A Web Framework for Cross-device Gestures Between Personal Devices and Public Displays

Marco Barsotti, Fabio Paternò, Francesca Pulina
CNR-ISTI, HIIS Laboratory
Via Moruzzi 1, 56124 Pisa, Italy
{marco.barsotti, fabio.paterno,
francesca.pulina}@isti.cnr.it

ABSTRACT
In order to exploit the wide availability of public displays and personal devices on the mass market at affordable prices it is important to provide developers with frameworks that ease obtaining cross-device user interfaces able to exploit such device ecosystems. We present the design and implementation of a Web framework for the development of cross-device user interfaces able to take advantage of both personal devices and public displays, and support various types of gestures and their combinations in such multi-device environments. We introduce the design space addressed, describe the framework functionality, its application interface and run-time support, show some example applications, and report on a first test with developers.

Author Keywords
Public Displays; Personal devices; Cross-Device Interaction; Gestures; Framework; Web Applications.

ACM Classification Keywords
H.5.2 [User Interfaces]: Input Devices and Strategies, Interaction Styles

INTRODUCTION
Nowadays, public displays are pervasive in many contexts of use with different purposes, generally to heighten user experience in specific environments, such as museums, hospitals or shopping centres. For example, in museums, visitors can obtain more information about exhibitions and artworks or special events can be promoted in the public display; in hospital centres, it can be useful to connect to public displays in waiting rooms or in departments to check exams, booked appointments or medical information; in shopping centres, customers can use public displays to select products or to access additional information to discuss with others. In general, contents can be abundant, derive from various sources, and be split into several sections in order to improve their consultation. Mobile device technology is evolving fast as well and is not limited only to smartphones. For example, smartwatches are becoming popular as well, and the usage of such wearable devices, either alone or in combination with other personal devices or public displays, leads the way to new models of interaction, which still have not been totally defined [13]. Each device type has specific features that make it more suitable for some task type: for example, smartwatches have distinguished themselves for fast access, glanceability, and in general for being more convenient than other devices [12] in performing some daily activities.

Overall, the usage of smartwatches, as well as other personal devices, in combination with public displays can be exploited to create new interaction possibilities involving more than one device at a time. In this way, the available devices can cooperate and coordinate with each other to encourage interaction with the information presented in public displays. However, despite several research efforts current tools for user interface development still provide poor support for the development of interactive applications accessible through public displays in combination with personal devices. In order to contribute to overcome such limitations, we have analysed the various possible options, and then designed and implemented a corresponding framework that is able to support communication and interaction across personal and public devices, also involving mid-air gesture sensors, such as Microsoft Kinect, when interacting with Web applications. Such novel framework can be used for the development of various domain applications, which can exploit different kinds of multi-device interactions.

In particular, the contribution of this work consists in:

- The design and implementation of a Web framework to facilitate the development of applications supporting cross-device gestures between personal devices and public displays;
- Example applications of such framework in order to show how it can be concretely exploited;
- Test with developers carried out in order to check whether they can learn how to use it with limited effort.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
MUM 2017, November 26–29, 2017, Stuttgart, Germany
© 2017 Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5378-6/17/11…$15.00
https://doi.org/10.1145/3152832.3152858

RELATED WORK

Cross-device interaction and communication in the context of public screens and personal devices has already been considered. Related work shows some example applications. Touch&Screen [2] is a set of interaction techniques for the remote control of widgets (menu, lists, videos, maps etc.) for large screens through smartphones. That study has reported interesting user feedback, but does not provide any framework for developing new multi-device interactive applications, and no other personal device type has been considered. WallSHOP [15] is an application for interactive shopping and is composed of a client-server system in which some users are connected simultaneously to a public screen and can remotely interact with it. Thus, it uses Web-based technology, but the mobile device is only a remote controller for selecting elements in the public display. Also in [10] the only type of interaction with the public display is to control it remotely and change contents on the screen with the user's mobile devices. CurationSpace [3] is an environment for curating and composing digital historical artefacts by using smartwatches to support content modification and development during the creation process. In the context of public transport, some authors [11] have shown an example of remote control of a public display by using mobile devices. This is an example of a remote control application without support for direct interaction with the public display. Other authors [9] have developed some applications through which users can use their smartphones to interact with the touch screen by analysing data from internal sensors, but the work does not address any other type of device.

WatchConnect [8] targets an interaction space in the complex process of integration between a smartwatch and an interactive surface for general purposes. The authors found some important principles and concepts in the context of cross-device interaction with smartwatches and, to deploy such aspects, they built a software toolkit to support an Arduino-based hardware prototype. This application does not define a framework, nor does it aim to exploit interactions with other personal devices. WEAVE [5] is a JavaScript framework to to easily distribute user interface output across mobile and wearable devices. It provides functionalities to select target devices, perform output actions, and handle input events on one device. It has then been improved with a visualization technique to simulate the actual behaviour of the multi-device user interface [6]. Its limitations are that it requires a Weave proxy to be pre-installed and run on each individual device, it is not able to manage mid-air gestures, and it does not provide explicit functionalities to directly create and manage events obtained by combining events from multiple devices. Moreover, the WEAVE framework was not designed to support interaction with public displays, but was limited to addressing mobile and wearable devices. Duet [4] aims to address the mobile interaction in a cross-device context when a smartphone is paired with a smartwatch. Thus, also in this case public displays have not been addressed. The authors start with the study of the spatial configuration of the two devices and then propose a design space with four different typical configurations. For each configuration, the authors have indicated some corresponding applications and example interactions but they do not provide a framework for developing applications able to address the identified design space.

XDBrowser [16] is a tool that supports the implementation of some cross-device features through a browser extension: it supports the possibility to select parts of the user interfaces and distribute them to other devices running the tool, and then keep their state synchronised. Panelrama [20] supports the automatic distribution of user interface parts based on their features and the characteristics of the available devices. A model-based approach to customizing user interface distribution has been proposed in [14]. However, such approaches do not support interaction by means of gestures combined through multiple devices.

A couple of frameworks for cross-device Web applications [7, 18] have addressed the issue of independence from external servers, while in our case we focus on how to allow developers to specify gestures that involve multiple devices. The SoD-Toolkit [19] aims to support interactively prototyping and developing multi-sensor, multi-device applications. In this perspective, it supports a fusion method for multiple Kinect devices, but it does not provide an API for supporting developers in defining specific cross-device gestures.

Our research leverages such contributions, for example it considers the design space addressed in the Duet approach and examines how to consider public displays as well. In particular, we aim to investigate how these concepts and approaches can be extended in order to better support development of applications with cross-device interactions between public displays and personal devices, including mid-air gestures as well. For this purpose, we have designed and implemented a Web framework, which can be exploited to easily design and develop various types of cross-device gestures, including flexible combinations of gestures from different devices.

INTERACTION DESIGN SPACE

In a cross-device context [17] in which the public display is an important input/output system, it is crucial to define the roles that other devices can have. Generally, a key feature of a public display is its large touch surface, which is an important resource for the user, who can perform direct interactions. Mid-air gestures can be exploited too, allowing users to maintain a certain distance from the display to have a better point of view of the information presented on the screen, while still allowing interaction with it.

The set of interactive techniques can be extended to obtain wider and more interesting interactive scenarios. Starting with common and simple interaction techniques that can be performed on a single device (single-tap, double-tap, swipe

etc.), also using mid-air gestures when interacting with a public display, we can identify and design gestures that involve multiple devices at the same time.

The interactive design space becomes more complex when considering interaction techniques with the public display able to involve personal devices as well. We have identified some cross-device interaction techniques (see examples in Figure 1) that can be classified into two main categories:

- Cross-device gestures;
- Cross-device input and output.

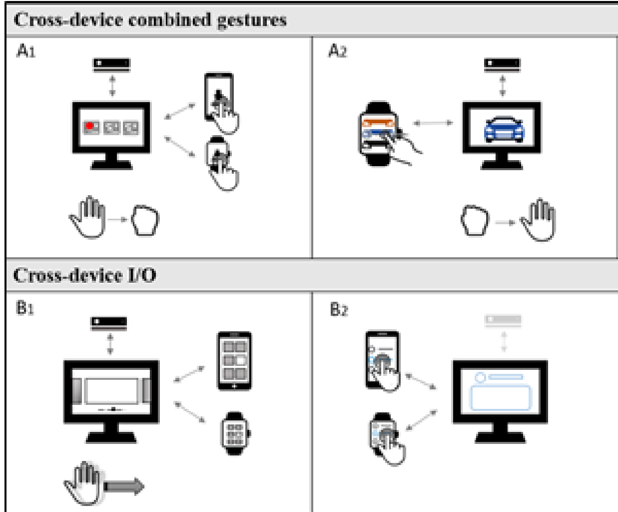

Figure 1. Examples of cross-device techniques supported by the framework: (A1) Closedin-Tap, (A2) Swipe-Closedout, (B1) Overview+Detail, (B2) Auxiliary Tool.

The first group (cross-device gestures) refers to gestures that are performed in multiple devices within some temporal constraint in order to obtain a combined gesture. Thus, also a simple tap or a swipe gesture on the personal device can be a part of a more complex gesture that involves the public display, even when using mid-air gestures. Some examples of cross-device gestures are:

- *Swipe-Hand open*: the user performs a swipe on a personal device to select some information, then this content is presented on the public display by performing a hand-open mid-air gesture in front of its large screen;
- *Press-press*: the user performs a double press, first on the smartwatch and then on the public display or vice versa; for example, this technique is useful to synchronize some information between devices;
- *Hand close-tap*: the user performs a mid-air gesture closing her hand in front on a public display, then taps her personal device; for example, the user can use this technique to download some information from the public display to the personal device.

By the second group (Cross-device input and output), we refer to gestures that are performed on one device but their effects change the user interface of another device as well. Therefore, the public display and the information that it shows are only a part of the entire interface and the user can use the personal device to take advantage of some additional tools. Examples of this group are the following techniques:

- *Auxiliary Display with Task-based Tools*: the personal device is an auxiliary display and its interface can contain tools, links or other useful information to perform a specific task or to better navigate the public display content;
- *Overview+Detail*: the public display provides an additional viewpoint for the user and it can show some device content in detail while the personal device can be used to have an overview of the information space, thus providing a spatial separation between focused and contextual views (Figure 1-B1).

THE CDI FRAMEWORK FOR CROSS-DEVICE INTERACTIONS

To allow designers and developers to more easily address such design space, we have designed and implemented a framework that allows multiple devices to interact through both cross-device gestures and cross-device input and output. Our framework has been designed to address the development of Web applications, and it allows developers both to choose which cross-device interactions to manage and to decide what Web page element(s) the gesture applies to. In this way, developers can abstract out of the devices' features and communication, thereby enabling simplified and faster programming, and rapid prototyping of cross-device user interfaces.

The Development Framework

To this end, we have designed and developed the JavaScript framework "CDI" (Cross Device Interaction). It has been implemented as a JavaScript library with a server for run-time support.

As Figure 2 shows, the framework uses communication channels based on WebSockets specifications in order to connect the application parts running on multiple devices.

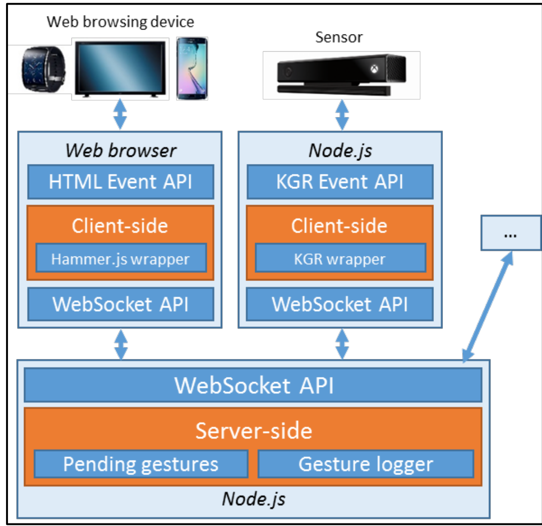

Figure 2. The CDI Framework Architecture.

In order to detect the basic “native” gestures (i.e. gesture performed on a single device) our framework exploits wrappers, which can be used to easily extend its possibilities. Thus, for example, if there is a need to integrate events from a new sensor connected to a specific device, then it will be possible to add a few code lines to access the wrapper of that device in order to detect those events without modifying the rest of the library. The use of the Node.js server is also useful to detect events that occur on devices not directly running Web browsers, such as a Kinect device.

Our framework exploits the functionalities offered by the Hammer.js [1]library for detecting the basic touch events such as single-tap, double-tap, swipe, pinches etc. In addition, in order to support mid-air gestures we have also developed a wrapper and a library KGR (Kinect Gesture Recognizer), which is able to recognize gestures detected through a Microsoft Kinect 2 in Node.js environment.

Conversely, complex Cross-device gestures detection is performed through the CDI server: in fact, as soon as a device detects a native gesture, it notifies the server. If other compatible gestures are also detected on other devices within a certain time, the server combines the two detected gestures as a single complex gesture.

Developers can design their applications by writing an event-based code, in which a callback function is called when the cross-device event occurs. In this JavaScript function, developers define the actual behaviour of their applications, which can be updating the interface or transferring a piece of information between the devices.

[1] http://hammerjs.github.io/

Framework API

Developers can use the framework by writing the application code for all the application parts in the various devices. When they create new instances of CDI objects two parameters should be specified: an ID, used for identifying the device, and the address of the CDI server, used for the WebSocket connection. Optionally, it is possible to indicate the wrappers to use for recognizing the native gestures. Table 1 shows an example of JavaScript code necessary to create an instance of the CDI client-side library in two devices. In this case, three scripts are included: Hammer.js, the corresponding wrapper, and the CDI library. Then, it is necessary to instantiate the library by calling its constructor. In this case, the device model has been used as the ID for the connection. The CDI constructor has also received an instance of the Hammer.js wrapper as a parameter.

```
// HTML in the first device
<html>
  <head>
    <script src="./js/hammer.min.js"></script>
    <script src="./js/cdi-wrapper-hammer.min.js"></script>
    <script src="./js cdi-client.js"></script>
    <script>
      var cdi = new CDI('Samsung Galaxy S5', 'ws://192.168.2.10:50500', {
        wrappers: [new CDIWrapperHammer()]
      });
    </script>
    ...
```

```
// HTML in the second device
<html>
  <head>
    <script src="./js/hammer.min.js"></script>
    <script src="./js/cdi-wrapper-hammer.min.js"></script>
    <script src="./js cdi-client.js"></script>
    <script>
      var cdi = new CDI('Apple iPhone 5S', 'ws://192.168.2.10:50500', {
        wrappers: [new CDIWrapperHammer()]
      });
    </script>
    ...
```

Table 1. Example of CDI instantiation on two devices.

After having instantiated the clients, it is possible to bind gestures on the devices and define cross-device interactions.

The framework manages two types of events: single and combined. A single event has five properties: name, the target element on which the event is recognised, recognizer (which is the corresponding wrapper), the optional associated data, and a serial number indicating how many times that event has occurred (this can be useful also when combining events from multiple devices in order to facilitate matching events generated by different devices).

A combined event is defined by a name and an array of segments, which are either single events already recognised or elements still undefined. For such single events already recognised both the device where they occurred and an optional attachment (which is the data transmitted by the *onSend* callback) are indicated.

In terms of methods there are *on* and *off* that activate or deactivate the recognition of an event from one device. The *sendInput* method allows a device to send an input or some data from one device to another. The receiving device can enable or disable receiving data from one specific device through the *onInput* and *offInput* methods.

The *onCombined* method is particularly important because it allows developers to endow the application with cross-device gesture management; it has four parameters:

- the target element, is the identifier of an element which has a wrapper able to handle the associated events, for example it can be the ID of a DOM element,
- a string indicating the sequence of gestures to combine,
- a number for selecting an element in the sequence of gestures (0 for the first, 1 for the second, …), which is used to identify the one to be recognised in the current device, and that should occur on the element indicated as the first parameter,
- the callback associated with the cross-device gesture. Actually, this fourth parameter may be not only a single, but also multiple optional callbacks (which can be either generic functions or one of the possible types indicated in Table 2).

Callback	Usage
onComplete	Activated when the cross-device gesture has been recognized.
onSend	Activated when a device sends to the server a gesture for possible composition. It associates some information to the gesture that can be exchanged across devices.
onTimeout	Activated when the cross-device gesture fails because it has not been completed within the given time.

Table 2. The possible specific callbacks associated with the *onCombined* method.

For example, consider the case in which the developer wants to implement a *swipe-tap* combined gesture, which will be used to send an image from one device to another (for example from a smartphone to a public display). To achieve this behaviour, the *onCombined* method with *onSend* and *onTimeout* callbacks should be used. *onTimeout* is used because if the gesture composition fails then the user should be notified. *onSend* is used to return the image selected by the swipe. Then, the indicated image (which is in the attachment of the first segment associated with the combined event) is inserted in the DOM of the application in the target device in the position indicated when the tap event occurs. Table 3 shows the code lines necessary to recognize this example of cross-device gesture.

As can be seen, it is possible to obtain the desired behaviour in a few lines thanks to the expressiveness of the callbacks accessible from the *onCombined* method.

```
// JavaScript in the first device, where the swipe on the image with ID "img" occurs

cdi.onCombined('img', 'swipe tap', 0, {
   onSend: function() {
      // the image url is attached to the swipe gesture
      return document.getElementById('img').src;
   },
   onTimeout: function() {
      alert('You have to tap on the second device');
   }
});
```

```
// JavaScript on the second device, where the tap on the container with ID "area" triggers the receipt of the image sent from the first device

cdi.onCombined('area', 'swipe tap', 1, function(e) {
   // the image url is obtained from the first segment (swipe)
   // of the cross-device gesture
   document.getElementById('area').src = e.segments[0].attachment;
});
```

Table 3. Example of use of the framework on two devices.

Figure 3. The CDI Framework Architecture.

CDI FRAMEWORK APPLICATION

In order to indicate the potentialities of the framework, especially using personal devices such as smartphones and smartwatches in combination with public displays, we present some example applications that have been implemented with it.

The first one, *Tour*, is an application supporting tourists during a city tour. It shows some interesting places to visit through an image gallery displayed in the public display. The user can slide images performing *mid-air* gestures, and read short descriptions about the corresponding place, as well as route indications to reach it, in her personal device. The second one, *Dress Shop*, is an example application for a clothing store equipped with a public display, in which the products for sale in that shop are showed: the user can perform mid-air gestures to display details of a specific product on her smartphone, and optionally add it in the whishlist (performing a *closedin-tap* combined gesture, like the one showed in Figure 1-A1).

We carried out a first user test on these two applications in order to gather some initial user feedback regarding the cross-device gestures that can be implemented with the framework. The obtained feedback has been taken into account in the development of a third application, a *Car configurator* application. Thus, in this one we have introduced the use of tooltips on the public display and vibrations on personal devices, useful to inform the user about the availability and the completion of the gestures.

This complex application allows users to personalize a car model by selecting specific accessories, and receive associated price quotes. For the smartwatch we have tested it with a Samsung Gear S, running the Web-based Tizen OS.

Below we indicate the main tasks of the "Car configurator" application, and the interactions to accomplish them (a video showing such interactions is available at https://youtu.be/AcJ91pjz00I):

- *Rotate the car*: holding their arm outstretched toward the screen, users can move their hand to the left or right (*panleft* or *panright* gestures) to rotate the car in the direction of the hand. The larger the movement the greater the rotation.
- *Show an accessory*: with the arm bent, users have to close their hand into a fist holding the index out (*lasso* gesture). The public display shows a cursor that can be placed on various parts of the car by moving their hand. When the cursor is located on an editable part, the personal device vibrates and displays the chosen accessory. This task uses public displays and personal device for an "overview + detail" technique.
- *Select an accessory*: this task requires also closing the index finger (*lassoclosed* gesture), which triggers the personal device to display the options available for the part currently selected. Users can scroll by swiping their finger across the personal device. This task uses public displays and personal device according to the "auxiliary display with task-based tools" technique.
- *Install an accessory*: during the task "select an accessory", users must open their hand towards the public display (*closedout* gesture) on the car to install the option chosen on the personal device.
- *Save a quote*: it is necessary to close the hand towards the public display (*closedin* gesture), and then tap a corresponding button on the personal device. The car quote shown on the public display is also added to a searchable list displayed on the personal device.
- *Open a quote*: users can perform the reversed cross-device gesture of "save a quote". A quote can be selected by tapping on a list item on the personal device, then opening their hands to the public display to show the quote on the larger device (Figure 4, top part).

While the "rotate the car" task only uses the Kinect sensor as an input device for the public display, the other tasks require a composed interaction between multiple devices. Despite this, the application development required writing no more than 200 lines for each device. This number is low, given that most of the code has the purpose of improving the graphical interface through the use of jQuery and Google MDL template in order to make the Web application graphics similar to those of native apps.

In this application, it is possible to use two types of personal devices (smartwatches and smartphones) at different times. For example, the user can change the accessory via the smartwatch (see Figure 4), and then finish saving the quotes in the smartphone. This type of interaction technique can be interesting because it exploits the glanceability of smartwatches, used both as an external display and toolbox near the user's hand used in the mid-air gestures. This will be further investigated in future user tests.

Figure 4. The "Car Configurator"CDI application.

A TEST WITH DEVELOPERS

To evaluate the framework's usability, we have carried out a study with 10 developers. Our study's goal was to verify from the developer's point of view if the CDI library is easy to learn and use, and if it actually does facilitate the development of cross-device interactions in Web applications. We also collected their feedback through a post-study questionnaire.

Participants and Setup

Ten users (3 females), aged between 26 and 48 years (mean 33), took part in the study. A good knowledge in programming with JavaScript was required for participation. We recruited them from researchers working in our Institute who had not used CDI before, and from students who had subscribed to a mailing list and answered our invitation. They all have an educational qualification in informatics or digital humanities: high-school (1), bachelor's degree (2), master's degree (5), Ph.D. (2).

We asked them to evaluate, on a 5-point scale (1 min – 5 max), their level of experience in programming with JS/jQuery, in Web programming (server-side and other programming languages), and in creating applications with UIs that adapt to devices with responsive design rules. The results indicate that, except for one (who rated himself 2 for all the three aspects), participants had a generally good background in programming (on average rating themselves 4). Moreover, two of them had already realized applications with distributed user interfaces, and 6 out of the 10 know other JS tools such as jQueryUI, Hammer.js, Node.js, Angular, D3. None of them has ever programmed with

Kinect SDK; just 2 have used applications involving the Kinect sensor.

A couple of days before the experiment, participants received a summary documentation explaining the framework's architecture, the types of cross-device interactions that the CDI library allows developers to define, the supported gestures, all API's methods provided with related examples. They received also a video showing the execution of a cross-device Web app created with CDI.

First, we asked participants to compile a pre-study questionnaire, containing questions about demographic and professional background information, whose results are referred above.

After that, they were asked to complete three programming tasks using the CDI library, in order to implement cross-device interactions – with both single and combined gestures – on a Web application created for this study. This application allows users to colour, with mid-air gestures, a white grid shown on the public display by picking colours from the palette available in the wearable device. It involves a public display, two personal devices (smartphone and smartwatch) and the Kinect sensor. First, both grid and palette are on the smartphone; then the former element can be transferred to the public display, and the latter to the smartwatch. In this way, the white grid, now visible in the public display, becomes ready to be coloured. Tasks are consequential and, one by one, lead to the application complete with all functionalities. Moreover, they have been conceived in order to encompass the various possible ways of interaction that CDI allows: "mixed" (touch + mid-air) combined gestures; only touch combined gestures; single mid-air gestures with hand position detection; sending and receiving of inputs with attachments to be sent from a device to another.

Participants worked with three screens. In the first one, they found the editor in which they could write their code, and the command shells showing devices running on the CDI server. In the second one, they found the two mobile emulators (smartphone and smartwatch) with the correspondent app's pages opened. The third one, with the Kinect sensor placed above, showed the public display view.

Once they completed all tasks, we asked them to complete a post-study questionnaire, in order to get feedback about their programming experience with CDI, and additional positive and negative comments concerning CDI features.

Tasks

Participants worked in the presence of one researcher who at the beginning clarified doubts regarding the summary documentation or the instructions given. They could also look up the summary documentation at any moment during the test, and decide whether or not to execute their scripts. We asked them to complete three tasks. Task 1 and Task 2 are divided into 2 subtasks.

Task 1 – distribute the smartphone's interface elements with cross-device combined gestures:

> Subtask 1 – transfer the uncoloured grid to the public display with a *swipe closedout* combined gesture;
>
> Subtask 2 – transfer the colour palette to the smartwatch with a *swipe tap* combined gesture.

Task 2 – identify a rectangular area inside the grid now visible on the public display:

> Subtask 1 – select a cell with a *lasso* mid-air gesture using the left hand, as the first extremity of the rectangular area;
>
> Subtask 2 – select another cell with a *lasso* mid-air gesture using the right hand, as second extremity of the rectangular area.

Task 3 – colour the rectangular area with a combined *lassoclosed tap* gesture: the tap is on the smartwatch element containing the colour we want the area to be coloured with.

Results

Participants worked without a maximum time limit. We measured the completion time for each task (and subtask) without stopping the timer when they looked up the documentation or tested their scripts. Therefore, the collected time values also include the time dedicated to these two activities. The box-plot in Figure 5 presents the completion time for each task and subtask.

Subtask 1 of Task 1 is the one that took most time to be completed followed by Task 3. Not all participants read the summary documentation we sent to them, so they spent most of the time first looking up what the necessary methods were, and then understanding their syntax. Even participants who read the documentation before the test needed some time to get familiar with those methods, and, once completed the first task, they went on with less hesitation and therefore more quickly. This aspect partially explains the long completion time recorded for the ST1 of Task 1. Furthermore, we should consider that ST1 of Task

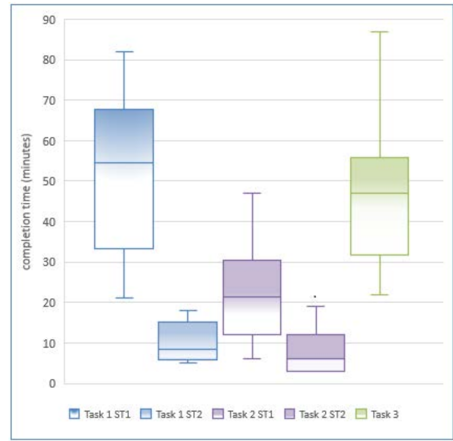

Figure 5. Task completion time

1, as well as Task 3, requires the implementation of an interaction that includes, besides the combined gesture, an input exchange between devices: a more complex interaction that probably justifies the greater amount of time needed by participants to complete those two tasks.

Subtasks 2 is very similar to Subtasks 1 as regards the interaction logic. Taking into account this analogy and considering that participants had become more familiar with the methods, we can suppose that these reasons explain why Subtasks 2 took much less time to be completed.

Task 3 requires a bit more effort than ST1 of Task 1: as ST1 of Task 1, it requires the implementation of a mixed combined gesture, but the input to the public display has to be enhanced with a piece of information (the background colour value) provided through another device. The Task 3 average completion time is higher than Task 2, but still lower than ST1 of Task 1. Even though at that point of the study participants were more efficient in using CDI methods, they encountered some difficulties in identifying the coordinates of the target elements to be coloured. However, this aspect is unrelated to the CDI library specifically, so we did not consider failure to identify the area an error, when it occurred. We only verified that the transfer of the colour to the public display was implemented correctly.

Not all participants tested their scripts during the study: 5 tested all scripts, 3 tested none, 2 instead tested only for some tasks. Participants who read the summary documentation a couple of days before the test were more inclined to test their scripts. Those who tested without having read the documentation before registered a high global completion time (170-180 minutes). Global completion times are lower for participants who read the documentation before the test and also tested all scripts (70-120 minutes), and for those who tested not all scripts (135 minutes approximately).

Errors

To evaluate the correct use of the CDI library, we have taken into account the number of methods required to perform the logic of interactions required by our tasks. After assigning to each method written by the developers a value between 0 and 1 depending on its degree of correctness, we have expressed the global level of correctness in % values. We have considered "wrong" a use with correctness value less than 50%. Almost all participants had used the relevant methods to implement the cross-device interactions, and used them correctly, without making mistakes in sending and receiving inputs between devices. The lowest overall level of correctness was detected for Task 3 (84.1%), followed by Task 1 (92.8%) and Task 2 (94.8%), as shown in Figure 6. Most of the errors found in the participants' code concern the methods parameters. For example, someone failed in identifying the right action target to be recognized by the *onCombined* method (e.g. writing the same target ID in the versions for both devices for ST2 of Task 1). Someone else made a mistake in the *sendInput* method, writing the wrong device name as first parameter (which is the string used to identify it when the CDI framework is instantiated), e.g. because they did not understand they had to specify the name of the sending device in ST1 of Task 1.

In addition, we observed that there is a strong positive correlation ($r = 0.88$) between the level of global accuracy and the level of programming knowledge and experience stated by participants in the pre-study questionnaire (there were more errors and uncertainties in participants who evaluated themselves less experienced than others).

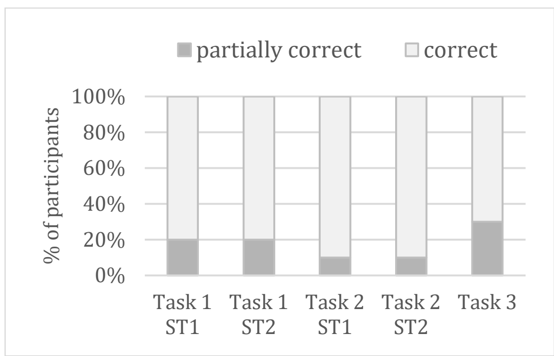

Figure 6. Accuracy levels for the various tasks

Developers' Feedback

Overall, the developers' feedback concerning the CDI framework has been positive (see Figure 7). In a 1 (min) to 5 (max) scale, they found fairly easy to understand in general its functioning (median 4; mode: 4). At the same time, they did not encounter excessive difficulty in understanding how methods worked (median 4; mode: 4). The referencing to elements and devices involved in combined gestures has been considered quite intuitive and easy to handle (median 4; mode: 4).

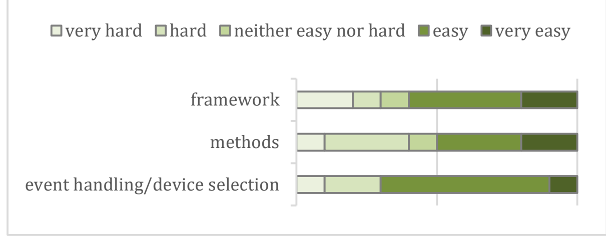

Figure 7. Evaluation of the CDI features

As for the CDI features, we asked participants to evaluate on a 5-point scale the level of difficulty encountered in task completion, see Figure 8. In case of low ratings (1 and 2), we asked them to explain their difficulties.

They considered Task 3 as the one that required more effort to complete. Task 3 is indeed the one where we have detected more errors in the use of methods. Difficulties in Task 3 completion are related to how to transfer the

selected colour from one device to another ("*I had some troubles in transferring the cell's colour from smartwatch to public display*", "*just the ultimate part for colouring the cell*"). Two participants also suggested that it would be better to find an already implemented function for helping them to detect the two extremes of the rectangular area However, this was an issue related to general Web programming and not to the specific CDI framework.

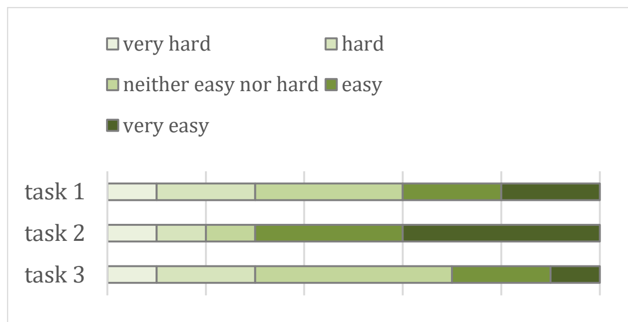

Figure 8. Difficulties in performing tasks assessment

Problems regarding Task 1 completion (median 3; mode: 3) are mostly related – as we expected – to the ST1, as it was more complex (the interaction involves also the Kinect sensor) and it was the first task, thus it required participants greater effort to understand how to use methods to achieve the interaction logic ("*I found more difficulties with subtask 1 than with subtask 2 because I still had to get familiar with the functioning of methods*"). Task 2 was generally considered simpler (median 4; mode: 5) and no particular comment was given for difficulties encountered.

Qualitative developer evaluation

We asked participants to provide their feedback about aspects they most liked as well as the ones they did not like. Some comments containing a negative feedback were about the Kinect part (for example, they would prefer an automatic way to execute again the corresponding .js file on Node.js after making any change) and the framework functioning: "*with regard to syntax and operating principles it's difficult to understand without a minimal training phase*", "*understanding well how to send and recover attachments to messages*". Among positive comments, instead, participants stated methods themselves and their ease of use (e.g. "*I find quite simple the general functioning of methods, once the initial difficulty is over*"). Moreover, they appreciate salient library's features such as "*the possibility to recall events on multiple devices*", "*the possibility to combine different gestures*", "*methods for communication between devices, the logic for exchanging messages between the Kinect and the other devices*", "*the immediacy of transferring events from one device to another*". Finally, "*the possibility of distributing interfaces, transferring them with gestures on various devices is very interesting*".

We asked participants if they would use CDI in the future to develop cross-device applications with interaction techniques such as those experienced during the test. Since they found the libraries easy to use, and were interested in the possible techniques enabled by CDI, they said they would use it in the future to develop applications involving both a public display and personal devices in domains such as museums, games, smart-cities, or for diagnostic imaging in the medical field, and to access content such as encyclopedias and newspapers. .

Someone provided also additional suggestions for improving the CDI framework, such as integrating vocal interaction, and introducing the possibility to customize the timeout duration according to the specific combined interactions.

CONCLUSION

Our life is becoming a multi-device experience and we need support to facilitate the development of cross-device user interfaces able to exploit such technological richness.

We have analysed the interaction design space considering public displays and their integration with personal devices. In this context, we have identified a set of relevant interaction techniques and gestures that involve personal devices in conjunction with public displays. We have presented a framework, together with its Web-based architecture, which enables developers to implement user interactions with limited effort in such a way to exploit the large surface of public displays together with personal devices through various types of gesture-based cross-device interactions. Then, we have described some example applications, and reported on a first test with developers. The framework has been designed in such a way to be easily extensible with new gesture types, and provides an expressive API for supporting such features.

Future work will be dedicated to further validating the usefulness and the usability of the framework with end users and Web developers.

REFERENCES

1. Matthias Baldauf, Florence Adegeye, Florian Alt and Johannes Harms. 2016. Your Browser is the Controller - Advanced Web-Based Smartphone Remote Controls for Public Screens. In *The 5th International Symposium on Pervasive Displays* (PerDis '16), pages 175-181. https://doi.org/10.1145/2914920.2915026
2. Alessio Bellino, Federico Cabitza, Giorgio De Michelis and Flavio De Paoli. 2016. Touch&Screen: Widget Collection for Large Screens Controlled through Smartphones. In *Proceedings of the 15th International Conference on Mobile and Ubiquitous Multimedia* (MUM '16), pages 25-35. https://doi.org/10.1145/3012709.3012736
3. Frederik Brudy, Steven Houben, Nicolai Marquardt, and Yvonne Rogers. 2016. CurationSpace: Cross-Device Content Curation Using Instrumental Interaction. In *Proceedings of the 2016 ACM on Interactive Surfaces and Spaces* (ISS '16), pages 159-168. https://doi.org/10.1145/2992154.2992175

4. Xiang ‘Anthony’ Chen, Tovi Grossman, Daniel J. Wigdor and George Fitzmaurice. 2014. Duet: exploring joint interactions on a smart phone and a smart watch. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI ’14), pages 159-168. https://doi.org/10.1145/2556288.2556955
5. Pey-Yu (Peggy) Chi and Yang Li. 2015. Weave: Scripting Cross-Device Wearable Interaction. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI ’15), pages 3923-3932. https://doi.org/10.1145/2702123.2702451
6. Pey-Yu (Peggy) Chi, Yang Li and Björn Hartmann. 2016. Enhancing Cross-Device Interaction Scripting with Interactive Illustrations. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 5482-5493. https://doi.org/10.1145/2858036.2858382
7. Luca Frosini and Fabio Paternò. 2014. User interface distribution in multi-device and multi-user environments with dynamically migrating engines. In *Proceedings of the 2014 ACM SIGCHI symposium on Engineering interactive computing systems* (EICS ’14), pages 55-64. https://doi.org/10.1145/2607023.2607032
8. Steven Houben and Nicolai Marquardt. 2015. WatchConnect: A Toolkit for Prototyping Smartwatch-Centric Cross-Device Applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI ’15), pages 1247-1256. https://doi.org/10.1145/2702123.2702215
9. William Hutama, Peng Song, Chi-Wing Fu and Wooi Boon Goh. 2011. Distinguishing multiple smart-phone interactions on a multi-touch wall display using tilt correlation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI ’11), pages 3315-3318. https://doi.org/10.1145/1978942.1979433
10. K.P. Ludwig John and Thomas Rist. 2012. xioScreen: Experiences Gained from Building a Series of Prototypes of Interactive Public Displays. In *Ubiquitous Display Environments*. Springer Berlin Heidelberg, 125-142. https://doi.org/10.1007/978-3-642-27663-7_8
11. Romina Kühn, Diana Lemme and Thomas Schlegel. 2013. An interaction concept for public displays and mobile devices in public transport. In *Human-Computer Interaction. Interaction Modalities and Techniques*. Springer Berlin Heidelberg, pages 698-705. https://doi.org/10.1007/978-3-642-39330-3_75
12. Kent Lyons. 2015. "What can a dumb watch teach a smartwatch?: informing the design of smartwatches." In *Proceedings of the 2015 ACM International Symposium on Wearable Computers* (ISWC ’15), pages 3-10. https://doi.org/10.1145/2802083.2802084
13. Kent Lyons. 2016. Smartwatch Innovation: Exploring a Watch-First Model. In *IEEE Pervasive Computing*, 2016, 15.1, pages 10-13. https://doi.org/10.1109/MPRV.2016.21
14. Marco Manca and Fabio Paternò. 2016. Customizable dynamic user interface distribution, in *Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (EICS ’16), pages 27-37. https://doi.org/10.1145/2933242.2933259
15. Soh Masuko, Masafumi Muta, Keiji Shinzato and Adiyan Mujibiya. 2015. WallSHOP: Multiuser Interaction with Public Digital Signage using Mobile Devices for Personalized Shopping. In *Proceedings of the 20th International Conference on Intelligent User Interfaces Companion* (IUI ’15), pages 37-40. https://doi.org/10.1145/2732158.2732179
16. Michael Nebeling and Anind K. Dey. 2016. XDBrowser: User-defined cross-device web page designs. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 5494–5505. https://doi.org/10.1145/2858036.2858048
17. Fabio Paternò, Carmen Santoro. 2012 A logical framework for multi-device user interfaces. In *Proceedings of the 2012 ACM SIGCHI symposium on Engineering interactive computing systems* (EICS ’12), pages 45-50. https://doi.org/10.1145/2305484.2305494
18. Mario Schreiner, Roman Rädle, Hans-Christian Jetter and Harald Reiterer. 2015. Connichiwa: a framework for cross-device web applications. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, pages 2163-2168. https://doi.org/10.1145/2702613.2732909
19. Teddy Seyed, Alaa Azazi, Edwin Chan, Yuxi Wang and Frank Maurer. 2015. SoD-Toolkit: A Toolkit for Interactively Prototyping and Developing Multi-Sensor, Multi-Device Environments. In *Proceedings of the ACM Conference on Interactive Tabletops and Surfaces* (ITS ’15), pages 171-180. https://doi.org/10.1145/2817721.2817750
20. Jishuo Yang and Daniel Wigdor. 2014. Panelrama: enabling easy specification of cross-device web applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2783-2792. https://doi.org/10.1145/2556288.2557199