Introducing visualViewport

By Jake Archibald
Human boy working on web standards at Google

What if I told you, there's more than one viewport.

BRRRRAAAAAAAMMMMMMMMMM

And the viewport you're using right now, is actually a viewport within a viewport.

BRRRRAAAAAAAMMMMMMMMMM

And sometimes, the data the DOM gives you, refers to one of those viewport and not the other.

BRRRRAAAAM... wait what?

It's true, take a look:

Layout viewport vs visual viewport

The video above shows a web page being scrolled and pinch-zoomed, along with a mini-map on the right showing the position of viewports within the page.

Things are pretty straight forward during regular scrolling. The green area represents the *layout viewport*, which `position: fixed` items stick to.

Things get weird when pinch-zooming is introduced. The red box represents the *visual viewport*, which is the part of the page we can actually see. This viewport can move around while `position: fixed` elements remain where they were, attached to the layout viewport. If we pan at a boundary of the layout viewport, it drags the layout viewport along with it.

Improving compatibility

Unfortunately web APIs are inconsistent in terms of which viewport they refer to, and they're also inconsistent across browsers.

For instance, `element.getBoundingClientRect().y` returns the offset within the *layout viewport*. That's cool, but we often want the position within the page, so we write:

```
element.getBoundingClientRect().y + window.scrollY
```

However, many browsers use the *visual viewport* for `window.scrollY`, meaning the above code breaks when the user pinch-zooms.

Chrome 61 changes `window.scrollY` to refer to the layout viewport instead, meaning the above code works even when pinch-zoomed. In fact, browsers are slowly changing all positional properties to refer to the layout viewport.

With the exception of one new property…

Exposing the visual viewport to script

A new API exposes the visual viewport as `window.visualViewport`. It's a draft spec, with cross-browser approval, and it's landing in Chrome 61.

```
console.log(window.visualViewport.width);
```

Here's what `window.visualViewport` gives us:

`visualViewport` properties

`offsetLeft`	Distance between the left edge of the visual viewport, and the layout viewport, in CSS pixels.
`offsetTop`	Distance between the top edge of the visual viewport, and the layout viewport, in CSS pixels.
`pageLeft`	Distance between the left edge of the visual viewport, and the left boundary of the document, in CSS pixels.
`pageTop`	Distance between the top edge of the visual viewport, and the top boundary of the document, in CSS pixels.
`width`	Width of the visual viewport in CSS pixels.
`height`	Height of the visual viewport in CSS pixels.
`scale`	The scale applied by pinch-zooming. If content is twice the size due to zooming, this would return `2`. This is not affected by `devicePixelRatio`.

There are also a couple of events:

```
window.visualViewport.addEventListener('resize', listener);
```

`visualViewport` events

`resize`	Fired when `width`, `height`, or `scale` changes.
`scroll`	Fired when `offsetLeft` or `offsetTop` changes.

Demo

The video at the start of this article was created using `visualViewport`, check it out in Chrome 61+. It uses `visualViewport` to make the mini-map stick to the top-right of the visual viewport, and applies an inverse scale so it always appears the same size, despite pinch-zooming.

Gotchas

Events only fire when the visual viewport changes

It feels like an obvious thing to state, but it caught me out when I first played with `visualViewport`.

If the layout viewport resizes but the visual viewport doesn't, you don't get a `resize` event. However, it's unusual for the layout viewport to resize without the visual viewport also changing width/height.

The real gotcha is scrolling. If scrolling occurs, but the visual viewport remains static *relative to the layout viewport*, you don't get a `scroll` event on `visualViewport`, and this is really common. During regular document scrolling, the visual viewport stays locked to the top-left of the layout viewport, so `scroll` *does not fire* on `visualViewport`.

If you're wanting to hear about all changes to the visual viewport, including `pageTop` and `pageLeft`, you'll have to listen to the window's scroll event too:

```
visualViewport.addEventListener('scroll', update);
visualViewport.addEventListener('resize', update);
window.addEventListener('scroll', update);
```

Avoid duplicating work with multiple listeners

Similar to listening to `scroll` & `resize` on the window, you're likely to call some kind of "update" function as a result. However, it's common for many of these events to happen at the same time. If the user resizes the window, it'll trigger `resize`, but quite often `scroll` too. To improve performance, avoid handling the change multiple times:

```
// Add listeners
visualViewport.addEventListener('scroll', update);
visualViewport.addEventListener('resize', update);
addEventListener('scroll', update);

let pendingUpdate = false;

function update() {
  // If we're already going to handle an update, return
  if (pendingUpdate) return;

  pendingUpdate = true;

  // Use requestAnimationFrame so the update happens before next render
  requestAnimationFrame(() => {
    pendingUpdate = false;

    // Handle update here
  });
}
```

I've filed a spec issue for this, as I think there may be a better way, such as a single `update` event.

Event handlers don't work

Due to a Chrome bug, this *does not work*:

Buggy – uses an event handler

```
visualViewport.onscroll = () => console.log('scroll!');
```

Instead:

Works – uses an event listener

```
visualViewport.addEventListener('scroll', () => console.log('scroll'));
```

Offset values are rounded

I think (well, I hope) this is another Chrome bug.

`offsetLeft` and `offsetTop` are rounded, which is pretty inaccurate once the user has zoomed in. You can see the issues with this during the demo – if the user zooms in and pans slowly, the mini-map snaps between unzoomed pixels.

The event rate is slow

Like other `resize` and `scroll` events, these no not fire every frame, especially on mobile. You can see this during the demo – once you pinch zoom, the mini-map has trouble staying locked to the viewport.

Accessibility

In the demo I used `visualViewport` to counteract the user's pinch-zoom. It makes sense for this particular demo, but you should think carefully before doing anything that overrides the user's desire to zoom in.

`visualViewport` can be used to improve accessibility. For instance, if the user is zooming in, you may choose to hide decorative `position: fixed` items, to get them out of the user's way. But again, be careful you're not hiding something the user is trying to get a closer look at.

You could consider posting to an analytics service when the user zooms in. This could help you identify pages that users are having difficulty with at the default zoom level.

```
visualViewport.addEventListener('resize', () => {
  if (visualViewport.scale > 1) {
    // Post data to analytics service
  }
});
```

And that's it! `visualViewport` is a nice little API which solves compatibility issues along the way.
