Enabling Second Display Use Cases on the Web: The Second Screen Presentation Community Group

Dominik Röttsches, Intel Corporation
Anssi Kostiainen, Intel Corporation

Abstract

The W3C Second Screen Presentation Community Group[1] was formed at the end of 2013 as an Intel initiative to develop a specification for accessing nearby secondary displays from a browser. The W3C group brought together interested parties across the web standards community, and current active participants include two major browser vendors, Google and Mozilla, along with over 30 participants across the industry. Its goal is to define an API that allows web applications to use secondary screens to display Web content. Phones, tablets, laptops and other devices support ways to attach additional display screens. Common methods include attaching through video ports or HDMI, or wirelessly through Miracast, WiDi, or AirPlay. Screens can also be attached over a network. For many of these techniques the operating system hides how the screen is attached and provides ways for applications to use the screens. Native apps on an operating system can easily use these additional screens without having to know how they are attached to the device.

The goal of this CG is to define simple APIs to request displaying an HTML page on the secondary screen and some means for the first screen to communicate with and control the second page, wherever it is rendered. That API should hide the details of the underlying connection technologies and use familiar, common web technologies for messaging.

This will enable use cases like showing presentations on a nearby screen, playing multi-screen games, and performing media sharing and continuous media playback/media flinging.

Introduction

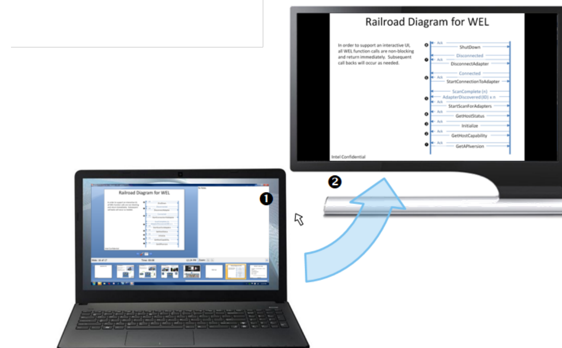
We are motivated by mainly two reasons to enable the web platform to access nearby secondary displays. Primarily we are interested in enabling use cases that allow the user to do new things using the web. Secondly, we would like to close the gap between native and mobile: Allow web applications to do things that traditionally only applications implemented using native APIs were able to do.

[1] "Second Screen Presentation Community Group." 2013. 5 Feb. 2014 <http://www.w3.org/community/webscreens/>

Use Cases

Presentations

A user is preparing a set of slides for a talk. Using a web based service, she is editing her slides and speaker notes on the primary screen, while the secondary larger screen shows a preview of the current slide. When the slides are done, her mobile phone allows her to access them from an online service while on the go. Coming to the conference, using wireless display technology, she would like to present her slides on the stage screen from her mobile phone. The phone's touch screen helps her to navigate slides and presents a slide preview, while the projector shows her slides to the audience.



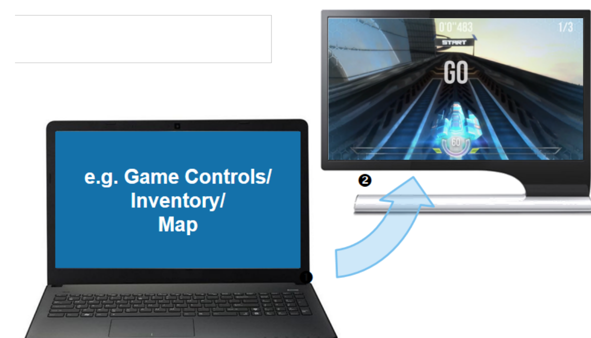
Video and Image Sharing

Using an online video or image sharing service, a user would like to show memorable moments to her friends. Using a device with a small screen, it is impossible to show the content to a large group of people. Connecting an external TV screen or projector to her device - with a cable or wirelessly - the online sharing service now makes use of the connected display, allowing a wider audience to enjoy the content.

The web page shows UI elements that allow the user to trigger displaying content on the secondary display (e.g a "send to second screen") only if there is at least one secondary screen available.

Gaming

Splitting the gaming experience into a near screen controller and a large screen visual experience, new gaming experiences can be created. Accessing the local display on the small screen device and an external larger display allows for richer web-based gaming experiences.



Media Flinging to Multiple Screens

Alice enters a video sharing site using a browser on her tablet. The browser asks Alice which screens she would like to give the site access to, and whether she would like the browser to remember this setting. Alice selects all the screens she has around her at home from the user interface provided by the browser. Next, Alice picks her favorite video from the site, and the video starts to play on her tablet. While the video is

playing, the site asks Alice whether she would like to use another screen for playback, and provides a user interface that lists all the screens Alice has allowed the site access to. The screens are identified by names that are familiar to Alice. Alice picks the largest screen she has at home named "Alice's big TV", and the video playback continues seamlessly on the selected screen. Later on, when Alice moves to another room in the house, she picks her tablet again, and the site automatically continues the playback of the video on the tablet.

Availability of Multi Screen APIs on Desktop & Mobile Operating Systems

We see that the general topic has been addressed in various ways on the native, operating system level API side.

- On Windows 8 and Windows 8.1 the operating incorporates Miracast[2] as a wireless display technology, giving native applications to nearby, wirelessly connected displays.
- On Android, API's like Presentation API and Media Router API[3] provide access to nearby displays, connected via cable or wirelessly over Miracast.
- On Apple iOS and OS X, multi monitor support is provided via the UIScreen[4] and Quartz Display Services API[5]. Both support connecting to nearby displays using a cable or wirelessly using Apple Airplay.
- Alternative platforms and products are using related technologies for bringing content to nearby displays through wired or wireless networking, such as DLNA[6].

Operating Modes for Multi Screen APIs

Generally, we can distinguish two operating mode for the available multi screen APIs mentioned above.

Extended Mode

In this mode, the primary device extends the total display surface by showing independent contents on the secondary display.

Clone / Mirror Mode

In this mode, the primary device mirrors the contents of the primary display 1:1 to the secondary display.

[2] "Wi-Fi CERTIFIED Miracast™ | Wi-Fi Alliance." 2012. 5 Feb. 2014 <http://www.wi-fi.org/wi-fi-certified-miracast%E2%84%A2>
[3] "Presentation | Android Developers." 2012. 5 Feb. 2014 <http://developer.android.com/reference/android/app/Presentation.html>
[4] "UIScreen Class Reference - Apple Developer." 2010. 5 Feb. 2014 <http://developer.apple.com/library/ios/documentation/uikit/reference/UIScreen_Class/Reference/UIScreen.html>
[5] "Overview of Quartz Display Services - Apple Developer." 2012. 5 Feb. 2014 <https://developer.apple.com/library/mac/documentation/graphicsimaging/Conceptual/QuartzDisplayServicesConceptual/Articles/Overview.html>
[6] "Technical Overview - DLNA." 2013. 5 Feb. 2014 <http://www.dlna.org/dlna-for-industry/technical-overview>

Situation on the Web

Implementing such multi screen use cases as listed above is currently not feasible using existing APIs available in browsers. We believe the same holds for APIs that are only in standards tracks and have not made it to browsers yet.

Practically, it is not possible to open secondary windows in browsers these days due to prior abuse of the `window.open()` function, leading to popup-blocking mechanisms in all major browsers.
The Fullscreen API[7] is available but does not have a notion of controlling which screen the fullscreen content should be shown on.
The Network Service Discovery API[8] covers technologies like Zeroconf, DIAL and DLNA, but does not address the case of local displays and has generally a different scope than what we try to enable.
There are proprietary solutions for connecting HTML5 Video elements to different target displays, like Microsoft's PlayTo approach and Apple's proprietary extensions for displaying video elements or AirPlay devices.

Standardization Gap

Comparing the native situation and the situation on the web, both as described above, we clearly see a gap in standardization. Defining an API that addresses the community group's use cases would bridge this gap and allow several modern multi-screen use cases. It is the community group's goal to solve this situation.

Draft API & Discussion

Intel has contributed one initial draft API draft to the community group, which is called Presentation API and available from here: http://webscreens.github.io/presentation-api/ - After discussion on the mailing list, the Media Flinging use case was added.
The community group is now discussing an updated proposal on its mailing list.
http://lists.w3.org/Archives/Public/public-webscreens/2014Jan/0044.html

[7] "Fullscreen." 2011. 5 Feb. 2014 <https://dvcs.w3.org/hg/fullscreen/raw-file/tip/Overview.html>
[8] "Network Service Discovery." 2012. 5 Feb. 2014 <https://dvcs.w3.org/hg/dap/raw-file/tip/discovery-api/Overview.html>

In the following, we provide a walkthrough to the draft that is currently under discussion:

```
// Running on the primary screen.

// Start the search for nearby available or resumable screens.
var discovery =
navigator.presentation.discoverScreens("http://example.com/player.html");

discovery.ondiscovered = function (e) {
  // e.screens is an array of PresentationScreen
  / objects the user has given permission to.
  var selected = e.screens[0];

  function stateChanged() {
    switch (selected.state) {
      case "available":
        // The selected screen has not been
        // showing any content previously, load playerURL now.
        selected.present();
        break;
      case "presenting":
        // The primary and secondary screen are
        // connected with each other and can communicate
        // using MessagePorts, communicate() abstracts
        // away those details in this example.
        communicate(selected);
        break;
      case "disconnected":
        console.log("Screen disconnected.");
        break;
    }
  }

  // A human-readable string intended to identify the screen for the user.
  console.log("Using the screen " + selected.name);

  // Handling the initial state.
  stateChanged();

  // Handling any future state changes.
  selected.onstatechange = stateChanged;
};
```

A playerURL is passed to the discovery entry point of the API in order to discover potentially existing running sessions. The discoverScreens call returns a Discovery object on which you can listen to discoveredevents: One initial event if there is at least one available display, and then continuously whenever the available screens configuration changes. If there is no available screen after calling discoverScreens, the discoveredevent does not fire.

The discovered event type will then expose a property screens which lists available screens, each of which has a name property containing a human readable name describing the screen, for example "Living Room TV".

Rationale: Having an explicit function call discoverScreens() makes it clear when the discovery is to be started. One cannot listen to the discovered event before the discoverScreens() has been explicitly called to avoid programming mistakes.

User's consent: A permission request is brought up after the discoverScreens() method is called to ask the user for permission which screen to give access to. The user can grant access to either one or multiple screens, which are then exposed to the script for enumeration and usage.

For simplicity then we select the first screen and log its name to the console. In a real world example, the user would choose one based on the names of available screens.

The screen can generally be in one of multiple states: "available", "presenting" and "disconnected".

After we have found and selected one screen, we handle its initial state. Here, it would be either "available", in which case we can launch a presentation on that screen by calling present() on it. The URL to be launched on that screen is the same as the one used for the discoverScreens() call.

Alternatively, the screen is already in state "presenting" which means that we have found a session that can be resumed, in which case we can call an imaginary communicate() function which uses the MessagePort of the screen object.

If a screen becomes unavailable by disconnection, a network error or similar, the screen will transition into a state "disconnected" - which we here just log to the console.

After the initial state handling is complete, we listen to changes in screen configuration by registering our stateChanged function as the onstatechange event handler.

Summary

Seeing the clear standardization gap between web and native when it comes to multi screen APIs, the community group’s work will eventually help solving the problem by creating and defining Presentation API: An API for web applications to connect to nearby displays, abstracting from the connection technology. This will enable a variety of new and interesting use cases such as running presentations, gaming and media sharing/flinging.