Panelrama: Enabling Easy Specification of Cross-Device Web Applications

Jishuo Yang, Daniel Wigdor
University of Toronto
{jishuoyang, daniel}@dgp.toronto.edu

ABSTRACT
We present Panelrama, a web-based framework for the construction of applications using distributed user interfaces (DUIs). Our implementation provides developers with low migration costs through built-in mechanisms for the synchronization of a UI state, requiring minimal changes to existing languages. Additionally, we describe a solution to categorize device characteristics and dynamically change UI allocation to best-fit devices. We illustrate the use of Panelrama through three sample applications which demonstrate its support for known interaction methods, we also present the results of a developer study, which validates our belief that cross-device application experiences can be easily implemented using our framework.

Author Keywords
Distributed user interfaces, multi-device environments.

ACM Classification Keywords
H.5.2. [Information Interfaces and Presentation]: User Interfaces - Graphical user interfaces

INTRODUCTION
With the emergence of new computing form factors such as smartphones and tablets, it is increasingly likely that, at any given moment, a user has two or more computing devices available to them. Currently, the majority of applications on these devices operate independently, with isolated application state and I/O devices. The possibility of sharing state information and I/O resources between devices opens up a wealth of interaction possibilities: a user may retain workflow between devices, annex additional hardware to overcome device limitations, or use additional devices to enrich interactions (such as [25]). For example, Figure 1 shows a video streaming application that distributes portions of its interface to devices which are ideal for the role: the video stream may be distributed to a TV, video controls to a smartphone for use as a remote, and related videos list to a tablet for easy browsing and selection of the next video.

CHI 2014, April 26 - May 01 2014, Toronto, ON, Canada

ACM 978-1-4503-2473-1/14/04…$15.00.
http://dx.doi.org/10.1145/2556288.2557195

Existing works make use of distributed user interfaces (DUIs) to allow applications to take advantage of multiple devices. DUIs enable developers to distribute portions of an application's interface to a number of devices, users, displays, and platforms [7, 24]. Previous work has found that a DUI framework should support a number of relationships between its component UI elements, including: 1:1, 1:few, few:few, and many:many [24].

Many existing DUI solutions utilize model-based or schema-based approaches, which have the advantage of being independent of language and platform [6, 24]. However, software implementations of these solutions commonly rely on platform-specific native code [1] or require per-element markup tags to categorize UI division [23, 24]. This makes the tools heavyweight, and time-consuming for developers.

We believe that with the current level of device ownership, the time is ripe for a simple, cross-platform DUI framework to gain traction where others failed to achieve wide adoption. The growing popularity and cross-platform support of web and hybrid applications [32] make HTML5 the ideal choice for our framework, *Panelrama*. Panelrama allows developers to easily create DUIs by making modest modifications to existing tools, rather than formalizing new standards. We observed that, partially due to HTML's document object model, UI elements are commonly arranged in logical groups that should remain together (*e.g.*, the playback controls of video player, or the contents of a menu). We thus provide a new XML element, *panel*, which may be placed around such groupings of controls. Panelrama facilitates the distribution and synchronization of panels among the connected devices.

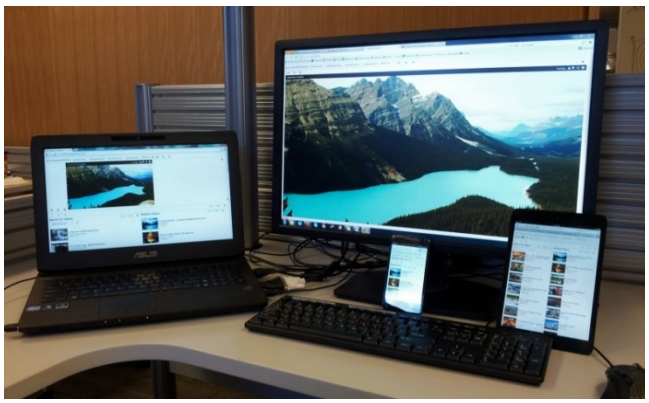

Figure 1. A YouTube browser which automatically annexes connected devices using Panelrama. Application state is maintained on the server, and UI panels are automatically distributed to suitable devices.

Once decomposed as a set of panels, an application is next further modified to specify state information which should be synchronized across devices; for example: the position of the scrubber thumb in the playback controls should be synchronized to the playback position of the video control. Panelrama provides mechanisms to automate synchronization of state, as well as the ability for developers to customize how synchronizations should be handled.

A challenge in DUI framework design is to determine the distribution of the UI across devices. Even in single-user applications, the number of possible combinations of devices can be prohibitive for hand-design of panel arrangements. For example, keeping with our video player application, a developer may want their application to support tablets, smartphones, televisions, public displays, laptops, and desktop computers. *A priori* designing the allocation of each panel to 2^6-1=63 possible combinations of devices is prohibitive. It becomes overwhelming when considering how to handle each new connection.

To address this, Panelrama recognizes that the constituent elements of a given panel may make it more or less suitable for a given device (e.g. a video stream may be more suitable to a television than a mobile phone). Thus, in addition to splitting the interface, our framework lets developers to specify the suitability of panels to different types of devices, given the context of their application behavior. This allows our optimization algorithm to distribute panels to devices that maximize their match for the developer's intent; as devices are added or disconnected, panels are automatically reallocated according to our optimization scheme.

In the following sections, we present related work, and then describe Panelrama in greater detail, including the concept of panels, its architecture, as well as details on the automatic panel allocation. We then demonstrate how Panelrama can be utilized to support a multitude of multi-device application scenarios through three sample applications: a map browser, a video browser, and a PDF reader. Finally, we validate our claims on Panelrama's features and ease of use through a developer study with experienced industry participants.

RELATED WORK

Significant efforts have been expended in the HCI community in the design of interfaces for multi-device applications. While we draw inspiration from this work, we have not focused on innovation in this area. We thus refer the reader to [7] for a thorough survey, and focus our related work on efforts to enable distributed user interfaces.

Past projects have attempted to provide frameworks to support these sorts of applications. One of the early examples of a DUI framework is Multibrowsing [19], allowing users to move web pages between displays. Other works such as WinCuts [29] allows users to distribute sub-regions of window contents to another device. However, the functionality in these early projects are limited, since they do not provide methods for state synchronization and resource modelling, which are answered in more recent works such as Shared Substance [14], VIGO [20], XICE [1], and works by Melchior et al [21]. However, this comes at the cost of changes in existing applications as developers modify their applications to support new models: data-oriented paradigm for Shared Substance, the new architecture introduced with VIGO, and heavy use of native code by XICE and Melchior.

Non-native web frameworks such as WebSplitter [16], on-demand UI migration [12, 13], and automatic partitioning [26] enhance HTML to offer cross-platform DUI, similar to Panelrama. Among these only WebSplitter attempts low-cost migration. However, WebSplitter only synchronizes a single global state for the entire application, making it more difficult for developers to make applications that could scale with any number of devices, or have each device selectively synchronize states for private interactions. It also takes a binary treatment of device characteristics in terms of whether a device possesses certain hardware, making it unsuitable for differentiating devices with different characteristics that perform the same function (e.g. hard vs. soft keyboard).

More importantly, in all aforementioned projects, the burden of UI distribution falls to the user. Ideally, a DUI framework should give the users the flexibility of adjusting the interface components but not require them to spend excess time on the adjustments [18]. Unfortunately, with existing solutions, if the number and variation of devices increases or should the application become more complex, the user can quickly become overwhelmed, resulting in a sub-optimal user experience.

To reduce this burden on the user, one approach is to automate the UI distribution for an optimal experience. For a DUI framework, this automation exists in two forms: 1) assigning UI elements to best-fit devices, and 2) once assigned, optimize UI appearance for the assigned device. Existing works such as SUPPLE [11] and other adaptive interfaces [3] focus on the latter – UI retargeting, by generating UI elements suited to the device on which they are located. Often the new interface is produced through a constrained optimization algorithm. Panelrama does not attempt to replicate these existing solutions, but instead tackles the first problem of assigning UI to devices. Currently we rely on existing HTML5 tools to adapt the contents of a panel to the device it is located on, but we recognize that tools like SUPPLE are superior alternatives for possible future implementations. Instead, currently we draw inspiration from these works in developing our automatic panel distribution mechanism.

From these existing approaches, we determine that while current distributed UI frameworks are powerful and allow for a degree of control, at the same time they necessitate sweeping changes to existing applications. Existing cross-platform web DUI frameworks do an inadequate job of characterizing devices and state synchronization. Finally, nearly all existing frameworks require users to manually specify UI distribution, which becomes overwhelming for complex applications. We developed Panelrama to answer these shortcomings, providing flexible state synchronization, automatic UI distribution, and a simple cross-platform developer experience.

ARCHITECTURE & DEVELOPER EXPERIENCE

We built Panelrama to allow the building of applications in HTML5, enabling truly cross-platform distribution of interfaces. It provides three main features for developers: easy division of UI into panels, panel state synchronization across multiple devices, and automatic distribution of panels to best-fit devices. A guiding principle was to avoid significant changes to traditional application development, ensuring that experienced developers would quickly become facile with Panelrama, and that porting of existing applications would be straightforward. In contrast to earlier works, Panelrama is object-oriented and does not require developers to re-learn and rewrite applications in new paradigms. Instead, Panelrama is designed to use existing technology and facilitate code reuse. To convert an existing application to use Panelrama, a developer needs only to:

1. Wrap groupings of HTML UI elements within a 'panel' tag
2. Complete a Panelrama panel definition, including:
   a. Select state information for synchronization
   b. Rate each panel for its needs with respect to various device characteristics (for automatic panel distribution)
3. Modify business logic to access the new synchronized state information

Post-migration, the application gains multi-device support while retaining existing functionality, allowing it to function identically on a single device as it did prior to the conversion. This frees developers from the need to maintain codebases for both single-device and multi-device versions of the application, since the multi-device version can be trivially made single-device by disabling the panel distribution.

We now describe the development experience of Panelrama, as well as the high level architecture of the framework enabling its function.

Panel Creation & Shared State Information

The code assets for defining a panel are divided into three categories: HTML/CSS, JavaScript, and a Panelrama-specific panel definition. This design makes it convenient for developers to port existing applications to take advantage of Panelrama, since pre-existing assets in HTML/CSS and JavaScript can be reused with only minor changes.

To define a new type of panel, the developer wraps groups of existing HTML within "panel" tags. Since the *id* and *class* properties of the HTML elements are unaltered, this also allows developers to reuse existing JavaScript code that refer to these HTML elements. HTML hierarchy also allows the developer to choose the granularity of UI distribution in Panelrama at any DOM level. This HTML-based division is similar to DOM segmentation, an automatic segmentation algorithm for webpage partition, explored by Romero and Berger [26]. In contrast, Panelrama provides full control to the developer in the allocation of UI components to panels. An example of a divided HTML page is shown in Figure 2.

```
<panel name="VideoPanel">
    <div class="{{panelType}}" id='{{panelId}}'>
        <iframe id='player' class="youtube-player"></iframe>
    </div>
</panel>

<panel name="PlaybackPanel">
    <div class="{{panelType}}" id='{{panelId}}'>
        <button class="button" id="play">Play</button>
        <button class="button" id="pause">Pause</button>
        <button class="button" id="stop">Stop</button>
    </div>
</panel>
```

Figure 2. Sample HTML templates of a video streaming application

After dividing the HTML into templates, the developer creates a short Panelrama-specific panel definition by programmatically extending Panelrama's default *panel object*. A complete definition includes naming the new panel type, setting default state information for new panels of this type, and setting this type of panel's allocation weight for different device characteristics (*e.g.,* Figure 3).

The panel's state information represents the portions of the application's model mapped to the panel's view. This enables Panelrama to update the HTML view of panels to reflect any change in the state information of the overarching cross-device application model. Each state variable contains both its value and a boolean flag indicating whether the variable is to be synchronized with the global state. This provides a built-in solution for synchronizing panel states between devices and is explained in greater detail in the next section.

By defining a new panel type through extending the default *panel object*, the panel gains access to Panelrama's built-in panel allocation functions, these include:

```
Panel.createPanel(panel, destDeviceId)
Panel.removePanel(panelId)
<panel instance>.move(destDeviceId)
<panel instance>.clone(destDeviceId)
<panel instance>.getData(variableName)
<panel instance>.setData(variableName, value)
```

With the business logic's state information now encapsulated in a shared panel instead of stored in the local device, it is necessary to modify the Javascript business logic to access the panel states. Additionally, Javascript code dependent on the DOM elements and event listeners need to be relocated under the *rendered* event of the parent panel, similar to a HTML *Body onload* event, as seen in Figure 4.

```
function VideoPanel(stateVariables, id) {
    this.type = "VideoPanel";
    this.typeName = "Video Panel";
    this.affinity = {physicalSize: 5};

    // playbackState may alternate between "play",
    // "pause", and "stop"
    this.stateVariables = {
        playbackState: {defaultValue: "stop", sync: true}
    };

    Panel.call(this, id);
}
VideoPanel.prototype = new Panel();
```

Figure 3. A panel definition of a video streaming panel.

```
Template.VideoPanel.rendered = function() {
    var panelId = this.firstNode.id;

    $("#player").click(function(e) {
        var clickedPanel = Panel.getPanel(panelId);
        var currentState = clickedPanel.getData("playbackState");
        if (currentState === "play")
        {
            clickedPanel.setData("playbackState", "pause");
        }
    });
};
```

Figure 4. Rendered event of a video streaming panel with event listener and panel state access.

Panel Synchronization

A key to enabling applications to easily span devices is to provide a mechanism to distribute the global state of the application. In Panelrama, the state information of individual panels is defined in the panel definition shown in Figure 3. A Panelrama session includes a list of all connected devices and a list of all available panels. Each device is assigned a certain number of panels, we refer to these panels as *local panels*. Additionally, for every type of panel, we create a shared *global panel* that is not assigned to any device. The developer is given the freedom to choose which state variables a given local panel should sync with the global version using the *sync* flag (see Figure 3). When a panel's state variable is synchronized, any updates to that state automatically change the associated global state (and vice versa), while an unsynchronized state variable does not propagate its changes to the global panel, nor does it receive updates when the global panel is changed. Developers have the flexibility to change the *sync* value at runtime to determine when and how state variables are synchronized.

This solution provides developers the ability to adopt multi-device interaction techniques immediately. For example, a developer of a map application may choose to sync the location of a map panel but not its displayed layers (satellite vs. map view). When a user runs this application on two tablets, scrolling one of the maps will cause the other to scroll, keeping the same area in view. Similarly, changing the *sync* value on runtime allows developers to build a slide presenter app where audience members may "unsync" with the presenter and preview future slides, while being able to resume with following the presenter by "resyncing".

Developers are also free to sync certain variables on-demand. Panelrama provides three methods:

```
// Overwrites global state with local state
<panel instance>.pushData(variableName)

// Overwrites local state with global state
<panel instance>.pullData(variableName)

//Blank method implemented by child panels
<panel instance>.mergeData()
```

An example application making use of this feature is a video player application. For the best experience, a synchronized video panel is shown on the largest screen (*e.g.*, a TV) for group viewing, but uses private unsynchronized video panels to preview videos on tablets or phones. The developer can provide a 'show on TV' button which acts only to initiate a

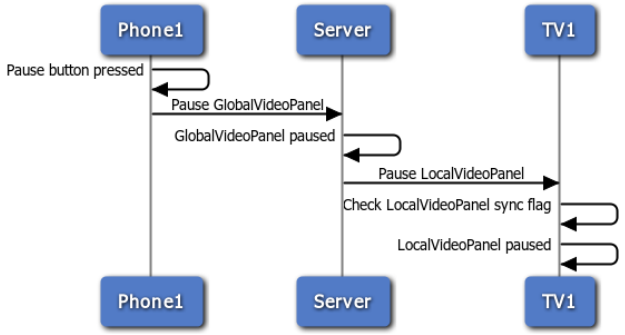

Figure 5. Sequence diagram of state synchronization between global and local panels in a video streaming application.

push synchronization to the global state, which immediately causes the TV to show the video seen on the tablet. The developer could also elect to sync the 'location' field of the tablet and TV videos, allowing the tablet to act as a remote control, or, if so desired, block such control except from a 'master' remote by setting the "sync" flag for one device only, if the application was intended for use in a curated entertainment experience.

The behavior of the *merge* synchronization is based on application logic. For instance, in a multi-user calendar application, a developer might use *merge* to produce a shared calendar showing common available times for all users. This is done by overriding *merge* in the base class.

Panelrama Architecture

Panelrama achieves state synchronization through a client-server architecture. The *frontend* reactivity of Panelrama is made possible through HTML templating technologies. The HTML contents of a panel are wrapped up in individual templates, which are rendered on-demand in the user's browser. In our implementation of Panelrama, we utilized the Handlebars [17] templating framework, although this could be replaced with any other templating framework, including the new HTML5 template tag as it becomes standardized.

The *backend* system of Panelrama uses a MongoDB [22] server to synchronize the panel state. WebSocket technology is used for propagating updates from the server to the clients whenever the state information in a panel changes. An example of the interactions is shown in Figure 5.

Connection Handling

While it is normal for web application developers to manage user accounts, we found early-on that it would be useful for Panelrama to provide a utility to facilitate connecting new devices in order to save the user from having to manually enter a URL and log-in to a web application. We thus included a utility that may be called at any time to generate a QR code on the screen of a device. Application developers may use this utility to share a URL encoding a unique ID, which is captured with another device's camera. Once connected, an application may then take advantage of our automatic panel distribution functionality to distribute the UI across connected devices.

Framework Limitations

Using existing web technologies is one of Panelrama's strengths; however, using them also imposes constraints on the framework. Panelrama's use of templating allows a web application to be split down to individual HTML elements. However, it is unable to split an application further without developer input. For instance, Panelrama does not natively support splitting a single video stream element into three vertical slices to be played on three devices. Achieving this would require the developer to split the video manually, whether by dividing up the video data or hiding the other slices from view.

In order to simplify the developer experience, Panelrama uses a flat panel hierarchy. This covers most common cases and offers developers a single, intuitive division of existing UI. This hierarchy makes the application distribution easily testable using a small test suite for most common devices. Simultaneously, this means a loss of flexibility for advanced users, since nested panels are not supported. This becomes a problem in applications where it makes sense for UI to be combined in some cases and separate in others, such as with color palettes that stick together when devices are few, but split apart when there are many. The trade-off lies in the developer's test suite complexity, which increases quickly with each additional nested layer. We decided against implementing nesting for the initial version of Panelrama, but recognize it as an important addition for future versions.

Finally, migration of an existing application to Panelrama can be made more difficult when the application is tightly coupled between the model and the view. If scripts in one panel references elements in a different panel, the developer needs to manually change the script so that this is no longer the case. However, this difficulty is not unique to Panelrama, existing frameworks which rely on UI division would also suffer from the same issues.

AUTOMATIC PANEL DISTRIBUTION

In addition to synchronizing states between panels, Panelrama enables developers to automatically distribute panels to best-fit devices, in effect, automatically annexing the most suitable device for a given panel [25]. Our optimizer relies on two pieces of information. First, various device characteristics, such as the screen size or presence of a touch sensor. Second, a score from the developer on the relative importance of each device characteristic to the usability of a panel, which we term *panel affinity* score. For example, in a video application, the size of the screen has a bigger impact on the usability of a *video stream* panel than of a *playback control* panel. Therefore a developer would assign a higher value to the *screen size* affinity score on the former panel compared to the latter. Affinity scores are ranked on a standard scale with a minimum score of 0 and a maximum score decided by the developer.

Similar to the approach used in SUPPLE [10] to adapt single-device interfaces, we cast our distribution algorithm as a linear optimization problem, intended to match the needs of panels to the capabilities of available devices.

Device and Panel Modeling

With the latest generation of computing devices, it becomes apparent that one cannot simply model devices by category or form factor. Devices such as the Microsoft Surface [28] or Asus Fonepad [2] combine characteristics from multiple device categories, and even devices within the same category tend to vary in capability. We designed Panelrama to abstract device characteristics away from existing categories for our panel distribution model. We model characteristics of a device D and affinity values of a panel P as tuples:

$$D \equiv (S, C_d), P \equiv (C_p)$$

where S is panel capacity of the device, and C_d represents the set of device-specific characteristics. A device may not be allocated panels exceeding the number specified by S. S is an abstraction of the capacity of the device resulting from a combination of resolution, pixel density, physical screen size, and panel UI size. Since currently it is not possible to correctly detect physical size or pixel density of a device, S is currently hand-coded. Finally, a panel's C_p is a set identical in size to C_d, representing the panel's affinity values. This representation is similar to one used by Vandervelpen *et al*. [30], but several key differences become apparent in implementation.

In Panelrama, both C_d and C_p are represented as a set of integer scores ranked on a standardized scale determined by the developer. By default, Panelrama supports allocation based on four device characteristics: *physical size*, *keyboard quality*, *touch quality,* and *mouse quality*. A developer may choose to extend this list to include additional characteristics as needed, such as presence of GPS sensors or a 3G radio.

Ideally, a DUI framework should detect and assign the default C_d scores programmatically, while the application developer needs only to define panel C_p scores. However, the current web standard does not support query on certain characteristics, such as physical screen size or presence of a physical keyboard, but device description repositories like WURFL [31] give us hope that soon such automatic scoring may be possible. At the moment, however, it is necessary for C_d scores in our sample application to be hand-coded. Our code for sample C_d and C_p scores can be seen in Figure 6.

To calculate the cost function, we treat each C_d and C_p as a vector. For every combination of C_d and C_p, we calculate the dot product of the vectors as a usability score, which we use to populate a cost matrix. Given P panels and D devices, we get a cost matrix C of size $D \times P$, where C_{dp} is the usability score of panel p on device d. For example, the usability score of a *VideoPanel* on the *Tablet1* in Figure 6 computes to *20*. These scores form the basis of the objective function used to solve the linear optimization problem of panel allocation.

```
Tablet1.characteristics = {
    physicalSize: 3,
    keyboardCapability: 2,
    touchCapability: 5,
    mouseCapability: 0,
    readyToHand: 4
};
```

```
VideoPanel.affinity = {
    physicalSize: 0,
    keyboardCapability: 0,
    touchCapability: 0,
    mouseCapability: 0,
    readyToHand: 5
};
```

Figure 6. Sample device characteristics for a connected tablet and panel affinity values for the *video stream* panel.

Linear Optimization Function

Formally, we represent the optimization problem as a linear sum assignment problem:

There are P panels assigned to D devices. Given cost matrix C, the usability score of panel p on device d is C_{dp}, for $d = 1, \ldots, D$ and $p = 1, \ldots, P$.

We want to arrive at assignment x_{dp} which maximizes the usability score, for $d = 1, \ldots, D$ and $p = 1, \ldots, P$, where x_{dp} represents the allocation of panel p on device d. Additionally, no panel instance p may be assigned to more than one device d. No device d may be allocated panels exceeding device capacity S_d. This gives us our objective function as shown in Equation 1.

Maximize:

$$\sum_{d=1}^{D} \sum_{p=1}^{P} C_{dp} x_{dp}$$

Subject to:

$$\sum_{d=1}^{D} x_{dp} \leq 1 \; for \; p = 1, \ldots P$$

$$\sum_{p=1}^{P} x_{dp} \leq S_d \; for \; d = 1, \ldots D$$

Equation 1. The objective function defining the linear optimization for optimal panel distribution.

Whenever a panel or device is added or removed, Panelrama computes the optimization problem using the GNU Linear Programming Kit [15] and updates the panel allocations on each device. Performance testing showed that optimal solutions were always found within 200ms for extreme examples of up to 50 panels distributed across 50 devices.

Naturally, there exist scenarios where users could find the automatic redistribution of panels disruptive. Each panel features a *lock flag* that when set, adds an additional constraint to the optimization problem to retain the panel on its current device. This flag may be exposed by the developer to the users through a toggled button.

Optimization Development Tool

In the ideal case, developers should need only assign the affinity values of panels based on the importance of each characteristic alone. However, some developers requested a tool to help them test their parameters. We therefore built a tool that allows developers to enter values and preview the results of the optimization in a GUI. This allows simulation of various device configurations for testing. The tool displays the current affinity values for each defined type of panel, as well as the cost matrix entry of the type of panel to each device connected to the application. Developers can make changes to the affinity scores within the tool, allowing them to preview the panel allocation results prior to assigning final affinity values in the code.

SAMPLE APPLICATIONS

In order to demonstrate the capabilities of Panelrama, we provide three sample applications developed using the framework: a distributed digital map, a video streaming application, and a PDF reader.

Distributed Map

The distributed map is intended, in part, as an exploration of user-initiated distribution of panels. In this application, the user is given full control over panel allocation and reassignment; our allocation algorithm is not used. The distributed map was developed using the Panelrama framework and the Google Maps API. The application's UI includes five panels: a *map*, a *map navigation*, a *map layer*, a *street view*, and a *street view directional* panels.

The application's first function is to allow the user to navigate a digital roadmap. This is represented using a map panel containing the following states: the map's geographic coordinates, zoom level, and the type of map being displayed. When the user loads the application on a single device, it appears as a traditional mapping application, including controls for both navigating through the map and changing the current layer being displayed.

When the user connects a second device to the application, differences quickly come to light: each panel may be added, removed, or transferred to a different device. This enables users to move panels between devices, such as from a smartphone to a projector in order to give directions to a group. The panel can then be controlled remotely through the map *navigation* panel, or by keeping another instance of the *map* panel fully synchronized on the original device.

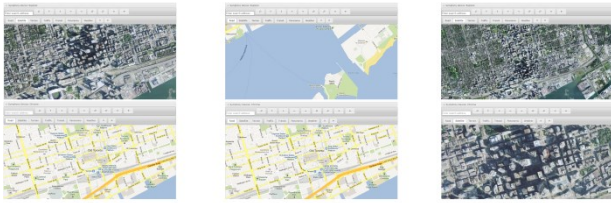

Figure 7. Different combinations of state synchronization between panels: (a) different information, (b) different locations, (c) different zoom levels.

Figure 8. Panoramic street view across three tablets. Geo-location is synchronized between the devices, but direction is programmatically offset by 90°.

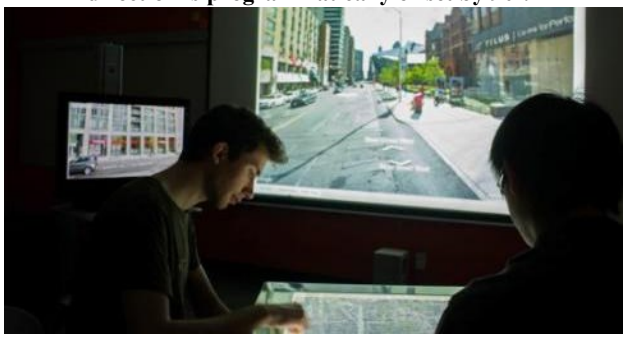

Figure 9. A recreation of some of the multi-surface Google Earth application created by Forlines *et al.* [9]

When multiple map panels are created on multiple devices, their state information can be synchronized in any combination to produce, for example: displays of different information on the same location, different locations at once, or different granularities. This example of Panelrama's state synchronization allows users to compare maps on multiple devices to gain access to different information, as seen in Figure 7.

The map application's second functionality enabled by Panelrama is that of the panoramic street navigation, demonstrated in Figure 8. By synchronizing the location of Google's street navigation service but allowing for an offset in view angle, users are able to use three devices to provide a 270 degree view of a street. This functionality is difficult to replicate on a single device and is a wonderful example of a potentially new interaction enabled by the synchronization behavior in Panelrama.

Panelrama may also be used to enable distribution of panels across other types of devices. Forlines and his colleagues demonstrated a modification of Google Earth that enabled a multi-touch table to control the location of cameras in 3D views on nearby vertical displays [9]. We connected a Microsoft Surface - Samsung SUR40, 40" multi-touch table to our mapping application allowing us to recreate much of the experience offered by the application described by Forlines' *et al*, without having to write any additional code.

Video Streaming Application

The second application was developed using Panelrama and the Google YouTube API. Our shared YouTube browser includes four panels: a *video stream* panel, a *playback controls* panel, a *search* panel, and a *related videos* panel.

The *video* panel holds state information such as the current playtime, the playback status (playing, paused, stopped), and the video's unique YouTube identifier. These states are controlled and changed by the three remaining panels. Panelrama's synchronization mechanism allows the distribution of a synchronized *video* panel to a television, and multiple users to interact with the global *video* panel using controls on personal phones or tablets. The user also has an option to add a personal *video* panel that is not synced to the group video, allowing the user to search for and preview videos without interrupting the group. When the user has selected an appropriate personal video, she may choose to use the *video* panel's "push" functionality to share the new video with the group.

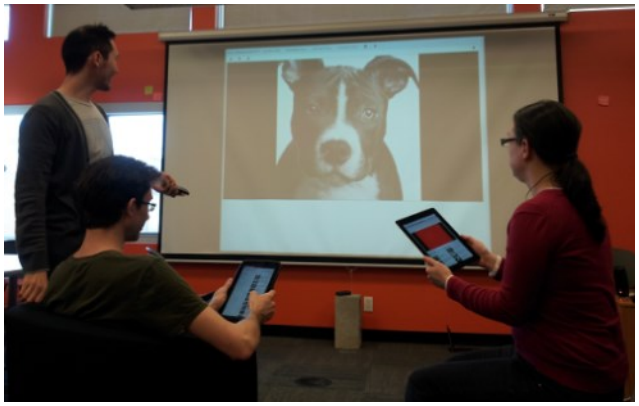

Figure 10. Multiple users using the YouTube application.

The video streaming application is also designed to take advantage of Panelrama's automatic panel allocation feature. As mentioned in the previous section, the developer defines how important each device characteristic is for a panel (*e.g.*, physical *size* is important for a video panel, and *ready to hand* is important for playback controls), and Panelrama assigns panels dynamically to a user's devices as each device or panel is added or removed.

For instance, a user who begins with all panels on a desktop computer may have the video panel automatically distributed to a newly connected television, followed by playback controls to a connected smartphone, leaving only search functions on the desktop. When a tablet is connected, all the playback and search controls migrate to the tablet, minimizing the number of devices a user needs to reach to interact with the video on the television. An example interaction sequenced is shown in Figure 11.

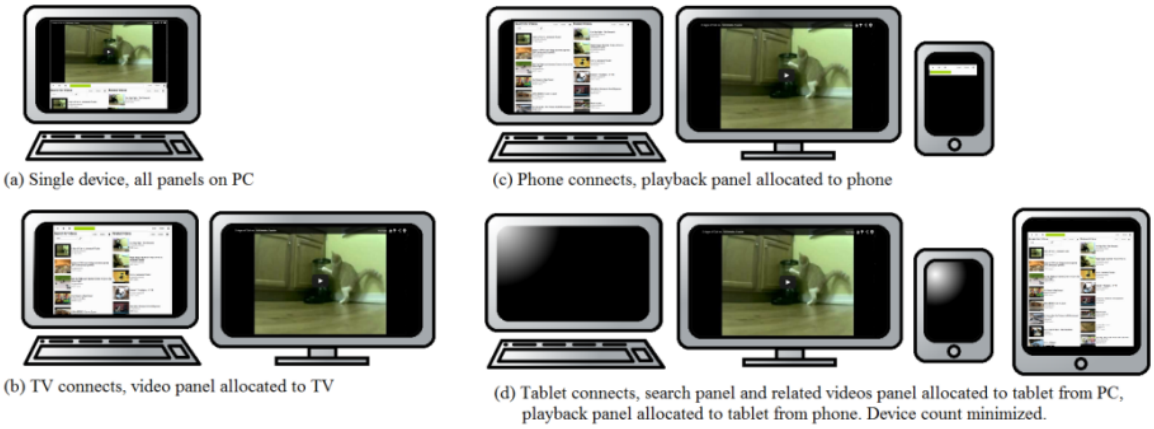

Figure 11. Example of automatic panel distribution in our sample application. The handling of additional devices of the same type may be customized by the developer (eg: allow multiple *search* and *related videos* panels for additional users).

Figure 12. The reader application displaying 4 pages of a paper. Top tablet locked with a helpful diagram, another, outside the frame, is locked to a reference list.

Distributed PDF Reader

The third sample application is a distributed PDF reader, inspired by the multi-slate reading environment described by Chen *et al.* [4]. Our application uses additional tablets to increase its screen real estate. It is unique from the previous two applications in that it takes advantage of all additional devices, while the other applications could only utilize as many devices as there are types of panels before duplicating functionality.

This application's interface uses only one panel: the *page* panel. As each device is added, the new device is allocated a new instance of the panel, which is set to display the next page of the document. When any tablet scrolls to a new page, all synchronized tablets also scroll to the successive page.

The user may also lock any panel to a certain page. This allows the user to assign pages permanently to a panel for ease of reference, such as diagrams and reference lists. When a user chooses to lock a page to a tablet, the synchronized tablets displaying a successive page all scroll up by one to fill in the void left by the locked tablet. Figure 12 shows an interaction example.

These sample applications have demonstrated the various functionalities of our toolkit. In particular, we have demonstrated how recent DUI user experience explorations may be implemented using the framework. To validate our belief Panelrama is easy to use, we conducted a study.

DEVELOPER STUDY

We conducted a study of Panelrama to evaluate its usability for developers. Specifically, we wanted to learn whether Panelrama's two features of synchronization and panel allocation are easy to use, and, in general, if Panelrama provides easy development of cross-device applications, with a focus on porting of existing applications. Finally, we wanted to solicit professional developer feedback on Panelrama, including opinions on the panel concept and its suitability for industry projects.

There exist several proposed methods for evaluating a development framework, including user studies [27], formal API reviews [8], and usability metrics such as the cognitive dimensions framework [5]. The latter two methods tend to focus on usability issues that occur in larger frameworks, such as "learning style, penetrability, consistency, and role expressiveness" [8]. Owing to the small size of Panelrama, where developers call upon only one class, these are less suitable. Consequently we decided to use the first method - the user study, to evaluate our framework.

Participants

We recruited eight professional web developers between the ages of 21 and 33 each with 2 to 14 years of industry experience. As an indicator for frequency of adopting new tools, we asked participants to indicate the number of new API libraries / frameworks learned in the last year; their responses were: 1-3 tools: 4, 4-6: 2, 7-9: 2. Participants were paid $150 for a 2-hour session.

Tasks

Participants were first asked to write a simple finger-painting application for a single device, without yet being introduced to Panelrama. This provided a common task for all developers and would allow them to work with their own application during the conversion process. Finger painting was chosen to have sufficient complexity to allow the developers to later experience all of the features of Panelrama yet simple enough to complete in the time allotted. Figure 13 is a participant's implementation.

Once complete, developers were then asked to alter their application so that it would split across devices, with the panel on the larger screen, and the palette on a screen that is touch-enabled. This provided developers the opportunity to work with Panelrama to divide their application into panels, as well as to work with property synchronization.

Once completed, they were asked to further subdivide the palette so that each color would present on a different device, if available, so that their application would then span up to four screens. This task forced developers to work with Panelrama's panel affinity and automatic allocation mechanisms to achieve the desired allocations.

Procedure

Participants first completed a consent form and a pre-study questionnaire. They were then provided with instructions for completing the three tasks, and online documentation about the Panelrama framework for their reference. The documentation included sample code for a Youtube application implemented in Panelrama. In addition, participants were free to use the Internet to look up resources

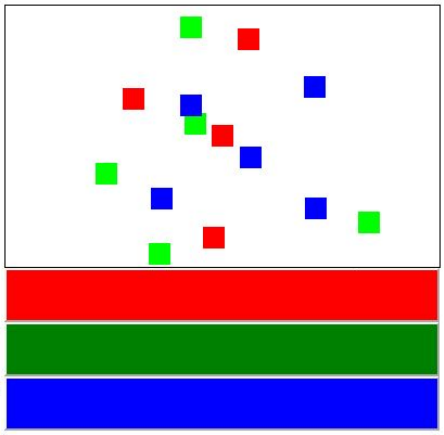

Figure 13. An example of the application, pre-conversion, a participant developed in the first phase of the study. Bottom: color palette. Top: 'painted' canvas.

for completing the first task. Finally, participants were permitted to ask clarifying questions regarding the task instructions and the Panelrama documentation.

We measured the time required for participants to finish each task. In addition, we took source code snapshots of the web application at the end of each of the three tasks. This allows us to determine the number of lines of code (*LOC*) that needed to be changed for the conversion, as well as the lines of code that developers could reuse. Upon completion of the three tasks, participants were given a post-study questionnaire, where they rated their impressions of Panelrama using a number of metrics

Quantitative Results and Code Analysis

Quantitatively, developers on average took 83.4 minutes to complete all three tasks. Individually: Task 1: 30.2, Task 2: 38.5, Task 3: 14.7. Further, on average, developers changed 33.1 lines of code and kept 43.1 when moving between Tasks 1 & 2. No developer changed any code to complete task 3, as only adjustment of panel affinity parameters was required. Not all 'changed' lines of code were completely new; many involved only minor modifications.

All participants were able to complete the three tasks within the 2 hour time limit and correctly use Panelrama. The quantitative results suggest that Panelrama strongly facilitates code reuse since more of the code was reused than added or modified. Due to the simplicity of the sample application, it is safe to assume that the ratio of reused to changed code would rise in a real-world scenario.

While most participants took longer to convert the application to Panelrama in Task 2 than to create the application in Task 1, Task 2 included participants learning the framework. During the study we observed that, after defining their first panel, participants defined subsequent ones much more quickly, and we would expect the time to continue to decrease as a project grows. The participants share our opinion, as indicated in their feedback in the following section.

Qualitative Developer Feedback

Developers were asked to provide feedback in the form of Likert-scale questions, as well as in oral discussion. In addition to a focus on usability of the framework, we asked the developers to professionally assess Panelrama for its suitability to the task of cross-device enabling their existing web applications. Developers were enthusiastic about using Panelrama for multi-device application building. They reported that Panelrama was easy to learn (median score of 4 on a 5-point Likert scale) and enabled easy conversion of existing applications (median = 4). They also believed that once learned, Panelrama would be easy to use in the future (median = 4.5).

Conceptually, participants indicated that they strongly preferred using panels over developing separate applications (or websites) for different devices (median = 5), and believed that automatic allocation of panels is a necessary feature (median = 5). In the hypothetical case that they were asked to develop a multi-device application, all participants were strongly in favor of choosing Panelrama for development (median = 5).

Developers did provide some critical feedback. The first participant (P1) commented that Panelrama's design "*might have a hard time supporting use cases where developers want to split nested UI elements*", a limitation that we plan to address in future. P2 raised concerns with each panel requiring three pieces of information, since it introduces three points of failure when the names might be misspelled. P3 and P6 were worried about excess boilerplate code since the three button panels were identical except in color choice and shared large chunks of code. To resolve this, P3 suggested allowing developers to define a generic panel that could be inherited. Lastly, P6 expressed concerns with the degree of abstraction of affinity scores in automatic allocation and suggested that a labeled scale or enumeration be used to identify them.

DISCUSSION AND FUTURE WORK

Panelrama was designed to facilitate easy conversion of existing web applications to enable cross-device interaction, allowing users to leverage the many devices which surround them every day. Feedback from professional developers indicates we have largely hit the mark, and that they would be highly likely to use Panelrama given the goal of creating such an experience. Nonetheless, some issues remain which will be the focus of our ongoing development of the framework.

Adaptive UI and Device Identification

The current implementation of Panelrama is unable to take full advantage of its panel allocation properties due to the inability to detect device characteristics using existing software frameworks. As an improvement, we would like to explore formal specifications to provide developers with the complete list of a device's characteristics, such as physical size and sensor suites.

Multi-User and Security

At present, Panelrama does not provide an identity service to differentiate between devices belonging to different users. This allows developers to be flexible in selecting services of their choice (e.g. Facebook, Google ID) for the application without replication from Panelrama. In the future, we intend to explore multi-user applications with automatic user identification and identity-aware panel access privileges.

We also intend to improve Panelrama by tackling security concerns since the allocation of panels to potentially compromised systems poses a security risk [25]. Our solution involves allowing developers to define content restrictions for public and private devices. Panels containing sensitive information would only remain on private devices or the sensitive content would be censored when moved to an untrusted device.

Opportunistic Annexing and Dynamic Linking

More ambitiously, our long term goal for Panelrama is to provide the ability for opportunistic annexing of public devices. We will investigate hardware and software solutions to dynamically establish direct links between multiple devices to improve latency. We will also implement adaptive interface designs to not only allocate the ideal panels for a device, but also modify the panels to use UI elements considered most usable on the device.

CONCLUSION

In this paper, we presented Panelrama, an HTML5-based DUI framework. Through the combination of UI distribution, automatic panel allocation based on device characteristics, and support for developer extensibility, Panelrama provides a solution for allowing users to interact with a single application from multiple devices. We have also demonstrated three sample applications, a map browser, YouTube browser, and distributed electronic reader, which demonstrate the various functionalities of the Panelrama framework. We believe that HTML-based frameworks, which minimize the additions required for distribution of user interfaces, show great promise for the realization of DUIs as a viable interaction paradigm.

ACKNOWLEDGEMENTS

We thank Peter Hamilton, Steven Sanders, Katie Barker for their ideas and support, the DGP Lab members for their advice, and the study participants for their kind feedback.

REFERENCES

1. Arthur, R., and Olsen, D. (2011). XICE windowing toolkit: Seamless display annexation. *ACM Trans. Comput.-Hum. Interact*. ACM Press, 18, 3, Article 14, 46 pages.
2. Asus Fonepad. http://www.asus.com/Tablets_Mobile/ASUS_Fonepad/
3. Calvary, G., *et al*. (2003) A unifying reference framework for multi-target UI. *Interact. Comput*. 15,3, 289–308.
4. Chen, N., Guimbretiere, F., and Sellen, A. (2012). Designing a multi-slate reading environment to support active reading activities. *ACM Trans. CHI*. 19, 3, A. 18.
5. Clarke, S., and Becker, C. Using the Cognitive Dimensions Framework to evaluate the usability of a class library In *Proc. Joint Conf. EASE & PPIG 2003*, (2003), 359-366
6. Demeure, A., Calvary, G., Sottet, J., and Vanderdonkt, J. A reference model for distributed user interfaces. In *Proc. TAMODIA 2005*, ACM Press (2005), 79-86.
7. Elmqvist, N. Distributed User Interfaces: State of the Art. *In Proc. DUI 2011*. ACM Press (2011), 7-13.
8. Farooq, U. *et al.* (2010) API usability peer reviews: a method for evaluating the usability of application programming interfaces. *CHI 2010*. 2327-2336.
9. Forlines, C., Esenther, A., *et al.* (2006). Multi-user, multi-display interaction with a single-user, single-display geospatial application. *UIST '06*. 273-276.
10. Gajos, K., and Weld, D.S. (2005). Preference elicitation for interface optimization. *UIST '05*. 173-182.
11. Gajos, K., and Weld, D.S. (2004). SUPPLE: automatically generating user interfaces. *IUI 2004*, ACM Press, 93-100.
12. Ghiani, G., Paternò, F., and Santoro, C. (2010). On-demand cross-device interface components migration. *MobileHCI 2010*, 299-308.
13. Ghiani, G., Paternò, F., and Santoro, C. Push and pull of web user interfaces in multi-device environments. In *Proc. AVI 2012*. ACM Press (2012), 10-17.
14. Gjerlufsen, T. *et al*. Shared substance: developing flexible multi-surface applications. In *CHI 2011*. ACM Press (2011), 3383-3392.
15. Glpk.js. http://hgourvest.github.com/glpk.js/
16. Han, R., Perret, V., and Naghshineh, M. WebSplitter: a unified XML framework for multi-device collaborative Web browsing. In *Proc. CSCW 2000*, 221-230.
17. Handlebars.js. http://handlebarsjs.com/
18. Hutchings, H.M., and Pierce, J.S. DIAMOND: A Framework for Dividing Interfaces Across Multiple Opportunistically Annexed Devices. *GVU Technical Report* (2005), GIT-GVU-05-21.
19. Johanson, B., Ponnekanti, S., *et al*. (2001). Multibrowsing: Moving Web Content across Multiple Displays. In *Proc. UbiComp 2001*. Springer-Verlag, 346-353.
20. Klokmose, C.N., and Beaudouin-Lafon, M. (2009). VIGO: instrumental interaction in multi-surface environments. *CHI '09*. 869-878.
21. Melchior, J. *et al.* (2009). A Toolkit for Peer-to-Peer Distributed User Interfaces: Concepts, Implementation, and Applications. In *Proc. EICS 2009*, 69-78.
22. MongoDB. http://www.mongodb.org/
23. Mori, G., Paternò, F., and Santoro, C. (2003). Tool support for designing nomadic applications. *IUI 2003*, 141-148.
24. Peñalver, A., Lazcorreta, E., *et al.* (2012) A. Schema driven distributed user interface generation. *INTERACCION 2012*. Article 1, 8 pages.
25. Pierce, J., Mahaney, H., Abowd, G. (2003). Opportunistic Annexing for Handheld Devices: Opportunities and Challenges, *GVU Tech Report*, GIT-GVU-03-31.
26. Romero, R., and Berger, A. (2004). Automatic Partitioning of Web Pages Using Clustering. *Mobile HCI, volume 3160 of Lecture Notes in Computer Science*, 388–393.
27. Stylos, J., Graf, B., *et al.* (2008). A Case Study of API Redesign for Improved Usability. *VLHCC 2008*, 189-192.
28. Surface by Microsoft. http://www.microsoft.com/surface/
29. Tan, D. S., *et al.* (2004). WinCuts: manipulating arbitrary window regions for more effective use of screen space. *CHI EA '04*, 1525-1528.
30. Vandervelpen, C., Vanderhulst, G. *et al.* (2005). Light-Weight distributed web interfaces: preparing the web for heterogeneous environments. *ICWE 2005*, 197-202.
31. WURFL. http://wurfl.sourceforge.net/
32. Xanthopoulos, S. and Xinogalos, S. (2013). A comparative analysis of cross-platform development approaches for mobile applications. *BCI 2013*, 213-220.