InputDeviceCapabilities API sketch

rbyers@chromium.org - Mar 6, 2015

Summary

This document provides a rough sketch of what a future 'InputDeviceCapabilities' API for the web might look like. There are lots of details here that deserve a lot of debate, but the goal now is just to get an idea of the possible shape to help decide if we should take some simple initial steps in this direction.

An initial simple subset of this API has now shipped in Chrome.

Problem

DOM input events are an abstraction above low-level input events, only loosely tied to physical device input (eg. 'click' events can be fired by a mouse, touchscreen or keyboard). There is no mechanism to drill down and get lower-level details about the physical device responsible for an event. This tends to lead to a variety of problems with developers needing to make assumptions or rely on heuristics. For one concrete example, see Identifying mouse events derived from touch events.

Related patterns in other platforms

Windows 8

Windows Runtime has a PointerDevice class, which is correlated to a pointer input event via Windows.UI.Input.PointerPoint.PointerDevice. Additional information is available from the MouseCapabilities, KeyboardCapabilities and TouchCapabilities classes. But note that this information is not available per-device. If the system has two mice, for example, you can apparently determine whether any of them have a horizontal wheel, but not whether all of them do.

At the low level, all WM_POINTER event messages provide access to a sourceDevice HANDLE for each point which can be used to get device details through GetRawInputDeviceInfo.

Mac OS X

MacOS X has various properties directly on NSEvent for determining device details. Eg. the mouse event sub-type, hasPreciseScrollingDeltas, and various properties (such as vendorPointingDeviceType) for tablet device information.

Android

Android has a rich InputDevice API similar to what's being proposed here. Applications can determine the InputDevice for an event via InputEvent.getDevice, and can watch for changes in input devices using an InputDeviceManager.

Design sketch

Here is a rough incomplete outline of what such an API might look like. There are lots of details that could be debated.

```
partial interface UIEvent {
       // The capabilities of physical device responsible for the generation of this event, or
      // null if no physical device was responsible.
       readonly attribute InputDeviceCapabilities? sourceCapabilities;
};

partial dictionary UIEventInit {
       InputDeviceCapabilities? sourceCapabilities = null;
};

// Note that a touchscreen keyboard is still seen as a keyboard device.
enum DeviceClass {
       “keyboard”,
       “mouse”,
       “touchscreen”,
       “pen”,
       “gamepad”
};

// How precisely a pointer can point to a location on the screen.
// Corresponds to the pointer and any-pointer media features.
enum PointerGranularityClass {
       // Not a pointing device
       “none”,
       // Can reliably point at least at individual CSS px units (eg. a mouse / stylus)
       “fine”,
       // Requires larger targets for reliable activation (eg. a touchscreen)
       “coarse”,
};

// The extent to which a pointer can hover over a location on the screen.
// Corresponds to the hover and any-hover media features.
enum PointerHover {
       // Not a pointing device, or incapable of hover.
       “none”,
       // A pointing device that is capable of hovering.
       “hover”,
       // A device which can ‘hover’ only through some deliberate act of the user
       // (eg. a long press gesture on a touchscreen).
       “on-demand”
};
```

```
// Represents capabilities of input device (or group of related devices).
// To ease implementation, developers cannot rely on comparing two InputDeviceCapabilities
// instances for equality.  Eg. two mice with the same properties in a system may
// appear as a single InputDeviceCapabilities instance.
[Constructor(optional InputDeviceCapabilitiesInit deviceInitDict)]
interface InputDeviceCapabilities {
	// What type of device is this.  Can be one of the standard values or a
	// script-defined extension.
	readonly attribute (DeviceClass or DOMString) deviceClass;

	// How precisely can this device point to a location on the screen.
	readonly attribute PointerGranularityClass pointerGranularityClass;

	// Approximate granularity in CSS px.  A box with this width and height is the smallest
	// target the application should expect a user to be able to activate reliably.
	readonly attribute long pointerGranularity;

	// Whether the device is capable of hovering over a location on the screen.
	readonly attribute PointerHover pointerHover;

	// When the device supports vertical scroll input.
	readonly attribute boolean canScrollVertically

	// Whether the device supports horizontal scroll input.  For touchscreens and
	// touchpads this is generally true.  For traditional mice with a non-tilting wheel this is
	// false.
	readonly attribute boolean canScrollHorizontally;

	// Whether scrolls generated by this device are accurate at least down to a
	// single CSS px unit.  This is true for smooth scrolling devices like a touchpad
	// but false for traditional mouse wheels.
	readonly attribute boolean scrollsPrecisely;

	// Whether this device dispatches touch events for movement.  This is used to detect
	// mouse events which represent only an action that has already been handled  by
	// touch event handlers.
	readonly attribute boolean firesTouchEvents;

	// The maximum number of simultaneous contacts (eg. touch points) supported by
	// the device, or 0 for non-pointer devices.
	readonly attribute unsigned long maxContacts;

	// True if this device (eg. keyboard) is implemented by consuming space on a screen
	// otherwise potentially available to the application.
	// TODO: Should this be a keyboard type enum?
	readonly attribute boolean isOnScreen;

	// When the above is true, this will be true if the device is currently being displayed on
	// the screen.
	readonly attribute boolean isCurrentlyOnScreen;
```

```
        // Return all of the (unique) input devices.
        static sequence<InputDeviceCapabilities> getAllDevicesCapabilities();

        // Return all of the input devices with the specified deviceClass string.
        static sequence<InputDeviceCapabilities> getDeviceCapabilitiesOfClass(string class);

        // APIs to be notified when input devices Capabilities are added or removed.
        // Do we also need some sort of properties-changed notification?
        // Perhaps these should be just DOM events fired at the window (like
        // 'gamepadconnected' etc.).
        static addDeviceAddedListener(InputDeviceCapabilitiesListener listener);
        static removeDeviceAddedListener(InputDeviceCapabilitiesListener listener);
        static addDeviceRemovedListener(InputDeviceCapabilitiesListener listener);
        static removeDeviceRemovedListener(InputDeviceCapabilitiesListener listener);
};

callback InputDeviceCapabilitiesListener = void (InputDeviceCapabilities inputDeviceCapabilities);

dictionary InputDeviceCapabilitiesInit {
        ...
};
```

Open questions

For TouchEvents, should each Touch have it's own InputDeviceCapabilities**?**
Windows supports pointer events where points from multiple devices are changed at the same time. The only bundled input API like this on the web is TouchEvents. Arguably each Touch object should have it's own sourceCapabilities property (rather than all referring to the same one on the TouchEvent). I believe this would add unnecessary complexity. If two touches are really changing on two input devices simultaneously, then there should be no harm in firing separate touchMove events for these (with both points available in the 'touches' array, but each event having only one changedTouches entry). I.e. two touches moving simultaneously on different devices is equivalent to one touch moving immediately after the other, there is little value in requiring these be bundled together in a single event.

Do we need a way of indicating "unknown"?
Some platforms may not supply all the information necessary to reliably provide accurate values for all of these APIs. What should the guidance be to implementors on such platforms? Should we support 'undefined' return values or selectively missing properties in some environments?

What exactly is the connection to the GamePad spec?
Do GamePad devices ever generate UIEvents? Perhaps we should provide access to the GamePad instance when deviceClass is 'gamePad'.

What information should be supplied for keyboards?

It seems like we should at least have some mechanism for detecting the type of keyboard (eg. a virtual alphabetic keyboard vs. a physical PC keyboard), possibly as well as its layout and current language settings. But there are a lot of details here that a keyboard input expert will need to flush out.

Should we provide some notion of object identity?
Should we try to say anything concrete about what it means for two InputDeviceCapabilities instances to be equal? We need to be careful not to place an unreasonable compatibility burden on the API. Eg. a first implementation may not include 'maxContacts' and so represent all touchscreens in a system with the same InputDeviceCapabilities instance. If per-touchscreen information like maxContacts was added in the future, the change in equality semantics could conceivably break applications.

Do we want device Capabilities**Class at all?**
Defining DeviceCapabilitiesClass precisely may get tricky (Eg. which class should a depth sensor device fall under?). Different applications will make different assumptions about what exactly a device being of a specific class means. Perhaps we should just eliminate DeviceCapabilitiesClass entirely and rely on abstract properties that describe specific capabilities, like pointerGranularity? If we are going to keep it, does using an enum make JS extensibility too difficult? Should we just use a DOMString like PointerEvent.pointerType does?