https://developer.mozilla.org/en-US/docs/Web/API/Pointer_events

Most of today's web content assumes the user's pointing device will be a mouse. However, since many devices support other types of pointing input devices, such as pen/stylus and touch surfaces, extensions to the existing pointing device event models are needed and *pointer events* address that need.

Pointer events are DOM events that are fired for a pointing device. They are designed to create a single DOM event model to handle pointing input devices such as a mouse, pen/stylus or touch (such as one or more fingers).
The *pointer* is a hardware-agnostic device that can target a specific set of screen coordinates. Having a single event model for pointers can simplify creating Web sites and applications and provide a good user experience regardless of the user's hardware. However, for scenarios when device-specific handling is desired, pointer events defines a property to inspect the device type which produced the event.
The events needed to handle generic pointer input are analogous to mouse events (mousedown/pointerdown, mousemove/pointermove, etc.). Consequently, pointer event types are intentionally similar to mouse event types. Additionally, a pointer event contains the usual properties present in mouse events (client coordinates, target element, button states, etc.) in addition to new properties for other forms of input: pressure, contact geometry, tilt, etc. In fact, the PointerEvent interface inherits all of the MouseEvent's properties thus facilitatig migrating content from mouse events to pointer events.

Terminology

active pointer
Any *pointer* input device that can produce events. A pointer is considered active if it can still produce further events. For example a pen that is a down state is considered active because it can produce additional events when the pen is lifted or moved.
digitizer
A sensing device with a surface that can detect contact. Most commonly, the sensing device is a touch enable screen that can sense input from an input device such as a pen, stylus of finger.
hit test
The process the browser uses to determine a target element for a pointer event. Typically, this is determined by considering the pointer's location and also the visual layout of elements in a document on screen media.
pointer
A hardware agnostic representation of input devices that can target a specific coordinate (or set of coordinates) on a screen. Examples of *pointer* input devices are mouse, pen/stylus, and touch contacts.
pointer capture
Pointer capture allows the events for a pointer to be retargeted to a particular element other than the normal hit test result of the pointer's location.
pointer event
A DOM event fired for a *pointer*.

Interfaces

The primary interface is the PointerEvent interface which has a constructor plus several event types and associated global event handlers. The standard also includes some extensions to the Element and Navigator interfaces. The following sub-sections contain short descriptions of each interface and property.

PointerEvent interface

The PointerEvent interface extends the MouseEvent interface and has the following properties (all of them are Read only).

- pointerId - a unique identifier for the pointer causing the event.
- width - the width (magnitude on the X axis), in CSS pixels, of the contact geometry of the pointer.
- height - the height (magnitude on the Y axis), in CSS pixels, of the contact geometry of the pointer.
- pressure - the normalized pressure of the pointer input in the range of 0 to 1, where 0 and 1 represent the minimum and maximum pressure the hardware is capable of detecting, respectively.
- tiltX - the plane angle (in degrees, in the range of -90 to 90) between the Y-Z plane and the plane containing both the transducer (e.g. pen stylus) axis and the Y axis.
- tiltY - the plane angle (in degrees, in the range of -90 to 90) between the X-Z plane and the plane containing both the transducer (e.g. pen stylus) axis and the X axis.
- pointerType - indicates the device type that caused the event (mouse, pen, touch, etc.)
- isPrimary - indicates if the pointer represents the primary pointer of this pointer type.

Event types and Global Event Handlers

Pointer events have ten event types, seven of which have similar semantics to their mouse event counterpart (down, up, move, over, out, enter, leave). Below is a short description of each event type and its associated Global Event Handler.

Event	On Event Handler	Description
pointerover	onpointerover	fired when a pointing device is moved into an element's hit test boundaries.
pointerenter	onpointerenter	fired when when a pointing device is moved into the hit test boundaries of an element or one of its descendants, including as a result of a pointerdown event from a device that does not support hover (see pointerdown).
pointerdown	onpointerdown	fired when a pointer becomes *active*.
pointermove	onpointermove	fired when a pointer changes coordinates.
pointerup	onpointerup	fired when a pointer is no longer *active*.
pointercancel	onpointercancel	a browser fires this event if it concludes the pointer will no longer be able to generate events (for example the related device is deactived).
pointerout	onpointerout	fired for several reasons including: pointing device is moved out of the hit test boundaries of an element; firing the pointerup event for a device that does not support hover (see pointerup); after firing the pointercancel event (see pointercancel); when a pen stylus leaves the hover range detectable by the digitizer.
pointerleave	onpointerleave	fired when a pointing device is moved out of the hit test boundaries of an element. For pen devices, this event is fired when the stylus leaves the hover range detectable by the digitizer.
gotpointercapture	None (see Element extensions)	fired when an element receives pointer capture.
lostpointercapture	None (see Element extensions)	fired after pointer capture is released for a pointer.

Element extensions

There are four extensions to the Element interface:

- ongotpointercapture - an EventHandler that returns the event handler (function) for the gotpointercapture event type.
- onlostpointercapture - an EventHandler interface that returns the event handler (function) for the lostpointercapture event type.
- setPointerCapture() - this method designates a specific element as the *capture target* of future pointer events.
- releasePointerCapture() - the method releases (stops) *pointer capture* that was previously set for a specific pointer event.

Navigator extension

The Navigator.maxTouchPoints property is used to determine the maximum number of simultaneous touch points that are supported at any single point in time.

Examples

This section contains examples of basic usage of using the pointer events interfaces.

Registering event handlers

This example registers a handler for every event type for the given element.

```
<html>
<script>
function over_handler(event) { }
function enter_handler(event) { }
function down_handler(event) { }
function move_handler(event) { }
function up_handler(event) { }
function cancel_handler(event) { }
function out_handler(event) { }
function leave_handler(event) { }
function gotcapture_handler(event) { }
function lostcapture_handler(event) { }

function init() {
 var el=document.getElementById("target");
 // Register pointer event handlers
 el.onpointerover = over_handler;
 el.onpointerenter = enter_handler;
 el.onpointerdown = down_handler;
 el.onpointermove = move_handler;
 el.onpointerup = up_handler;
 el.onpointercancel = cancel_handler;
 el.onpointerout = out_handler;
 el.onpointerleave = leave_handler;
 el.gotpointercapture = gotcapture_handler;
 el.lostpointercapture = lostcapture_handler;
}
</script>
<body onload="init();">
<div id="target"> Touch me ... </div>
</body>
</html>
```

Event properties

This example illustrates accessing all of a touch event's properties.

```
<html>
<script>
var id = -1;

function process_id(event) {
  // Process this event based on the event's identifier
}
```

```
function process_mouse(event) {
  // Process the mouse pointer event
}
function process_pen(event) {
  // Process the pen pointer event
}
function process_touch(event) {
  // Process the touch pointer event
}
function process_tilt(tiltX, tiltY) {
  // Tilt data handler
}
function process_pressure(pressure) {
  // Pressure handler
}
function process_non_primary(event) {
  // Pressure handler
}

function down_handler(ev) {
 // Calculate the touch point's contact area
 var area = ev.width * ev.height;

 // Compare cached id with this event's id and process accordingly
 if (id == ev.identifier) process_id(ev);

 // Call the appropriate pointer type handler
 switch (ev.pointerType) {
   case "mouse":
     process_mouse(ev);
     break;
   case "pen":
     process_pen(ev);
     break;
   case "touch":
     process_touch(ev);
     break;
   default:
     console.log("pointerType " + ev.pointerType + " is Not suported");
 }

 // Call the tilt handler
 if (ev.tiltX != 0 && ev.tiltY != 0) process_tilt(ev.tiltX, ev.tiltY);

 // Call the pressure handler
 process_pressure(ev.pressure);

 // If this event is not primary, call the non primary handler
 if (!ev.isPrimary) process_non_primary(evt);
}

function init() {
 var el=document.getElementById("target");
 // Register pointerdown handler
 el.onpointerdown = down_handler;
}
</script>
<body onload="init();">
 <div id="target"> Touch me ... </div>
</body>
</html>
```

Determining the Primary Pointer

In some scenarios there may be multiple pointers (for example a device with both a touchscreen and a mouse) or a pointer supports multiple contact points (for example a touchscreen that supports multiple finger touches). The application can use the isPrimary property to identify a master pointer among the set of *active pointers* for each pointer type. If an application only wants to support a primary pointer, it can ignore all pointer events that are not primary.
For mouse, there is only one pointer so it will always be the primary pointer. For touch input, a pointer is considered primary if the user touched the screen when there were no other active touches. For pen and stylus input, a pointer is considered primary if the user's pen initially contacted the screen when there were no other active pens contacting the screen.

Determining button states

Some pointer devices, such as mouse and pen, support multiple buttons and the button presses can be *chorded* i.e. depressing an additional button while another button on the pointer device is already depressed. To determine the state of button presses, pointer events uses the button and buttons properties of the MouseEvent interface (that PointerEvent inherits from). The following table provides the values of button and buttons for the various device button states.

Device Button State	button	buttons
Mouse move with no buttons pressed	-1	0
Left Mouse, Touch Contact, Pen contact (with no modifier buttons pressed)	0	1
Middle Mouse	1	4
Right Mouse, Pen contact with barrel button pressed	2	2
X1 (back) Mouse	3	8
X2 (forward) Mouse	4	16
Pen contact with eraser button pressed	5	32

Pointer capture

Pointer capture allows events for a particular pointer event to be re-targeted to a particular element instead of the normal hit test at a pointer's location. This can be used to ensure that an element continues to receive pointer events even if the pointer device's contact moves off the element (for example by scrolling).
The following example shows pointer capture being set on an element.

```
<html>
<script>
function downHandler(ev) {
 var el=document.getElementById("target");
 //Element 'target' will receive/capture further events
 el.setPointerCapture(ev.pointerId);
}
function init() {
 var el=document.getElementById("target");
 el.onpointerdown = downHandler;
}
</script>
<body onload="init();">
<div id="target"> Touch me ... </div>
</body>
</html>
```

The following example shows a pointer capture being released (when a pointercancel event occurs. The browser does this automatically when a pointerup or pointercancel event occurs.

```
<html>
<script>
function downHandler(ev) {
 var el=document.getElementById("target");
 // Element "target" will receive/capture further events
 el.setPointerCapture(ev.pointerId);
}
function cancelHandler(ev) {
 var el=document.getElementById("target");
 // Release the pointer capture
 el.releasePointerCapture(ev.pointerId);
}
function init() {
 var el=document.getElementById("target");
 // Register pointerdown and pointercancel handlers
 el.onpointerdown = downHandler;
 el.onpointercancel = cancelHandler;
}
</script>
<body onload="init();">
<div id="target"> Touch me ... </div>
</body>
</html>
```

touch-action property

The touch-action CSS property is used to specifiy whether or not the browser should apply its default (*native*) touch behavior (such as zooming or panning) to a region. This property may be applied to all elements except: non-replaced inline elements, table rows, row groups, table columns, and column groups.
A value of auto means the browser is free to apply its default touch behavior (to the specified region) and the value of none disables the browser's default touch behavior for the region. The values pan-x and pan-y, mean that touches that begin on the specified region are only for horizontal and vertical scrolling, respectively. The value manipulation means the browser may consider touches that begin on the element are only for scrolling and zooming.
In the following example, the browser's default touch behavior is disabled for the div element.

```
<html>
<body>
 <div style="touch-action:none;">Can't touch this ... </div>
</body>
</html>
```

In the following example, default touch behavior is disabled for some button elements.

```
button#tiny {
  touch-action: none;
}
```
In the following example, when the target element is touched, it will only pan in the horizontal direction.
```
#target {
  touch-action: pan-x;
}
```

Compatibility with mouse events

Although the pointer event interfaces enable applications to create enhanced user experiences on pointer enabled devices, the reality is the vast majority of today's web content is designed to only work with mouse input. Consequently, even if a browser supports pointer events, the browser must still process mouse events so content that assumes mouse-only input will work as is without direct modification. Ideally, a pointer enabled application does not need to explicitly handle mouse input. However, because the browser must process mouse events, there may be some compatibility issues that need to be handled. This section contains information about pointer event and mouse event interaction and the ramifications for application developers.
The browser *may map generic pointer input to mouse events for compatibility with mouse-based content*. This mapping of events is called *compatibility mouse events*. Authors can prevent the production of certain compatibility mouse events by canceling the pointerdown event but note that:

- Mouse events can only be prevented when the pointer is down.
- Hovering pointers (e.g. a mouse with no buttons pressed) cannot have their mouse events prevented.
- The mouseover, mouseout, mouseenter, and mouseleave events are never prevented (even if the pointer is down).

Best practices

Here are some *best practices* to consider when using pointer events:

- Minimize the amount of work done that is done in the event handlers.
- Add the event handlers to a specific target element (rather than the entire document or nodes higher up in the document tree).
- The target element (node) should be large enough to accommodate the largest contact surface area (typically a finger touch). If the target area is too small, touching it could result in firing other events for adjacent elements.

Implementation and deployment status

The pointer events browser compatibility data indicates pointer event support among desktop and mobile browsers is relatively low, although additional implementations are in progress.
Some new value have been proposed for the css touch-action property as part of Pointer Events Level 2 specification but currently those new values have no implementation support.