Pointing the Way Forward

By Sérgio Gomes
Web DevRel @ Google

Pointing at things on the web used to be simple. You had a mouse, you moved it around, sometimes you pushed buttons, and that was it. Everything that wasn't a mouse was emulated as one, and developers knew exactly what to count on.

Simple doesn't necessarily mean good, though. Over time, it became increasingly important that not everything was (or pretended to be) a mouse: you could have pressure-sensitive and tilt-aware pens, for amazing creative freedom; you could use your fingers, so all you needed was the device and your hand; and hey, why not use more than one finger while you're at it?

We've had touch events for a while to help us with that, but they're an entirely separate API specifically for touch, forcing you to code two separate event models if you want to support both mouse and touch. Chrome 55 ships with a newer standard that unifies both models: pointer events.

A single event model

Pointer events unify the pointer input model for the browser, bringing touch, pens, and mice together into a single set of events. For example:

```
document.addEventListener('pointermove',
  ev => console.log('The pointer moved.'));
foo.addEventListener('pointerover',
  ev => console.log('The pointer is now over foo.'));
```

Here's a list of all the available events, which should look pretty familiar if you're familiar with mouse events:

`pointerover`	The pointer has entered the bounding box of the element. This happens immediately for devices that support hover, or before a **`pointerdown`** event for devices that do not.
`pointerenter`	Similar to **`pointerover`**, but does not bubble and handles descendants differently. Details on the spec.

`pointerdown`	The pointer has entered the active button state, with either a button being pressed or contact being established, depending on the semantics of the input device.
`pointermove`	The pointer has changed position.
`pointerup`	The pointer has left the active button state.
`pointercancel`	Something has happened that means it's unlikely the pointer will emit any more events. This means you should cancel any in-progress actions and go back to a neutral input state.
`pointerout`	The pointer has left the bounding box of the element or screen. Also after a `pointerup`, if the device does not support hover.
`pointerleave`	Similar to `pointerout`, but does not bubble and handles descendants differently. Details on the spec.
`gotpointercapture`	Element has received pointer capture.
`lostpointercapture`	Pointer which was being captured has been released.

Note: Pointer events are confusingly unrelated to the `pointer-events CSS property`. Even worse, the two can be used together! The behaviour of `pointer-events` (the CSS property) with pointer events (the event model) is no different than with mouse events or touch events, so if you've used that CSS property before, you know what to expect.

Different input types

Generally, Pointer Events allow you to write code in an input-agnostic way, without the need to register separate event handlers for different input devices. Of course, you'll still need to be mindful of the differences between input types, such as whether the concept of hover applies. If you do want to tell different input device types apart – perhaps to provide separate code/functionality for different inputs – you can however do so from within the same event handlers using the `pointerType` property of the `PointerEvent` interface. For example, if you were coding a side navigation drawer, you could have the following logic on your `pointermove` event:

```
switch(ev.pointerType) {
  case 'mouse':
    // Do nothing.
    break;
  case 'touch':
    // Allow drag gesture.
    break;
  case 'pen':
    // Also allow drag gesture.
    break;
  default:
    // Getting an empty string means the browser doesn't know
    // what device type it is. Let's assume mouse and do nothing.
    break;
}
```

Default actions

In touch-enabled browsers, certain gestures are used to make the page scroll, zoom, or refresh. In the case of touch events, you will still receive events while these default actions are taking place – for instance, `touchmove` will still be fired while the user is scrolling.

With pointer events, whenever a default action like scroll or zoom is triggered, you'll get a `pointercancel` event, to let you know that the browser has taken control of the pointer. For example:

```
document.addEventListener('pointercancel',
  ev => console.log('Go home, the browser is in charge now.'));
```

Built-in speed: This model allows for better performance by default, compared to touch events, where you would need to use passive event listeners to achieve the same level of responsiveness.

You can stop the browser from taking control with the `touch-action` CSS property. Setting it to `none` on an element will disable all browser-defined actions started over that element. But there are a number of other values for finer-grained control, such as `pan-x`, for allowing the browser to react to movement on the x axis but not the y axis. Chrome 55 supports the following values:

`auto`	Default; the browser can perform any default action.
`none`	The browser isn't allowed to perform any default actions.
`pan-x`	The browser is only allowed to perform the horizontal scroll default action.
`pan-y`	The browser is only allowed to perform the vertical scroll default action.
`pan-left`	The browser is only allowed to perform the horizontal scroll default action, and only to pan the page to the left.
`pan-right`	The browser is only allowed to perform the horizontal scroll default action, and only to

	pan the page to the right.
pan-up	The browser is only allowed to perform the vertical scroll default action, and only to pan the page up.
pan-down	The browser is only allowed to perform the vertical scroll default action, and only to pan the page down.
manipulation	The browser is only allowed to perform scroll and zoom actions.

Pointer capture

Ever spent a frustrating hour debugging a broken **mouseup** event, until you realised that it's because the user is letting go of the button outside your click target? No? Okay, maybe it's just me, then.

Still, until now there wasn't a really good way of tackling this problem. Sure, you could set up the **mouseup** handler on the document, and save some state on your application to keep track of things. That's not the cleanest solution, though, particularly if you're building a web component and trying to keep everything nice and isolated.

With pointer events comes a much better solution: you can capture the pointer, so that you're sure to get that **pointerup** event (or any other of its elusive friends).

```
const foo = document.querySelector('#foo');
foo.addEventListener('pointerdown', ev => {
  console.log('Button down, capturing!');
  // Every pointer has an ID, which you can read from the event.
  foo.setPointerCapture(ev.pointerId);
});

foo.addEventListener('pointerup',
  ev => console.log('Button up. Every time!'));
```

Browser support

At the time of writing, Pointer Events are supported in Internet Explorer 11, Microsoft Edge, Chrome, and Opera, and partially supported in Firefox. You can find an up-to-date list at caniuse.com.

You can use the Pointer Events polyfill for filling in the gaps. Alternatively, checking for browser support at runtime is straightforward:

```
if (window.PointerEvent) {
  // Yay, we can use pointer events!
} else {
  // Back to mouse and touch events, I guess.
}
```

Pointer events are a fantastic candidate for progressive enhancement: just modify your initialization methods to make the check above, add pointer event handlers in the `if` block, and move your mouse/touch event handlers to the `else` block.

So go ahead, give them a spin and let us know what you think!