Disperse: Enabling Web-Based Visualization in Multi-screen and Multi-user Environments

Megan Monroe(✉) and Casey Dugan

IBM Research, 1 Rogers Street,
Cambridge, MA 02142, US
mmonroe@us.ibm.com

Abstract. For visualization developers, the design and construction of effective applications can often feel like a war against screen space. Every now and then, developers are tasked with building a visualization that will live exclusively on a large, high-resolution display. More often than not, however, visualizations must be built to survive across the varying screen sizes of laptops, tablets, and phones. This may explain why many developers have flocked to the web, where stylesheets can easily be swapped and modified to tailor an application's look and feel to the current screen size. But that *screen* is defiantly singular. If developers want to tap into a more elaborate hardware ecosystem, they must take on the additional workload of server-side or device-specific coding. To this end, we introduce Disperse, a server-based framework that allows developers to encode multi-screen capabilities into web-based visualizations using a simple set of client-side mark-ups. The framework is intended primarily for authoring new visualizations, but can also be used to add multi-screen capabilities to existing visualizations. Disperse not only imposes minimal time and complexity overhead on the development and deployment of these visualizations, as we show through five case studies, but also allows multi-screen visualizations to be realized across any set of web-enabled devices.

Keywords: Visualization · Authoring · Multi-screen · Multi-user · Collaboration

1 Introduction

For visualization developers, every pixel counts. As datasets become larger and more complex, it is increasingly rare for a single view of the data to sufficiently address a meaningful range of the potential questions that might be explored. As a result, developers frequently build visualizations comprised of multiple linked views of the data, each designed to address a unique set of questions or contexts. This approach has been widely adopted in both research [4, 12, 23] and industry [29, 32, 35].

When a multi-view visualization is being designed specifically for a large, high-resolution display, developers have a lot of freedom in deciding how many views of the data to include and how complex those views should be. However, this luxury is rare. Most visualizations must be built without prior knowledge of the hardware on which it will be deployed. In these cases, developers must build their application to

J. Abascal et al. (Eds.): INTERACT 2015, Part III, LNCS 9298, pp. 418–435, 2015.
DOI: 10.1007/978-3-319-22698-9_29

function on a standard laptop screen, which means that they must either limit the number of views that can be displayed at one time, or limit the complexity of the visualizations within those views. The ultimate design can be heavily dictated by the limited number of pixels in a typical display.

One could argue though, that pixels are never in short supply. Not every room is equipped with a wall-sized display, but walk into a typical meeting, and what do you see? There are probably between two and ten people sitting around a table, each with a laptop, a tablet, or, at the very least, a smartphone in front of them. Maybe the voices of remote collaborators emanate from the speakerphone system. There are probably computer screens in front of those people as well. In short, there are pixels everywhere! They're simply not connected in a way that is readily accessible to developers.

But all of these screens, from tablets to high-resolution displays, are connected by at least one feature, which is that they can all typically run a web browser. This is precisely why visualization developers have flocked to web development. Not only is it easy to customize the look and feel of an application based on varying screen sizes, but developers can reliably assume that their application can be quickly deployed on virtually any device. Additionally, a new fleet of approachable visualization libraries, such as Processing.js [28], D3.js [9], and Three.js [37], has made it increasingly easier to build web-based visualizations (Fig. 1).

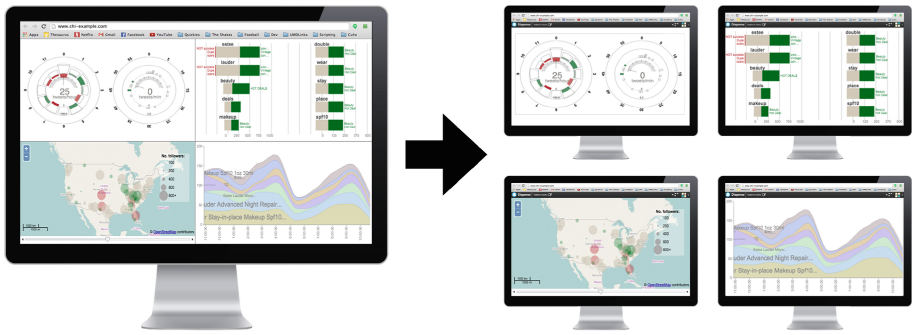


Fig. 1. The Disperse framework allows multi-view web-based visualizations to be easily split across multiple screens and multiple devices to enable more data-rich displays, provenance, and both remote and co-located collaboration.

Our goal with this work was to leverage the prevalence of web development skills, and the natural constructs of client-side programming languages to give visualization developers access to any pixels they need, regardless of what device they reside on. We introduce Disperse, a web-based framework that enables the development of multi-screen and multi-user visualizations. Disperse offers a unique set of advantages over other approaches for realizing multi-screen applications:

1. Developers are tasked only with client-side web programming. For many developers, this is well within their wheelhouse.

2. A web-based implementation means that (1) developers do not need to worry about the specific hardware configuration that their application will be deployed on and (2) end users are tasked only with opening browser windows and pointing them at the correct URLs in order to use a Disperse visualization.
3. The framework comes with history tracking and version control capabilities built in, further saving developer time and effort.

Given the complexity of executing multi-screen applications using other approaches, we argue that this is a substantial simplification of the overall process. This paper is organized as follows: in the following section, we discuss current approaches for creating and deploying multi-view visualizations. We then present the Disperse implementation, and walk through the process of building a multi-screen visualization. This is followed by five case studies that demonstrate the use of Disperse from the perspective of visualization developers. Finally, we discuss limitations and future work before a final conclusion.

2 Related Work

In 2000, Baldonado et al. advised that multi-view visualizations should be "used minimally" due to limited screen space and the increased demand they impose on cognitive attention [3]. Despite this warning, multi-view visualizations have become the norm across research and industry in order to support larger datasets and multi-faceted decisions. These visualizations typically adhere to one of the following four strategies:

2.1 Keep It Simple

When multiple views are presented on the same screen (tiled displays), users have the advantage of seeing the relationships between these views and understanding how a change to one will affect the others. However, this approach limits the screen space that can be allocated to each view. Because of this, each individual view must be kept relatively simple, limiting developers to scatterplots, line graphs, and pie charts (Fig. 2).

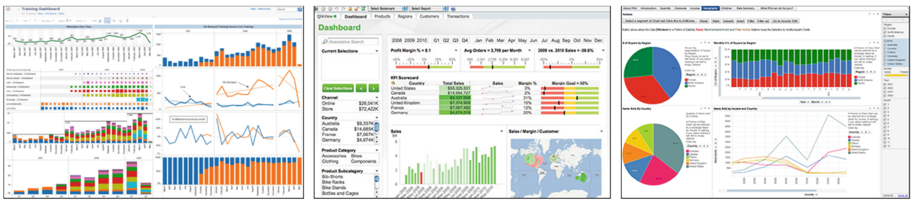


Fig. 2. Tableau (left) [35], QlikView (center) [29], and Spotfire (right) [32] - three of the biggest industrial visualization platforms - all allow users to create dashboards of simple views that are tiled across a single screen.

2.2 Hide and Seek

When more complex views are needed, the alternative to the above strategy is to display a single view at a time, and allow users to navigate to the additional views (navigational displays). This is usually accomplished using tabs or a menu of views [6, 15, 22, 33]. However, this strategy prevents users from seeing multiple views concurrently and directly observing how changes in one view affect the others. Obviously users can "fake" a multi-screen experience by opening independent instances of the application on different devices, and navigating to a different view on each device, but then the linking between the views is lost.

Disperse is designed to capture the advantages of both tiled and navigational displays. Developers can plan for each view to take up a full screen of its own, much like navigational displays, and users can see all of these views and the interactions between them at once, just as they can using tiled displays.

2.3 Go Big

Large, high-resolution displays offer an obvious solution for multi-view visualizations. They can display more data, more details, and provide a host of other benefits including leveraging spatial memory and facilitating collaboration [1, 38]. However, these displays are comparatively scarce. When developers build tools specifically for one of these large displays, they are essentially ruling out the use of their tool in the vast majority of collaborative, decision-making scenarios. Our goal with Disperse was to allow users to take advantage of large displays when they are present, but not be left entirely without a solution when they are not. Using Disperse, it is just as easy to position multiple views across a large display as it is to run each of those views on separate laptop screens.

2.4 Device-Sprawl

A considerable amount of previous work has focused on applications that can run across distributed user interfaces (DUI's). One of the biggest challenges of DUI's, however, is maintaining clear boundaries between development, deployment, and usage. Many efforts in this space require developers to have some understanding of the deployment environment, which can involve custom synchronization software [11, 14, 17, 34] and/or pre-determined tasks [7, 8, 16], and end users must have these environments available to them and properly configured. This can be a steep requirement for real world meetings, which get scheduled on the fly, can be brief in duration, and frequently involve non-technical users.

Much like the current trend in visualization, DUI applications are resolving these challenges by migrating to the web. Web-based DUI solutions began with efforts to facilitate collaborative browsing [5, 24] and have evolved to support more general functionality [10, 13, 39]. Badam and Elmqvist recently introduced a web-based framework that supports collaborative, multi-device interaction for single-view visualizations [2].

With Disperse, we extend this concept to multi-view visualizations, and propose an alternate strategy for allocating screen space and synchronizing interaction that is designed to require minimal additional coding effort from developers.

3 Disperse Implementation

Disperse is a server-side application that hosts other HTML code, much like the popular tool, JSFiddle [19]. Developers access the framework by either installing Disperse on the server that is currently hosting their visualization, or by uploading their code to a server that is already running Disperse. The framework is currently implemented as a J2EE Web Application, running on a Tomcat Application Server, however, this was simply the most accessible configuration available at the time of development. Disperse could just as easily be implemented in Node.js [25] or other server-side environments. The framework allows developers, using a simple set of client-side mark-ups, to split a single webpage into multiple browser windows across any set of web-enabled devices without losing the linking between these views.

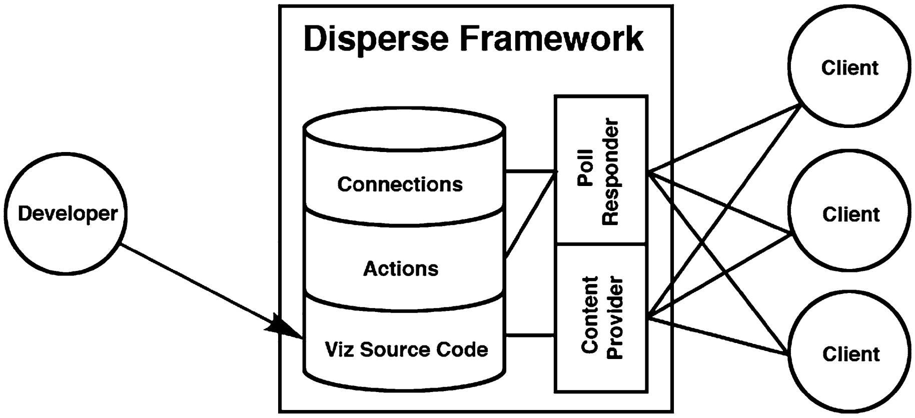


Fig. 3. The Disperse architecture consists of three data stores, which maintain (1) the source code of the multi-screen visualizations, (2) the clients that are connected to each visualization, and (3) a running list of actions that have been performed.

3.1 Visualization Development

Disperse tasks developers with two coding requirements: (1) indicate which HTML elements constitute the different views of their visualization, and (2) indicate which JavaScript functions enact meaningful changes across these views.

The first requirement is met using simple CSS identifiers. Developers must add the Disperse screen class, `<div class="disperse_screen1">`, to any HTML element that encapsulates a unique view of their visualization. Each element with the `disperse_screen` class will be able to function as a stand-alone view in its own browser window when the application is run through Disperse. Since CSS classes are additive, this class can be appended to an existing element, or it can be the sole class of a new `<div>` element that simply wraps around the relevant content. To further facilitate the screen-tagging process, we created two Disperse development templates, one navigational and one tiled, which come with pre-tagged `disperse_screen` elements in which developers can build their various visualization views (Fig. 4).

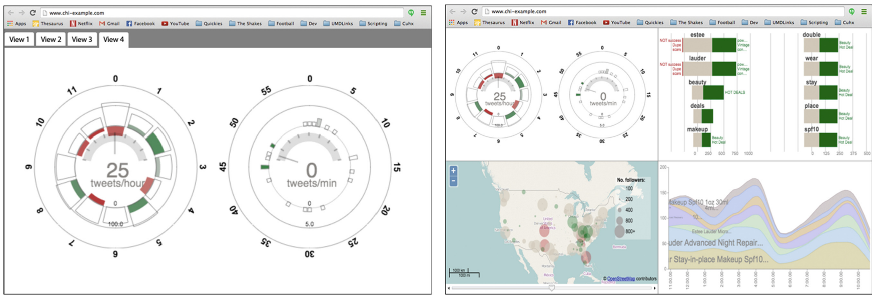


Fig. 4. We provide a tab-based, navigational template (left) and a tiled template (right), which can be used to build visualizations both within and outside of a multi-screen environment.

The second requirement that Disperse imposes on developers is to indicate which JavaScript functions propagate meaningful changes to one or more views within the application. This done by prepending the word `synchronize` to the function name (e.g. `function synchronize_applyFilter{}`). It is the actions of these functions that will be replicated across the other views (and thus screens) of the application. This function tagging can be done both at the end of development or to existing applications, however, the primary intent is for developers to build in these synchronized functions from the ground up. This allows developers to design their application around a comprehensive, yet minimal set of synchronized functions. For example, helper functions, and functions that modify a view temporarily (such as hover effects) do not necessarily need to be synchronized. The function-level synchronization also means that developers can leverage any web-based visualization libraries, including Processing.js [28], D3.js [9], and Three.js [37].

While we considered many alternate approaches for both splitting content onto multiple screens and synchronizing the actions between them, we chose this tagging approach for three primary reasons:

- It involves only client-side coding. Developers use the same set of constructs that they would use to build a single-screen application.
- The required mark-ups piggyback off the natural constructs of HTML, CSS, and JavaScript. It is difficult to imagine a discernable area of a webpage that is not encapsulated in `<div>` tags (or some equivalent), or a user interaction that is not routed through a JavaScript function.
- Neither the screen tags nor the function synchronization tags have any effect on the functionality of the application if it is run outside of Disperse. We wanted the same code base to function in both multi-screen and single-screen environments.

The goal of Disperse is to allow developers to construct multi-screen experiences as quickly as they can imagine them. The framework makes it extremely easy to prototype multi-screen concepts, a task that has been only minimally addressed in previous DUI efforts [5]. Developers do not need to tailor their application to a specific physical environment or implement any sort of synchronization between devices. We wanted developers to feel like adding multi-screen capabilities was just as accessible as importing a JavaScript library, much like jQuery has become a ubiquitous inclusion [18].

3.2 From Mark-Ups to Multi-screen

When end-users (clients) connect to a Disperse visualization, they are assigned a connection profile, which includes a connection ID, a branch ID (to be discussed later), the visualization ID, and a screen number. This information is maintained in the framework's "Connections" database (see Fig. 3). Disperse then serves up a webpage that includes a minimal navigation header (described in the next section) followed by an iFrame that fills the remainder of the browser window. The source code of the developer's visualization is loaded as the content of the iFrame. Since this code has been uploaded to the Disperse server, the framework can access and modify the iFrame code without violating same-origin policies.

The client page is also loaded with Disperse's two primary JavaScript libraries. The first of these libraries inspects the content of the iFrame, and hides any `disperse_screen` elements that do not match the screen number that is selected in the navigation header. We chose to hide (rather than remove) these elements in order to prevent potential errors from being thrown if a JavaScript function in the developer's code is looking for a removed element. The second of Disperse's JavaScript libraries searches the content of the iFrame for synchronized JavaScript functions. These synchronized functions are overridden with a new function that first runs the original function in the local iFrame, then pushes a message to the Disperse server with the client's connection profile, the name of the function that was called, and its parameters.

In turn, each connected client continually polls the Disperse server, checking for updates from other clients. When the server receives one of these updates, it is stored in Disperse's "Actions" database and delivered to all of the connected clients except for the one that originally produced it. The server also delivers updates about how many users are participating in the analysis session, which is reflected in the navigation header of all the clients. Each client then programmatically executes the prescribed JavaScript function to bring their iFrame view up to date. Because Disperse was designed specifically for conducting collaborative data analyses, there was no immediate need to optimize this update polling for high throughput situations. We envisioned a typical meeting consisting of around ten people, where the actions performed within the visualization take place at the pace of the discussion. Given that usage scenario, the current implementation has not encountered any scalability or message ordering difficulties. However, we are currently building a production-ready version of the framework that will likely adhere to an open connection architecture to better support higher throughput situations, should they arise.

In addition to the multi-screen capabilities, Disperse enables a whole host of features that would otherwise have to be implemented manually and independently by every visualization developer. For example, provenance is naturally kept during an analysis in the form of the functions that were called and the parameters that they took. This allows new users to be quickly "fast-forwarded" to the current state of the visualization. Users can also branch off into an independent exploration or return to previous steps. For example, if users notice a feature of potential interest in their view of the visualization, they can create a separate branch of the application that is not synchronized with the original branch. This is done using the branch ID field of the client's connection profile. If the user stumbles upon an interesting finding, the other users can move to this new branch and be brought up to date. Otherwise, if nothing of interest is found, the user can simply return to the original branch. This functionality is designed to mirror the Branch-Explore-Merge model described by MCGrath et al. [21]. The built-in usage tracking also means that, across multiple sessions, a complete history of the application's use is recorded, allowing for higher-level analyses of frequent/unused features as well as typical patterns of use. This data could guide the better overall design of interactive visualization tools or help to suggest potential next steps to users who are stuck in an analysis.

3.3 The Disperse Interface

For end users, deploying a Disperse visualization requires only opening browser windows and pointing them at the correct URL. Based on the URL parameters, Disperse visualizations can be opened in three different ways:

- A link to a specific screen of the visualization.
  (i.e. http://www.disperse.com/multi?id=ad762eb0&screen=5)
- A link to the visualization session, which defaults to the first screen.
  (i.e. http://www.disperse.com/multi?id=ad762eb0)
- A link to the single-screen version of the visualization, which displays the page without any of the Disperse functionality.
  (i.e. http://www.disperse.com/single?id=ad762eb0)

Once a visualization has been deployed through Disperse, users interact with the Disperse interface, which is designed to be nearly invisible to allow users to focus on the visualizations themselves. The interface consists of only a small navigation header in each browser window, as shown in Fig. 5. This header provides access to the framework's three primary features:

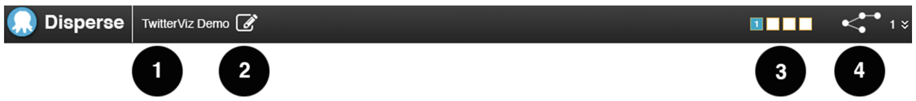


Fig. 5. The Disperse interface appears as a header in each browser window and includes (1) the name of the visualization that is being run, (2) a source code editor, (3) the screen layout, and (4) a branch selector (Color figure online).

Screen Layout. For every unique `disperse_screen` that developers specify in their CSS mark-ups, end users will see a unique box in the screen layout portion of the Disperse header. The view/screen that the user currently has displayed is highlighted in blue. Each box also lists the number of users that are logged into each view. Users can switch to a different view by simply clicking one of the other screen boxes in the header.

Branch Selection. The header displays the branch ID of each screen in a small, drop down menu. The initial/main branch is given an ID of 1. Users can create a new branch using the small, branch icon next to the drop down menu. When a new branch is created, it is silently added to the drop down menu, which allows users to complete work on their new branch without explicitly having to notify other users that the branch was created. Users can remove/terminate a branch only if they are the only user logged into that branch.

Source Editor. Much like the web testing tool, JSFiddle [19], Disperse allows the source code of an application to be edited within the framework itself (Fig. 6). This feature is primarily intended for developer use in finding and debugging JavaScript functions that have not been properly synchronized. However, it also allows more advanced end users to quickly reconfigure screen numbers in the CSS and turn on/off different function synchronizations. The source editor provides instructions for both of these mark-up requirements for developers to reference.

4 Case Studies

In order to understand the impact of Disperse on the visualization development and deployment process, we constructed five multi-screen and/or multi-user scenarios that were representative of Disperse's intended functionality (modeled after the evaluation in [14]). The first two of these scenarios were performed by two participants who were familiar with web development, but not with Disperse. Our goal was to observe whether these developers could grasp the fundamental building blocks of creating a multi-view visualization using Disperse. The final three scenarios, since they required a more substantial time commitment, were implemented by members of the Disperse design team. Each team member took a turn serving as the lead developer, while the other members observed the coding process and recorded any roadblocks or difficulties. All of the studies were conducted using the Google Chrome browser.

4.1 Splitting Static Content

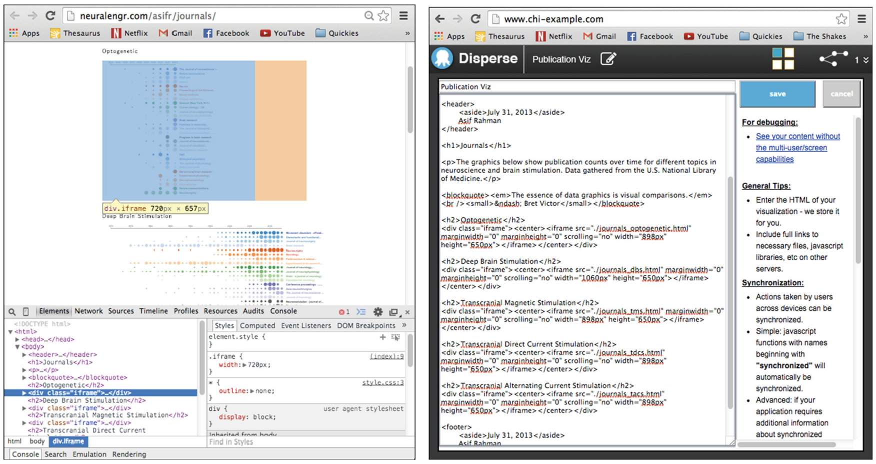
One of the works that inspired us was the WinCuts system by Tan et al. [36], which allows users to extract any content on their screen into a separate window. The benefit is that extraneous space between relevant content can be quickly eliminated. Our first goal was to simulate this basic functionality using Disperse. To do this, we selected

Asif Rahman's visualization of publication counts over time for different topics in neuroscience and brain stimulation, which is publicly listed in the D3.js example gallery [30]. This visualization consists of five static charts, each of which fills the full height of a standard laptop screen, meaning that users must scroll to see all of the content.

For this study, the developers were asked to use Disperse to display each chart in this visualization on a separate screen. This allows each chart to be seen concurrently, and for users to quickly switch between charts. The developers had not previously seen this visualization, and had no prior knowledge of its underlying structure. However, both developers immediately opened Chrome Developer Tools and isolated the `<div>` elements that contained each chart. The developers then added the `disperse_screen` class to each of these five elements (Fig. 6), one using the Disperse source editor and one using their text editor of choice. The source code was then uploaded to the Disperse server.

From there, the final step was to open a browser window on four additional screens, browse to the URL of the visualization on the Disperse server, and select one of the five screens to display. Both developers were able to accomplish this in just over a minute, about the same amount of time that it would have taken to open this visualization directly on all five screens and scroll each one to a unique chart. Accomplishing this using Disperse, however, has four advantages:

- This configuration is saved, so it can be quickly recreated.
- Users can switch views in a single click, instead of having to scroll.
- Errant scrolling does not perturb the display since the other views are hidden.
- The screen numbering provides an index of which chart is being viewed.



```
<!DOCTYPE html>
▼<html>
  ▶<head>…</head>
  ▼<body>
    ▶<header>…</header>
      <h1>Journals</h1>
    ▶<p>…</p>
    ▶<blockquote>…</blockquote>
      <h2>Optogenetic</h2>
    ▶<div class="iframe">…</div>
      <h2>Deep Brain Stimulation</h2>
    ▶<div class="iframe">…</div>
      <h2>Transcranial Magnetic Stimulation</h2>
    ▶<div class="iframe">…</div>
      <h2>Transcranial Direct Current
```

```
element.style {
}
.iframe {                         (index):9
  width:▶720px;
}
* {                               style.css:3
  outline:▶none;
}
div {                 user agent stylesheet
  display: block;
}
Inherited from body
```

Publication Viz

```
<header>
    <aside>July 31, 2013</aside>
    Asif Rahman
</header>

<h1>Journals</h1>

<p>The graphics below show publication counts over time for different topics in
neuroscience and brain stimulation. Data gathered from the U.S. National Library
of Medicine.</p>

<blockquote><em>The essence of data graphics is visual comparisons.</em>
<br /><small>&ndash; Bret Victor</small></blockquote>

<h2>Optogenetic</h2>
<div class="iframe"><center><iframe src="./journals_optogenetic.html"
marginwidth="0" marginheight="0" scrolling="no" width="898px"
height="650px"></iframe></center></div>

<h2>Deep Brain Stimulation</h2>
<div class="iframe"><center><iframe src="./journals_dbs.html" marginwidth="0"
marginheight="0" scrolling="no" width="1060px" height="650px"></iframe>
</center></div>

<h2>Transcranial Magnetic Stimulation</h2>
<div class="iframe"><center><iframe src="./journals_tms.html" marginwidth="0"
marginheight="0" scrolling="no" width="898px" height="650px"></iframe>
</center></div>

<h2>Transcranial Direct Current Stimulation</h2>
<div class="iframe"><center><iframe src="./journals_tdcs.html"
marginwidth="0" marginheight="0" scrolling="no" width="898px"
height="650px"></iframe></center></div>

<h2>Transcranial Alternating Current Stimulation</h2>
<div class="iframe"><center><iframe src="./journals_tacs.html"
marginwidth="0" marginheight="0" scrolling="no" width="898px"
height="650px"></iframe></center></div>

<footer>
    <aside>July 31, 2013</aside>
    Asif Rahman
```

For debugging:

- See your content without the multi-user/screen capabilities

General Tips:

- Enter the HTML of your visualization - we store it for you.
- Include full links to necessary files, javascript libraries, etc on other servers.

Synchronization:

- Actions taken by users across devices can be synchronized.
- Simple: javascript functions with names beginning with **"synchronized"** will automatically be synchronized.
- Advanced: if your application requires additional information about synchronized

Fig. 6. Even beginner-level web developers can use Chrome Developer Tools (left) to isolate the HTML that contains relevant content. Developers can then use the Disperse source editor (right) to add screen tags and function synchronization.

4.2 Basic Interaction Synchronization

For our second case study, we wanted to capture the experience of two remote users interacting with a single view. Again, the developers were tasked with augmenting an existing D3.js example. In this case, we chose the Zoomable Map (shown in Fig. 7), which allows users to click a U.S. map in order to zoom in on a particular state [40]. To enhance this example with synchronized, remote interaction, the developers both copied the Zoomable Map source code into the Disperse source editor. First, the page element that contained the actual map was given the `disperse_screen` class. Second, the developers located the JavaScript function that was enacted when a user clicked the map, and augmented this function name with the `synchronize` identifier. These two simple edits, which took both developers only a couple minutes to complete, now allowed any number of users to synchronously interact with this visualization. For example, two users in separate cities could both log into this view. If one user clicked to zoom in on a particular state, that action would also affect the display that the second user saw. The second user could then zoom back out, and again, that action would be propagated to the view of the first user.

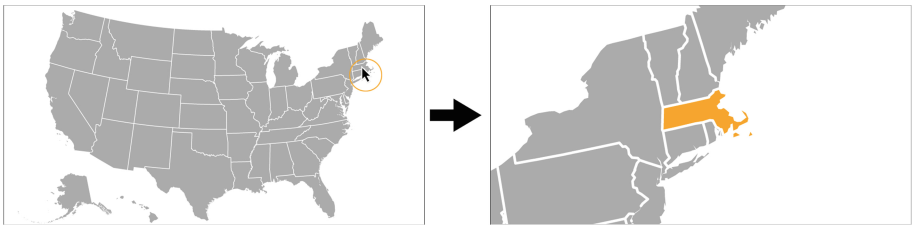
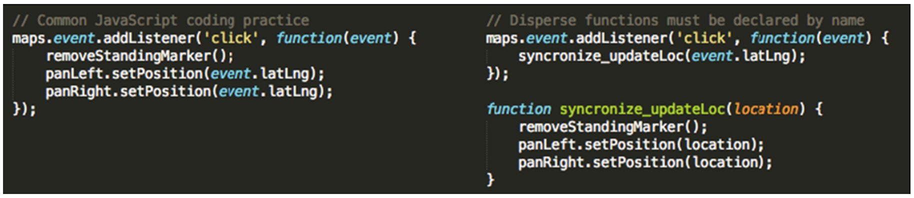


Fig. 7. The Zoomable Map allows users to zoom in on a particular state.

One consequence of the function synchronization requirement is that functions must be defined by name. A common practice in JavaScript is to pass anonymous functions as parameters. However, since Disperse relies on function names to identify the actions to synchronize, these functions must be defined outside of the parameter field and referenced by name (see Fig. 8). Both of the developers were informed of this requirement upfront and, as a result, neither had any difficulty adhering to this practice.



```
// Common JavaScript coding practice
maps.event.addListener('click', function(event) {
    removeStandingMarker();
    panLeft.setPosition(event.latLng);
    panRight.setPosition(event.latLng);
});
```

```
// Disperse functions must be declared by name
maps.event.addListener('click', function(event) {
    syncronize_updateLoc(event.latLng);
});

function syncronize_updateLoc(location) {
    removeStandingMarker();
    panLeft.setPosition(location);
    panRight.setPosition(location);
}
```

Fig. 8. Synchronized functions must be declared and referenced by name.

4.3 Modifying Existing Visualizations

One of the applications that motivated the development of Disperse was a web-based visualization dashboard called Social Pulse. Social Pulse performs text and sentiment analysis on the internal social media posts of our large, international organization. Its goal is to provide our human resources (HR) department with a better sense of what employees are interested in and passionate about, as determined from internal and external social media.

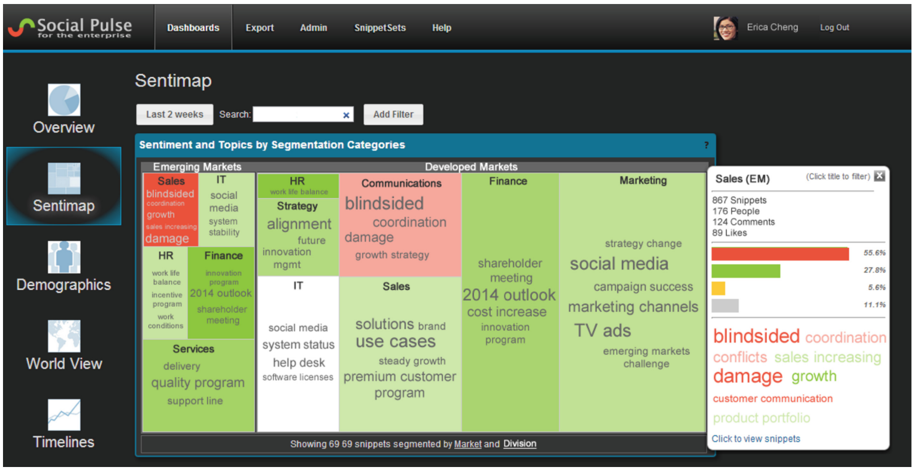


Fig. 9. Social Pulse allows uses to navigate to different views of the data using the icons on the left. However, in order to see two views side-by-side, users had to open the application in two windows, and manually set the appropriate filters in both.

The navigation-based interface provides multiple views, including a global map, colored by the average sentiment of posts from each region (choropleth map), and a treemap, which can break down post topics by any employee demographic (such as business unit and rank within the company). Users can select from a detailed array of filtering options, such as posts from a certain time period or containing an important keyword (see Fig. 9). These filters persist as users navigate to other views. However, two aspects of Social Pulse have proved useful for analysis, but cumbersome for users to accomplish within the tool:

- Seeing different views (constrained by the same filters) side by side. For example, users frequently would like to see a particular topic both geographically (using the choropleth map) and by business unit (using the treemap).
- Seeing the same view side by side with the same filters, but different time periods. For example, users would like to compare the topics from today to the topics from yesterday.

Because the Social Pulse filters are passed from view to view as URL parameters, the developer was tasked with creating these two additional capabilities without modifying the Social Pulse code base. This was accomplished by creating a new web page with multiple iFrames pointing to the different views of Social Pulse. Each of these iFrames was given a different `disperse_screen` class. The page included a synchronized JavaScript function that monitored the URL of each iFrame for changes to the filter parameters. When the function detected a change to the filters in one iFrame, it would update the URLs of the other iFrames to reflect the change. The developer also created a second version of this page in which changes to the time filter were ignored, allowing for users to see the same view at different timestamps with all of the other filters still synchronized.

Overall, a single day of coding yielded two much-needed extensions to a long-standing web application. Social Pulse can now be run as a multi-screen application either in a single-room environment or from opposite sides of the world, allowing our HR users to synchronously and collaboratively explore this data in new ways.

4.4 Creating New Visualizations

Visualization developers have long been aware of the fact that different visualizations better support certain questions over the same data [20]. New research also shows that a user's personality traits can affect the performance of using certain visualizations [26]. These works inspired our fourth implementation, which provides three different views over the same hierarchical dataset (all constructed in D3.js): a graph, a sunburst, and a treemap. Selecting an element in one view propagates the selection to the other views. This implementation is designed to be representative of the brushing and linking that is typically found across the views of a visualization.

From a development perspective, this implementation combined the approaches of our first two case studies. The developer used Disperse's tab-based template, so the `<div>` elements that comprised separate screens were already tagged with the `disperse_screen` class. The implementation included three synchronized functions, one to select an element of the hierarchy, one to deselect the hierarchy, and one to show the details of the selected element. Overall, the developer reported that it was not disruptive to plan function synchronization into the normal development process. Propagating changes between views inherently required special attention during development. The "synchronize" naming convention only made this thought process explicit. Additionally, the developer commented that this naming convention *helped* with general debugging because it made these critical functions extremely easy to locate through search.

In addition to observing the development process, this case study was also used observe the ease and flexibility of deploying a Disperse visualization. Our 3-view, hierarchical visualization was deployed in environments ranging from a trio of iPhones, to a wall-sized display, to combinations of the two (Fig. 10). Remote users who had been e-mailed a link to the application could immediately navigate to the URL and enact changes that could be seen by all users. Ultimately, the end user experience of deploying a Disperse visualization is virtually identical to deploying a normal web application.

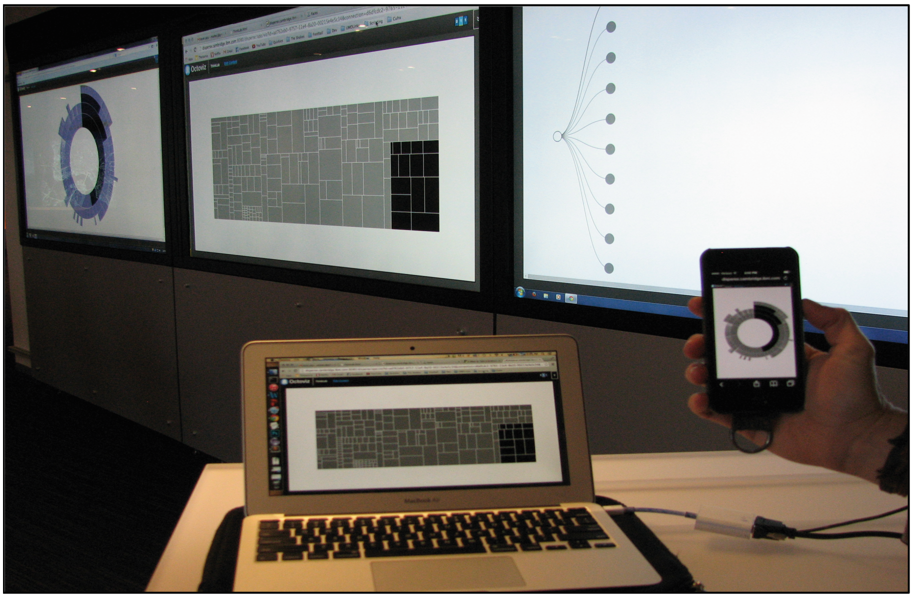


Fig. 10. Our 3-view, hierarchical visualization is deployed across five screens on three devices. First, all three views are tiled across a wall-sized display. Next, a user logs into the treemap view from a laptop. Finally, a user logs into the sunburst view from a phone browser. Now, any user can click on a component of their view and see the selection propagate to other screens.

4.5 Beyond Visualization

While Disperse was designed for the specific purpose of creating multi-screen visualizations, the framework can be used to create any sort of multi-screen web application. For our final implementation, we wanted to create an immersive experience for house hunting. The developer was asked to create an application that allowed prospective homebuyers to take a virtual walking tour of potential neighborhoods, all while maintaining their perspective of where they are on a broader scale. To do this, the developer created a three-view application that pairs a Google Map View with opposing Google Street Views (see Fig. 11). Using the list of potential properties on the Map View, the user can focus in on any property, which updates the opposing Street Views to show the property location. From there, users can use the Map View or either of the Street Views to "walk" around the neighborhood. As they do this, they can track their progress away from the original property on the Map View and see their surroundings get updated on the opposing Street Views.

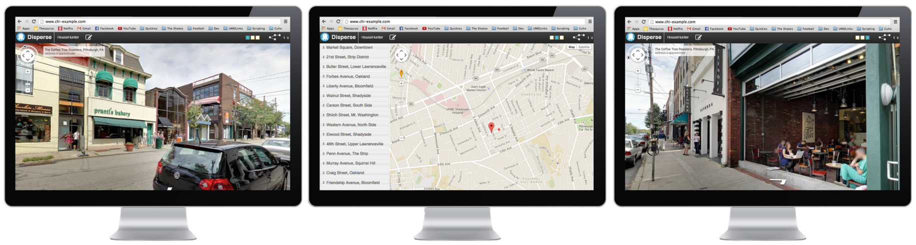


Fig. 11. Our house hunting application pairs two opposing Google Street Views with a Google Map View in the center.

Again, the house hunting application was built using Disperse's tab-based template, so the developer did not need to worry about designating the `<div>` elements that comprised separate screens. However, the developer was responsible for synchronizing the appropriate JavaScript functions. This process revealed two new considerations that resulted in updates to the Disperse framework.

Embedded Synchronization. Due to the nature of the Google Street View event listeners, it was impossible to distinguish between a user generated update to the location and a programmatic update to the location. Because of this, the developer arrived at impasse where a synchronized function needed to call another synchronized function. For obvious reasons, this would set off an infinite loop of synchronizations. To account for this, Disperse was updated to detect these embedded synchronizations, and suppress their propagation across views.

Object Serialization. Disperse synchronizes views by passing JavaScript functions the same parameters that were passed by the view that initiated the update. However, JavaScript does not serialize complex objects, which is a known problem throughout the language. To account for this, Disperse needed to be updated to manually serialize objects as they pass from the client to the server. Fortunately, serialization issues are so pervasive throughout JavaScript that custom serializers have been written for many complex objects. For example, the Google Maps location object that was being used in our house hunting application already had a serializer written for it. However, handling certain complex objects that do not have an existing serializer, such as objects with recursive functions, is still an open issue.

5 Limitations and Future Work

The case studies presented above are intended to serve as a sample of potential applications that can currently be built using Disperse. However, additional work is needed to ensure that the framework can handle a more complete spectrum of multi-view visualizations. For example, we encountered a D3.js application that relied on the location of the mouse on the screen to accomplish certain animations. These parameters were so deeply embedded in the D3.js code that it was not immediately clear how to surface them for synchronization. Further work is needed to understand the benefits and limitations of Disperse across a wider range of interactive visualization environments.

Still, this work demonstrates the ability of Disperse to take advantage of multi-screen environments across a wide range of tasks. Given the positive reception of our five case study implementations across the organization, it is clear that Disperse provides a capability that is needed by users of various levels of technical expertise. To this end, our next step is to build an interface that allows less technical users to create multi-screen environments. For example, in our first case study, the developer used the Chrome Developer Tools to identify `<div>` elements that would comprise separate screens. We would like to allow less technical users to identify these elements by simply hovering over them on the page and entering the appropriate screen number. Furthermore, we would like to provide more assistance in identifying functions that need to be synchronized. Once Disperse knows which components of a web page constitute separate views, we would like to automatically generate a list of JavaScript functions that will need to be synchronized.

6 Conclusion

In this paper, we presented Disperse, a web-based framework for splitting multi-view visualizations onto multiple screens. Disperse is designed to impose minimal time and complexity overhead on visualization developers, and eliminate hardware dependencies. Furthermore, the natural functioning of the framework allows for provenance to be kept without additional programming, and for users to both branch and merge their analyses using version control. We demonstrated the use of the tool through five case studies, which highlight the role of the developer in the creation of multi-screen, collaborative visualizations applications.

More broadly, Disperse allows multi-screen visualizations to be built and deployed without drastically deviating from the process of building and deploying a normal web application. Developers build the application, and users deploy it by navigating to the appropriate URL. There does not need to be any intermediary coordination between the hardware and the software, as is the case for many current DUI environments. Additionally, any user with basic knowledge of HTML and JavaScript can use Disperse to create multi-screen environments involving basic interaction. Our future work includes making this process even more accessible for less technical users.

Finally, while Disperse was designed expressly for the purpose of creating multi-screen visualizations, we have barely scratched the surface of its ability to create multi-screen environments in general. For example, the rise of online tools for presentation-making such as Prezi [27] and Reveal.js [31] offer an opportunity for using Disperse to create multi-screen presentations. Disperse is a first step towards providing access to screen space on a broader scale, using whatever hardware is available. This, in turn, will allow us to better understand the use of these environments and effects they have on collaborative decision-making processes.

References

1. Andrews, C., Endert, A., Yost, B., North, C.: Information visualization on large, high-resolution displays: issues, challenges, and opportunities. Inf. Vis. Spec. Issue State Field N. Res. Dir. **10**, 341–355 (2011)
2. Badam, S.K., Elmqvist, N.: PolyChrome: a cross-device framework for collaborative web visualization. In: Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces, pp. 109–118 (2014)
3. Baldonado, M.Q.W., Woodruff, A., Kuchinsky, A.: Guidelines for using multiple views in information visualization. In: Proceedings of the Working Conference on Advanced Visual Interfaces (AVI 2000), pp. 110–119 (2000)
4. Bernard, J., Wilhelm, N., Krüger, B., May, T., Schreck, T., Kohlhammer, J.: Motionexplorer: exploratory search in human motion capture data based on hierarchical aggregation. IEEE Trans. Visual Comput. Graphics **19**, 2257–2266 (2013)
5. Calderon, R., Blackstock, M., Lea, R., Fels, S., de Oliveira Bueno, A., Anacleto, J.: Red: a framework for prototyping multi-display applications using web technologies. In: Proceedings of the ACM International Symposium on Pervasive Displays (2014)
6. Carter, S.: Four ways to slice Obama's 2013 budget proposal. http://www.nytimes.com/interactive/2012/02/13/us/politics/2013-budget-proposal-graphic.html. Accessed 17 September 2014
7. Chung, H., North, C., Self, J.Z., Chu, S., Quek, F.: Visporter: facilitating information sharing for collaborative sensemaking on multiple displays. Pers. Ubiquit. Comput. **18**, 1169–1186 (2014)
8. Chung, H., Yang, S., Massjouni, N., Christopher Andrews, R.K., North, C.: Vizcept: Supporting synchronous collaboration for constructing visualizations in intelligence analysis. In: Proceedings of IEEE Symposium on Visual Analytics Science and Technology, pp. 107–114 (2010)
9. D3.js: Data Driven Documents. http://d3js.org/. Accessed 20 September 2014
10. Esenther, A.W.: Instant co-browsing: lightweight real-time collaborative web browsing. In: Proceedings of the World Wide Web Conference, pp. 107–114 (2002)
11. Fox, A., Johanson, B., Hanrahan, P., Winograd, T.: Integrating information appliances into an interactive workspace. IEEE Comput. Graphics Appl. **20**, 54–65 (2000)
12. Guerra-Gómez, J.A., Pack, M.L., Plaisant, C., Shneiderman, B.: Visualizing changes over time in datasets using dynamic hierarchies. IEEE Trans. Visual Comput. Graphics **19**, 2566–2575 (2013)
13. Han, R., Perret, V., Naghshineh, M.: WebSplitter: a unified XML framework for multi-device collaborative web browsing. In: Proceedings of ACM Conference on Computer Supported Cooperative Work, pp. 221–230 (2000)
14. Hartmann, B., Beaudouin-lafon, M., Mackay, W.E.: Hydrascope: creating multi-surface meta-applications through view synchronization and input multiplexing. In: Proceedings of the 2nd ACM International Symposium on Pervasive Displays (PerDis 2013), pp. 43–48 (2013)
15. Healey, C.G., Shankar, R.S.: Sentiment Viz: Tweet sentiment visualization. http://www.csc.ncsu.edu/faculty/healey/tweet_viz/tweet_app/. Accessed 17 September 2014
16. Huang, E.M., Mynatt, E.D., Trimble, J.P.: Displays in the wild: understanding the dynamics and evolution of a display ecology. In: Fishkin, K.P., Schiele, B., Nixon, P., Quigley, A. (eds.) PERVASIVE 2006. LNCS, vol. 3968, pp. 321–336. Springer, Heidelberg (2006)

17. Jettera, H.C., Michael Zöllnera, J.G., Reiterera, H.: Design and implementation of post-wimp distributed user interfaces with zoil. Int. J. Hum. Comput. Interact. **28**, 737–747 (2012)
18. jQuery. http://jquery.com/. Accessed 22 September 2014
19. JSFiddle. http://jsfiddle.net/. Accessed 17 September 2014
20. Kobsa, A.: User experiments with tree systems. In: IEEE Symposium on Information Visualization (INFOVIS 2004), pp. 9–16 (2004)
21. McGrath, W., Bowman, B., McCallum, D., Ramos, J.D.H., Elmqvist, N., Irani, P.: Branch-explore-merge: facilitating real-time revision control in collaborative visual exploration. In: Proceedings of the 2012 ACM International Conference on Interactive Tabletops and Surfaces (ITS 2012), pp. 235–244 (2012)
22. Miller, J.: Twitter Viz. http://www.twitterviz.com. Accessed 17 September 2014
23. Mühlbacher, T., Piringer, H.: A partition-based framework for building and validating regression models. IEEE Trans. Visual Comput. Graphics **19**, 1962–1971 (2013)
24. Nebeling, M., Mintsi, T., Husmann, M., Norrie, M.: Interactive development of cross-device user interfaces. In: Proceedings of the ACM Conference on Human Factors in Computing Systems (2014)
25. Node.js. http://nodejs.org/. Accessed 22 September 2014
26. Ottley, A., Yang, H., Chang, R.: Personality as a predictor of user strategy: how locus of control affects search strategies on tree visualizations. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2015), To appear (2015)
27. Prezi. http://prezi.com/. Accessed 22 September 2014
28. Processing.js. http://processingjs.org/. Accessed 20 September 2014
29. QlikView. http://www.qlik.com/. Accessed 17 September 2014
30. Rahman, A.: Publications in journals over time. http://neuralengr.com/asifr/journals/. Accessed 20 September 2014
31. Reveal.js. http://lab.hakim.se/reveal-js/. Accessed 22 September 2014
32. TIBCO Spotfire. http://spotfire.tibco.com/. Accessed 17 September 2014
33. Stolte, C., Hanrahan, P.: Polaris: a system for query, analysis and visualization of multi-dimensional relational databases. In: Proceedings of the IEEE Symposium on Information Visualization (INFOVIS 2000), vol. 8, pp. 5–19 (2000)
34. Streitz, N.A., Geibler, J., Holmer, T., Konomi, S., Müller-Tomfelde, C., Reischl, W., Rexroth, P., Seitz, P., Steinmetz, R.: I-land: an interactive landscape for creativity and innovation. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 1999), pp. 120–127 (1999)
35. Tableau Software. http://www.tableausoftware.com/. Accessed 17 September 2014
36. Tan, D.S., Meyers, B., Czerwinski, M.: Wincuts: manipulating arbitrary window regions for more effective use of screen space. In: CHI 2004 Extended Abstracts on Human Factors in Computing Systems, pp. 1525–1528 (2004)
37. Three.js. http://threejs.org/. Accessed 20 September 2014
38. Vogt, K., Bradel, L., Andrews, C., North, C., Endert, A., Hutchings, D.: Co-located collaborative sensemaking on a large high-resolution display with multiple input devices. In: Campos, P., Graham, N., Jorge, J., Nunes, N., Palanque, P., Winckler, M. (eds.) INTERACT 2011, Part II. LNCS, vol. 6947, pp. 589–604. Springer, Heidelberg (2011)
39. Wiltse, H., Nichols, J.: PlayByPlay: collaborative web browsing for desktop and mobile devices. In: Proceedings of the ACM Conference on Human Factors in Computing Systems, pp. 1781–1790 (2009)
40. Zoomable Map. http://bl.ocks.org/mbostock/2206590. Accessed 20 September 2014