one of the main benefits pointer events has over touch events).
    - It's an easier argument to make to say that developers should have full control over this policy in principle, rather than trying to argue they should be able to move it only in one particular direction.
    - Defining at least start-touch lets us explain, generalize, standardize and improve existing behavior which has been a source of confusion and pain - especially as it relates to time out policy.
- Is there any reasons to plan to implement full per-element control (eg. by extending compositor touch hit testing to track scroll-delay regions)? Or is relying on the union per GraphicsLayer good enough?
    - NO. Slimming pain will give us the finer granularity if desired. It's definitely not worth the complexity to add a new rect-tracking system at this stage.

Chromium Implementation notes

Each cc::Layer will have a scroll-blocks-on value. Applying any scroll-blocks-on value other than the default ('none') will cause the object to be promoted to a composited layer and ineligible for squashing. This makes the implementation relatively straight forward and should be OK from a performance perspective because the natural things to want to change scroll-blocks-on for are all likely to be composited layers already (the document, scrollers, elements translated from JS, etc). This overhead will disappear with slimming paint anyway.

In order to skip running touch handlers, we will need to implement touch-action hit testing on impl. touch-action should perhaps also cause composited layer promotion.

Sending touch events async will be a little tricky (will still need back-pressure for coalescing purposes). To start we can just omit sending touch events entirely when they shouldn't block scrolling.

Changing the scroll-blocks-on mode during a scroll gesture will be hard to implement (transferring the scroll between impl and main, preserving the correct scroll target). To start we'll only pay attention to the scroll-blocks-on mode at the start of a scroll.

The "show potential scroll bottlenecks" feature of devtools should be updated to show rects for 'scroll-blocks-on: scroll-event' and to suppress any touch/wheel handler rects that are disabled by scroll-blocks-on.

Timeout

- Goals:
    - GSB or any GSU times out, we switch back to impl scrolling
    - should scroll the same layer
    - scroll position shouldn't slip
    - possibly share logic with touch ACK timeout?
- Questions
    - How to preserve the scroll layer
        - CC has a 'ScrollOnMainThreadWithTimeout' mode
        - Saves the layer that would have scrolled
        - can be told to scroll after timeout - beginning as if ScrollBegin had decided to scroll on impl
    - How to avoid scroll slip
        - Can't cancel timed out GSU - race
        - Potentally handle GSUs on impl ahead of pending GSU on main?
        - Once pending GSU on main completes, offset will be caught up
        - Alternately, some mechanism for impl to ignore/undo main scrolls that were aborted?
        - Scroll sequence numbers. GSUs with number N is ignored if I've already processed one >N. Blink would need to plumb sequence number all the way through WebLayer::setScrollOffset
    - Can we guarantee scroll target match?
        - If DOM is in the process of changing, impl may latch one layer and blink another.
        - Perhaps blink should tell impl what the current gesture-scroll layer is?
        - Perhaps slimming paint will help?
    - Who trackers timer?
        - impl thread:
            - doesn't know when scroll is handled
            - timeout based on frame production? can't guarantee main will produce a frame. Maybe use latency tracking infrastructure?

            - browser:
                - doesn't know about scroll-block mode
                - sends at most one scroll timeout per gesture
                - impl thread determines if scroll timeout should do anything.
        - CC requires explicit 'assume' or free GSU routing
            - CC being prepared for a scroll to be transfered may be a similar state to cc executing a scroll without receiving any GSU events.
            - Is there value in having an explicit 'tranfer now' API on InputHandler?
            - Or should cc just support the more flexible design of blink and cc both being in active scrolling mode with GSUs going to one or the other as InputHandlerProxy sees fit?
            - Unless we can guarantee full fidelity, we shouldn't imply we can switch back and forth. So we should have an explicit transfer step.
        - Attempt to handle timeout during main thread fling?
            - No events are being handled in the browser / impl during this time.
            - Transferring the current state of a fling curve may be awkward
            - Ignore for now
- Concrete plan:
    - TryScroll return cases:
        - Scrolls on impl
        - Forces scrolling to occur on main
        - Enables main thread scroll, but doesn't itself scroll on impl
        - Enables main thread scroll and also scrolls on impl

Appendix - Potential additional properties

We could consider adding additional properties in the future to help fully explain the wide variety of behavior seen in different browsers. It's probably not worth the complexity to add most of these, but they're listed here for the sake of completeness.

- start-touch-timed
    - the touchstart and first touchmove event block scrolling but only up to a maximum timeout (eg. 150ms today)
    - we almost certainly don't want to specify such an irrational mode explicitly, but Chrome may need it for compatibility in some cases
- touch-overscroll
    - scrolling past the scroll limit triggers sending of touchmove events which can block scrolling
    - redundant if 'touchmove' is already applied
- touchmove
    - touchmove events are sent synchronously throughout the scroll - for each event, the disposition determines whether a scroll update is generated

    - without this or 'touchmove-overscroll', scrolling may trigger a touchcancel
- raster (?)
    - scroll updates are not displayed until all content for this node has been rasterized
    - gives control over the tradeoff between checkerboarding and scroll jank

This takes a big step towards explaining the different behavior we've had over time, and some of the behavior of other browsers. Eg:

- scroll-delay: start-touch wheel
    - This is our current behavior today (on mobile-viewport sites) - mousewheel handlers block all scroll updates, and touch event handlers block the start of scrolling but not the rest of the scroll updates.
- scroll-delay: start-touch-timed wheel
    - This is our current behavior on mobile when viewing a desktop website
- scroll-delay: start-touch touchmove scroll-event wheel raster
    - this is how we behaved on desktop before we had threaded scrolling - JS has full control and ability to jank scrolling
    - if we see this on the document (or event various subsets of it), we could potentially disable (or avoid spinning up entirely) the impl thread and just do everything main thread
    - This is effectively how Android native apps behave
- scroll-delay: start-touch touchmove wheel
    - This is how we behaved on desktop after we had threaded scrolling before we switched to the Android touchcancel model. When there are no touch handlers scrolling runs free on the thread, but as soon as you have a touchmove handler then every scroll updates blocks on the main thread.
    - I believe this is how safari behaves on divs without -webkit-overflow-scrolling: touch
    - this is effectively how the android browser behaves, except with start-touch-timed
- scroll-delay: start-touch touch-overscroll wheel
    - this is what I'm proposing we change our default behavior to with 'touchmove absorption'
- scroll-delay: wheel
    - the equivalent of 'touch-action-delay: none' which I was previously proposing (and so this proposal supersedes that).
    - This is IE's behavior
- scroll-delay: none
    - this is touch-action-delay: none generalized to also apply to mousewheel
    - IE folks in the PEWG agree with me that we might some day want to tackle the mousewheel equivalent problem of touch-action - eg. with a 'wheel-action' property. It's the same fundamental issue as with touch, but the psychological effect (and slower mobile devices) make the touch case much higher priority.