VisDock: A Toolkit for Cross-Cutting Interactions in Visualization

Jungu Choi, Deok Gun Park, Yuet Ling Wong, Eli Fisher, and Niklas Elmqvist, *Senior Member, IEEE*

Abstract—Standard user applications provide a range of cross-cutting interaction techniques that are common to virtually all such tools: selection, filtering, navigation, layer management, and cut-and-paste. We present VISDOCK, a JavaScript mixin library that provides a core set of these cross-cutting interaction techniques for visualization, including selection (lasso, paths, shape selection, etc), layer management (visibility, transparency, set operations, etc), navigation (pan, zoom, overview, magnifying lenses, etc), and annotation (point-based, region-based, data-space based, etc). To showcase the utility of the library, we have released it as Open Source and integrated it with a large number of existing web-based visualizations. Furthermore, we have evaluated VisDock using qualitative studies with both developers utilizing the toolkit to build new web-based visualizations, as well as with end-users utilizing it to explore movie ratings data. Results from these studies highlight the usability and effectiveness of the toolkit from both developer and end-user perspectives.

Index Terms—Visualization system and toolkit design, interaction design, user interface, qualitative evaluation.

1 INTRODUCTION

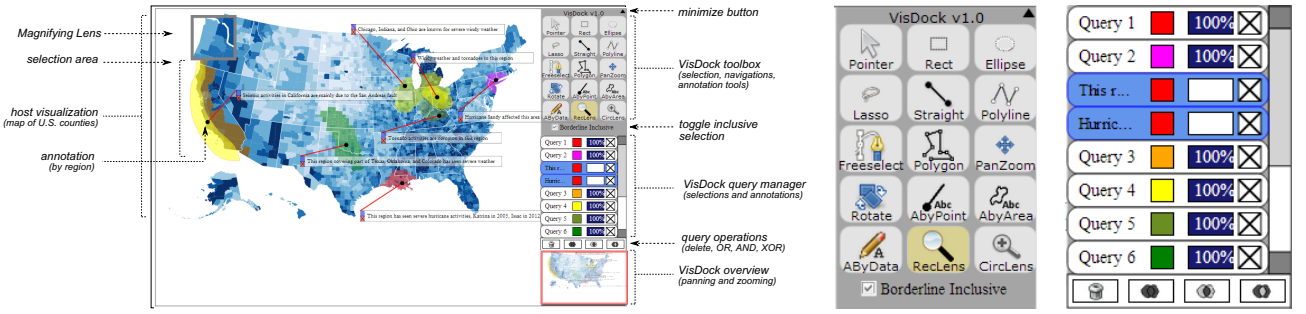
GRAPHICAL interfaces have become the norm rather than the exception for personal computing in the last two decades. Today's user environments, be they based on normal computers or mobile devices, are characterized by high-resolution displays, rich graphics, smooth animation, instant visual feedback, and complex transitions. Highly graphical applications, such as Adobe Photoshop, Microsoft PowerPoint, and CorelDraw, have evolved together with the user environments and now provide a rich and standardized set of *cross-cutting interaction techniques* for selection (by shape, path, or free-hand), manipulation (filtering, combining, masking), and navigation (pan, zoom, overview) in visual documents. While visualization applications are in many ways different compared to such general applications, they are also similar: they generally deal with managing (selecting, filtering, drilling down into) graphically-rich objects in multiscale visual spaces larger than the screen. However, while research has identified and isolated several abstract interactions common to visualization applications [1], there exist no standardized implementations for these tasks. In other words, there is much that visualization applications can learn from general applications in terms of interaction design and user experience. More formally, we define this concept as applied to visualization as follows:

> A *cross-cutting interaction for visualization* is an interaction technique that is common to a range of visual representations, data, and tasks.

- *Jungu Choi and Yuetling Wong are with Purdue University, West Lafayette, IN. E-mail: {choi88, wong64}@purdue.edu*
- *Deok Gun Park and Niklas Elmqvist are with University of Maryland, College Park, MD. E-mail: {intuinno, elm}@umd.edu*
- *Eli Fisher is with Microsoft Corporation, Redmond, WA. E-mail: fisher55@purdue.edu*

The purpose of this paper is to explore and operationalize such cross-cutting interactions for visualization. More specifically, our work concerns interaction techniques for selection [3] (lasso, feathering, shape selection, etc), navigation [4], [5] (pan, zoom, scroll, etc), layer management [3], [6] (visibility, depth order, transparency, merging, etc), and annotation [7] (by point or by region) in visualization. While many of these interaction techniques already have been applied to visualization, such work has been performed on a case-by-case basis. Our effort here is unified and comprehensive, and proposes a wider focus on user experience for visualization than traditionally has been done outside commercial settings. To facilitate broad adoption of best-practice interaction techniques in visualization, we also present an SVG-based JavaScript library called VISDOCK that provides implementations of all of these interactions that can be easily used by any web-based visualization implementation. VisDock consists of a graphical user interface toolkit and an event handler that connects predefined interaction tools to events that are defined by the visualization creators. Therefore, VisDock provides cross-cutting interactions to any SVG-based visualizations while giving developers flexibility for how to handle the response to these interactions in their particular visualization. The intended primary users of VisDock are thus developers, who in turn build web-based visualizations that are used by end-users exploring data using them.

To showcase the utility of VisDock, we have made the library publicly available as Open Source along with a large number of examples. We have also performed qualitative usability studies both involving four developers in an independent research group, as well as with 11 end-users taken from our university. The developers were asked to use VisDock to augment their existing web-based visualization projects, whereas the end-users explored movie ratings and business review data using a VisDock-enabled web-based



(a) U.S. unemployment visualization with VisDock. (b) VisDock toolbox. (c) Query manager.

Fig. 1. The VisDock toolkit integrated with a U.S. map built with D3 [2] to show unemployment in 2008. With a minimum of additional coding, VisDock provides advanced selection, query management, navigation, and annotation functionality for the existing map visualization. The VisDock interface (right side of the image) is inspired by graphical editors such as Adobe Photoshop and Illustrator.

visualization we developed for this purpose. Results show that both groups found the toolkit usable and efficient; the developers were all able to integrate the toolkit with their existing work with minor difficulty, and end-users made frequent and varied use of all of the cross-cutting interactions provided by the toolkit.

2 Background

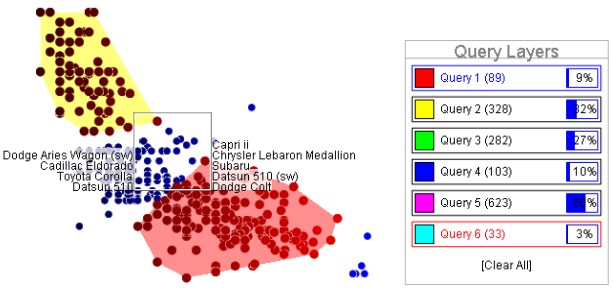
Based on the interaction taxonomy for visualization given by Yi et al. [1], we here review relevant work on selection, queries, navigation, and annotation for visualization. We also contrast each of these existing contributions with our proposed new VisDock library.

2.1 Selection

Selection for visualization is the ability to mark something as interesting [1], and is a very common operation when using a visualization. Selecting an item (or items) will usually change its visual representation, such as its fill or outline color, and then allows further manipulation on the item, such as zooming in, filtering out, or drilling down. For example, when viewing a large dataset, selecting one or several data points that are of interest allows the user to more easily focus on those particular points. However, the selection operation in most visualizations today is often limited in one or several ways: (a) users can often only select single items (by clicking on them) or use simple bounding box selection to select multiple items, and (b) many visualization systems do not allow for managing more than one selection at a time, even if the selection can contain any number of items.

It should be noted that our definition of selection encompasses only interactions designed as direct manipulation [8] techniques, whereas many visualization applications instead choose to focus on *filtering* [1] items, which can be seen as a form of selection. We disregard filtering here due to it often being a more data-driven and indirect operation conducted using range sliders or number fields.

Furthermore, due to its importance for visual tools, there exist several visualization systems that have implemented powerful selection mechanisms. For example, commercial tools such as Spotfire [9] and Tableau/Polaris [10] do include a range of direct selection tools. In addition, the ScatterDice [3] and GraphDice [11] systems support lasso selection (Figure 2(a)). However, perhaps the most advanced example is the graph selection techniques proposed by McGuffin and Jurisica [12], which interprets lasso or rectangle selection depending on the user's stroke. The techniques are also structurally aware, allowing selections based on the graph topology. The selection techniques in VisDock are inspired by this work, but aim to provide this functionality to a wide variety of data beyond graphs.



(a) Lasso selection + excentric labeling. (b) Query layers.

Fig. 2. Selection and queries in ScatterDice [3], [11].

2.2 Query Management

Once a set of items has been identified using selection, it is often convenient to be able to save the selection as a query result set, compare the set against other sets, and combine them in various ways (often using set operations such as union, intersection, and set difference). The benefit that the user gains from this type of interaction is the ability to add meta-data onto a visualization beyond the typically static dataset. Essentially a generalization of selection, we call this concept *query management*, or, as is often the case, visual query management, since many tools use visual representations for their query results.

Several existing visualization systems provide query management in various forms. Heer et al. [13] applied

query relaxation techniques to allow users to generalize selections according to different attributes in information visualizations. ScatterDice [3] and GraphDice [11] support a type of faceted search refinement called query sculpting, and maintain query control windows for this purpose (Figure 2). The TimeMatrix [6] temporal graph visualization system uses a similar overlay box to manage visibility and transparency of different visualization layers. While all these existing offerings are powerful, they all represent one-time designs and cannot easily be applied to new visualizations. The purpose of query management in VisDock is to make advanced query management available for virtually all visualization applications.

2.3 Navigation

Many visualizations incorporate *navigation* techniques to allow the user to move the viewport and explore more of the dataset [1]. This is particularly important for large-scale and/or visually complex visualizations where the number and complexity of data items may exceed the size of the viewport or the perceptual and cognitive capabilities of the user. Most common of these navigation operations are panning and zooming [1], [4]—as a case in point, zooming is included in Shneiderman's information seeking mantra [14]—but more advanced navigation is possible.

Since complex visual representations are often multiscale spaces [15], many visualization tools by necessity go beyond simple zooming and panning. OrthoZoom [16] allows for fast yet controlled zooming in very deep visual spaces. Topology-aware navigation [5] and gravity navigation [17] both utilize knowledge of the underlying visual representation to ease navigation by guiding the user along paths and towards targets, respectively. MatrixZoom [18] use controlled animations to transport users between different levels of an adjacency matrix representation of a graph. PolyZoom [19] and Mélange [20] allow for navigating while maintaining multiple focus regions of a visual space visible at the same time. Similarly, the Dynamic Insets [21] technique draws cutouts of the local neighborhood of multiple off-screen targets to provide context-aware navigation.

Unfortunately, advanced navigation techniques are typically non-trivial to implement. Furthermore, the lack of standardized navigation means that users often have a hard time transferring knowledge from one visualization to the next. Again, the purpose of VisDock is to provide a unified framework that makes it easy both for developers to integrate advanced navigation techniques in their applications, as well as for users to navigate in them in a consistent way.

2.4 Annotation

With few exceptions, visualization applications tend to be designed as viewers of read-only data, so the only way to modify the visualization is to add meta-level *annotations* on top of the data layer. An annotation is defined as user-created meta-data associated with a visualization, view, or dataset. Examples of annotations could include a label, a paragraph of text, or a circled item with a scribbled note.

The design space of annotation is clearly very broad, from being a simple acetate layer on top of the visualization such as in Sense.us [7], to the post-it labels attached to nodes in the DataMeadow data flow system [22]. Some visualization systems provide entire visual languages for annotation: examples include the semi-structured Sandbox system for visual thinking [23], the shoebox view for evidence marshalling in Jigsaw [24], and the knowledge view visual markup language in the Aruvi system [25]. However, as evidenced by Sense.us [7], even very simple visualizations can benefit from annotation support (particularly for collaboration [26]), so the goal of VisDock is to provide such baseline annotation functionality.

2.5 Toolkits for Information Visualization

Building toolkits is a common activity in the information visualization community, and integrating advanced interaction and navigation techniques directly into such toolkits is a good way to achieve widespread adoption across a wide range of developers. Indeed, many traditional visualization toolkits, such as Prefuse [27], IVTK [28], and Improvise [29], do integrate an array of such complex interaction techniques, including mechanisms for selection, focus+context navigation, bird's eye views, etc.

However, with the rise of the web browser as the premier platform for visualization, a new set of JavaScript toolkits—such as D3 [2], ProcessingJS[1], and the JavaScript InfoVis Toolkit[2]—has arisen to replace the old ones. Furthermore, tools such as Lyra [30] and iVisDesigner [31] are even allowing end-users to interactively create new visualizations without any need for textual programming. Ellipsis [32] allows users to create dynamic annotations in their visualizations by direct manipulation interface. However, the design rationale of such toolkits tend to be more minimalistic, perhaps due to the dominantly layperson audience of web-based visualization, so none of these toolkits provide more than basic interaction techniques. This motivates our choice to design VisDock as a mixin JavaScript library that can easily integrate with existing visualization toolkits, similar to Harper and Agrawala's restyling tool [33] that extends any existing D3 visualization.

Very recent work on interaction for information visualization toolkits includes the declarative interaction language proposed by Satyanarayan et al. [34]. Unlike the callback-based event handler used in VisDock, this approach is based on the Vega JSON-based visualization grammar where visualizations (and interactions) are specified as data-level interactors that operate on streams of input data. Our contribution instead provides a concrete set of cross-cutting interaction techniques accessed using a tool dock.

3 Design: Cross-Cutting Interaction

The design space of cross-cutting interaction techniques for visualization includes interactions that can be applied to

1. http://processingjs.org/
2. http://thejit.org/

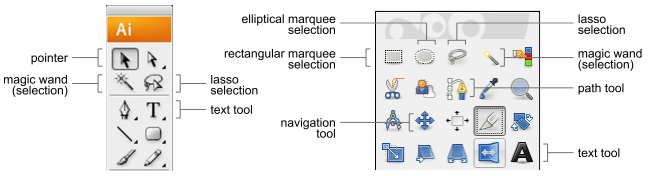


Fig. 3. Toolbars from Adobe Illustrator (left) and the GNU Image Manipulation Program (GIMP) (right).

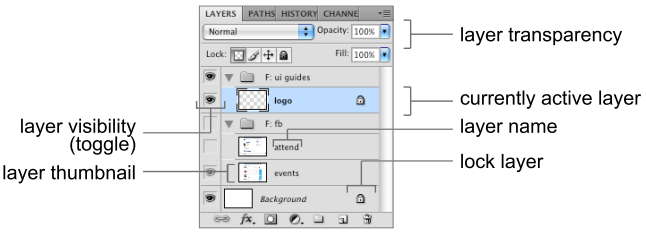


Fig. 4. Layer control window for Adobe Photoshop.

a wide range of visualizations for any type of data. For an interaction to be truly cross-cutting, however, it must be operationalized at a sufficiently concrete abstraction level that it can be implemented consistently across all datasets and visual representations. For example, the seven categories of interactions proposed by Yi et al. [1] describe high-level tasks based on user intent, and thus are too abstract to constitute actual cross-cutting interactions.

More specifically, we define the design space of cross-cutting interaction techniques for visualization as follows:

- **Any dataset:** The technique should not be tied to a data type. For example, the selection tools by McGuffin and Jurisica [12] are designed for graph data alone.
- **Any visualization:** The interaction should be useful for many different visual representations. For example, the link-sliding technique [5] is specific to node-link diagrams and will not directly transfer to scatterplots.
- **Concrete:** If an interaction technique is not specified at a sufficiently operationalized level, it will not be possible to build an implementation of the technique that works across multiple representations.
- **Direct manipulation:** We choose to consider only techniques based on *direct manipulation*, i.e., that operate on continuous visual representations of objects manipulated using physical actions that are rapid, incremental, and reversible [8].

To inform our design beyond the visualization field, we draw on examples of interaction and user experience design from general user applications. We review these below.

Selection techniques. Many graphical applications provide advanced selection techniques, including by lasso, paths, and shapes. Figure 3 shows toolbars for both Adobe Illustrator as well as the GIMP. Some of the tools depicted here include rectangular and elliptical marquee selection, the magic wand selection tool for selecting a contiguous area based on color, and the path tool which can be used as a selection input. All of these are cross-cutting in that they are highly concrete, they can be applied to virtually any collection of graphical objects, and use direct manipulation.

Layer management. The closest to query management tools for general graphical applications is the layer management for graphical editors such as Adobe Photoshop. Figure 4 shows the layer control window in Photoshop where graphical objects have been stratified into a layer stack which controls rendering order. Individual layers can be moved up and down in the stack, their visibility and transparency can be changed, and they can be given names. Photoshop and GIMP even treats text as a special layer type, which is of interest for our annotations—what if annotations were simply special query layers?

All of these are cross-cutting interaction techniques: any visual representation can be split into layers without loss of generality, and the layers can then be individually controlled. In fact, the IVTK toolkit [28] draws node-link diagrams as two layered visualizations, one for the links and one for the nodes, and TimeMatrix [6] composes multiple visual layers to produce its representation.

Navigation. Graphical editors typically operate on large, high-resolution images and illustrations, so navigation is a key operation in these tools. The typical navigation techniques in Adobe Photoshop, GIMP, and Adobe Illustrator are zooming as well as scrolling and panning (where the former is conducted using the window scroll bars and the latter using a hand tool). Recent versions of Photoshop are now starting to incorporate more advanced navigations, such as smooth zooming (using the GPU for high performance), flick panning (where the viewport will continue to glide for a bit after releasing a pan operation), and bird's eye views (where the user can temporarily zoom out to an overview before zooming back in to a new location).

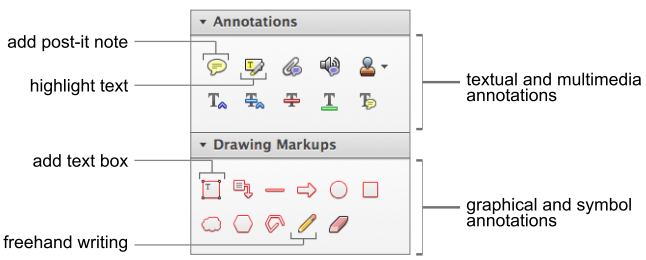


Fig. 5. Annotation toolbar for Adobe Acrobat Pro.

Text and annotation. Graphical editors such as Photoshop and the GIMP do not need annotation functionality because they are content creation tools by design (they do contain text tools for creating textual content, however). Visualizations, on the other hand, are generally designed as read-only viewers. For this reason, a better source of inspiration for cross-cutting annotation techniques is Adobe Acrobat Pro and Adobe Acrobat Reader. Even if the former is a content editor, the PDF format is not suited for editing, and thus both Acrobat applications are primarily designed for viewing PDF documents and not creating them. Figure 5 shows the annotations toolbar in Adobe Acrobat Pro. Some of the relevant interaction techniques shown here include highlighting, adding a post-it note or a text box, and adding

arrows and simple shapes to the document.

Additional interactions. Current graphical editors contain several additional interaction techniques that are candidates for adoption as cross-cutting interaction techniques for visualization. However, many of these are specific to content creation and modification, which is a poor fit for visualization. For example, cut-and-paste is perhaps the most well-known cross-cutting interaction technique common to virtually all modern user applications, but lacks a straightforward equivalent for read-only visual representations. Other techniques, such as search, replace, sort, etc, are less general or do not use direct manipulation.

One particularly interesting class of interactions is techniques based on *surrogate interaction* [35], where proxy objects are added to the interface to allow complex manipulation. Several visualization examples were proposed in the work. Furthermore, the interactive legends by Henry Riche et al. [36] is a similar idea. However, all of these techniques require significant customization for each visualization, so we currently exclude them from VisDock.

4 The VisDock Toolkit

VisDock is a JavaScript library for adding cross-cutting interaction techniques to web-based visualization. It currently supports advanced selection, query management, navigation, and annotation. The toolkit is based on SVG and uses a container approach for external visualization, allowing it to be used with any SVG-based toolkit such as D3 [2] or RaphaëlJS.[3] A minimalistic interface consisting of a toolbar, a query management window, and an overview provides access to the toolkit's functionality (Figure 1).

VisDock has two types of users: because it is a programming API, the primary user is the *client programmer* who is using the library to augment their web-based visualizations. The *end users* who are using VisDock to interact with a visualization are thus secondary users of the toolkit.

4.1 Basic Architecture

VisDock is designed as a mixin JavaScript library intended to augment an existing SVG-based visualization while requiring a minimum of coding. The visualization being augmented is called the *host visualization*. To achieve a minimal footprint on the host visualization, VisDock uses a container design. Instead of creating a new top-level SVG element (an `<svg>` tag), the host visualization creates a VisDock instance, which creates the element, and then requests the viewport from VisDock itself (Listing 1).

```
VisDock.init("container", {width: w, height: h});
var viewport = VisDock.getViewport();
```

Listing 1. Initializing VisDock to a specific DOM element and with a desired canvas size.

The container for VisDock can essentially be any element in the HTML document, such as the `<body>` or any `<div>` elements. The viewport node is an SVG group (`<g>`) element, which means that it can be populated with the host visualization's own graphical representation. If the developer wishes to import the VisDock toolkit into a web visualization consisting of multiple canvases (e.g. Figure 10), VisDock can be initialized with arrays of arguments as shown in Listing 2.

```
var container = ["#div1", "#div2"];
var sizes = [{width: w1, height: h1},
 {width: w2, height: h2}];
VisDock.init(container, sizes);
var viewport1 = VisDock.getViewport(0);
var viewport2 = VisDock.getViewport(1);
```

Listing 2. Initializing VisDock to accommodate multiple canvases.

VisDock creates additional scene graph nodes for internal use: the toolkit maintains an overall panel for the entire interface, including subtrees for the toolbox, query manager, and overview. Furthermore, ancestors to the viewport include nodes for clipping, annotation, and navigation.

4.2 Graphical User Interface

The core VisDock graphical interface (Figure 1) consists of the main viewport (which the host visualization fully controls), a toolbox, a query manager, and an overview. The entire interface can be hidden by clicking a minimize button at the top-right corner of the VisDock panel; clicking it again will bring it back. The purpose of this feature is to support situations where the full VisDock interface may become too intrusive and detract from the visual representation. The VisDock interface can also be dragged across the viewport, allowing for optimizing space usage, and can be docked when dragged to the corners or edges of the viewport. The docking orientation changes from vertical (Figures 6, 7, 8, and 9 to horizontal (Figure 12) when the dock is dragged to the top or bottom edges of the viewport.

Furthermore, VisDock is also responsive to varying device capabilities. which means that the toolkit adapts its size based on the amount of available screen and input space. For instance, on a smartphone or tablet, VisDock is initialized with the minimumized dock to give full view of the host visualization, whereas the dock is always visible when viewed on a computer screen. In addition, the VisDock toolkit is designed such that when the viewer resizes the window, the toolkit also adjusts its size so that it fits reasonably well in the canvas. This feature is inspired by Responsive Javascript[4] and is particularly useful when the original visualization is large.

Beyond this, all VisDock interface components can be activated and deactivated on an individual level, even down to specific operations. The intention is to give full control and flexibility to the client programmer who is using the library. For example, if the host visualization does not provide an event handler, the VisDock toolbox will not include selection tools. Similarly, if a particular visualization does not need a miniature overview, the client programmer can simply disable that feature with a single line of code.

3. http://raphaeljs.com/

4. http://www.responsivejavascript.com

The VisDock toolbox (see Figure 1(b)) is how end users of a VisDock-enabled visualization accesses its functionality. Clicking on a tool in the toolbox will make it the currently selected tool (indicated using a highlight). VisDock tools are organized into three groups: selection, navigation, and annotation tools (described below). In addition, VisDock provides a default Pointer Tool. When the Pointer Tool is active, all interaction in the viewport will be passed on to the host visualization. This allows for accessing the interactions provided by the host visualization, such as hovering over items to show data tooltips or dragging the nodes in a force-directed graph visualization.

Unlike the integrated pop-up widgets employed in a number of other systems [12], [37], VisDock uses a button-based toolbar. Pop-up widgets are often invoked by means of a mouse click or stylus gesture. We chose a toolbar design for VisDock to separate mouse/stylus events from the original interactivity of the host visualization, allowing for easy integration with new visualizations. Just like other multi-platform systems [3], [11], VisDock-enabled visualizations are compatible across various platforms including tablets and smartphones. This means users may have limited means of accessing events (such as mouse right click). Furthermore, the menu can be minimized if it becomes distracting or takes up too much space, as discussed above.

4.3 Event Handling

Due to VisDock's container design, the toolkit has no way of determining the structure of the embedded host visualization. Instead, the client programmer must supply the necessary visualization-specific logic that serves as the "glue" to the toolkit. In VisDock, this is achieved using the VisDock event handler: an abstract interface that all host visualizations must implement either fully or partially (depending on which cross-cutting interactions are desired). Listing 3 shows the interface[5] for VisDock's event handler.

```
var visdock.eventHandler = {

  // Find intersected items given a shape
  getHits: function(points, inc) { return []; },

  // Mark the selection with a color
  setColor: function(items, queryId) {},

  // Remove a selection
  removeColor: function(queryId) {},

  // Modify an existing selection color/opacity
  changeColor: function(style, queryId) {},
  changeVisibility: function(style, queryId) {},

  // Called when Pan/Zoom/Rotate events occur
  viewChanged: function(view) {}
}
```

Listing 3. Interface for the event callback handler.

To enable specific functionality, the client programmer is expected to create an implementation for all methods in the event handler (if no event handler is present, all VisDock operations are disabled). Because some operations such as shape intersection (for selection) can be challenging to implement, VisDock comes with a utilities library (`visdock.utils`) with helper functions, such as for intersection testing between most common shapes.

5. JavaScript is a dynamically typed language, so it does not explicitly support interfaces; the code only shows the spirit of the design.

4.4 Selection

VisDock selection tools include the following (Figure 1):

- **Shape selection** (e.g., rectangle, ellipse, polygon);
- **Free-hand lasso selection** for closed shapes; and
- **Path-based selection**, both free-hand and polylines.

Selection is invoked by selecting the appropriate tool in the VisDock toolbox and then interacting in the visualization viewport. Rubberband shapes are added to an overlay drawn on top of the host visualization in response to such interaction, and are removed when the operation ends. For example, using the Rectangle Tool lets users specify the position and dimensions of a rectangular marquee selection.

Most event handler operations (Listing 3) are used for handling selections: determining which marks (as opposed to background graphics) fall within a selection shape or path as well as adding, modifying, and removing color highlights in the visualization. The hit testing function (`getHits`) can take either open or closed shapes (depending on the `points` array), and also accepts a flag that indicates whether the intersection test should be inclusive or not. Inclusive intersection means that an item that only intersects but is not fully contained by a closed selection shape is still regarded as being selected. For non-inclusive intersection, items must be fully contained within the selection to be hit.

To alleviate the client programmer's burden, VisDock utilities library provides standard functions for selection, including intersection testing as well as maintaining color sets for highlighting queries. Typically, the functions in the utilities library create cloned layers of the original objects to mark the selection; these are later deleted when a particular item is deselected. More advanced behavior can be achieved using the arguments in Listing 3. The argument `points` array contains the coordinates of the outline of the bounding shape, the `items` array contains original elements that were selected for the query with the index `queryID`, the `style` argument contains the style information of the specific selection, and the `view` argument is the type of transformation, such as panning, zooming or rotating. Instead of following the typical example of selecting and cloning, the client programmer may make the selected elements shrink, expand or remove from visualization by defining his or her own event handler.

```
VisDock.updateLayers();
```

Listing 4. Updating the cloned layers.

VisDock is also able to handle selection of dynamic SVG elements that undergo transition in the host visualization. The transitions can be an event-triggered one, such as semantic zooming, or a continuous one, as for an animated object (Figure 9). In order to update the coordinates of these cloned layers in case of transition events, the update function can be regularly invoked as shown in Listing 4.

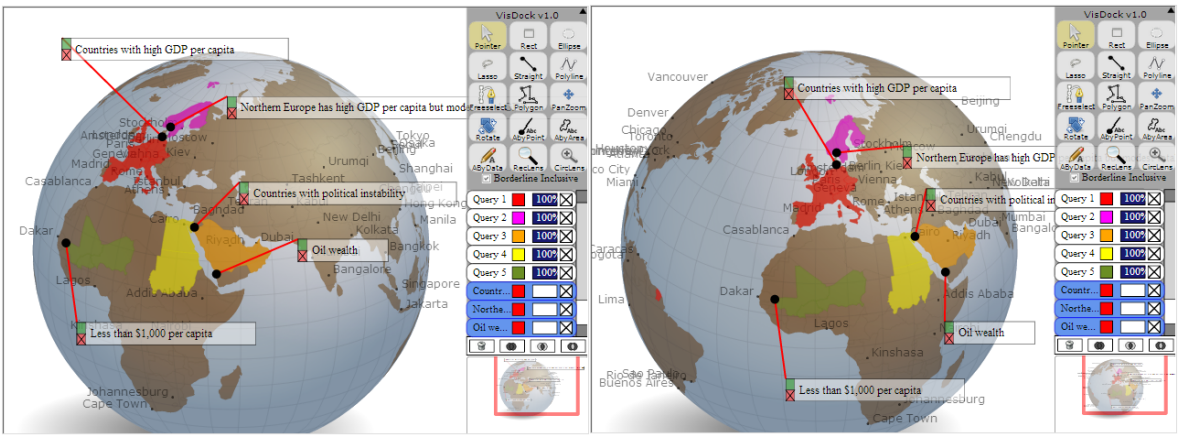


Fig. 6. VisDock-enabled faux-3D shaded Earth visualization with annotations by data space. These annotations stay attached to the original elements when the globe rotates.

4.5 Query Management

The VisDock query manager maintains a list of currently active queries. A query consists of a set of items (a subset of the entire dataset being visualized), a name, a color, and parameters for visibility, transparency, and depth order. Making a selection using the VisDock selection tools will automatically create a corresponding query for the selected items in the query manager, assigning it a default color and name (currently "Query N", with incrementing *N*).

The query management window is a listbox with vertical scrollbars that shows active queries (Figure 1(c)). Its design is inspired by the layer management interfaces from Adobe Photoshop and the GIMP. Each query is represented by a single line giving its name and color and providing widgets to show/hide, change transparency, and delete the query. Queries can be selected by clicking on them, changing their background color. Just like in Photoshop, the order of queries in the list governs their depth rendering order. The order can be modified by selecting a query and moving it up or down in the list. Similarly, changing the name of a query is done by clicking on its label (which will pop up a dialog box), and its color can be changed by clicking on the color box and selecting a new one in a color picker.

While maintaining a history of all selections is a useful feature for a query manager, the potential of this cross-cutting interaction technique is even greater. VisDock provides several set operations for combining the currently selected queries in various ways: AND computes an intersection between the item sets for all selected queries (i.e., including only those items that appear in all of them), OR computes a union (all items that exist in at least one query), and XOR computes the exclusive disjunction (all items that appear in only one query). Instead of deleting the operand queries (which is the logic of the analogous "merge layers" functionality in Photoshop), the results of set operations create a new query that is added to the end of the list.

Since all of our query management tools operate on anonymous sets of items, VisDock does not require the client programmer to write any glue code to support most of its operations. The only exception is that the event handler (Listing 3) must contain implementations for adding, modifying, and removing color sets in the host visualization; these methods are called in response to creating, changing, and deleting visual queries. This is made slightly more complex by the need to maintain multiple color sets, since a single item can feature in many queries. Again, the VisDock utility library provides helpful bookkeeping code and data structures for this purpose.

4.6 Navigation

Two facts contribute to enabling VisDock to support advanced navigation for any SVG-based visualization:

1) The viewport containing the host visualization is an SVG group (`<g>`) node, which means that it has an affine transform that affects the visualization; and
2) Except when the Pointer Tool is active, VisDock will intercept all inputs (such as mouse or touch presses, drags, and releases) performed on the viewport.

This means that we can provide navigation tools that will modify the viewport's transform in response to interaction. In the current VisDock implementation, we provide standard Pan, Zoom, and Rotate Tools which behave as expected: they change the translation, scaling, and rotation of the viewport's transform, respectively. While uncommon for visualization, the intention behind the Rotate Tool is to be able to change the orientation of a visual representation, such as when using it on a horizontal (i.e., tabletop) display.

```
var overview = visdock.getOverview();
```

Listing 5. Accessing the overview viewport.

Beyond these direct manipulation techniques, VisDock also provides an overview window for displaying a miniature version of the visualization. This is simply another SVG group node; Listing 5 shows how the client programmer can access this node using JavaScript. Instead of replicating the SVG code for the main visualization, the client programmer may choose to use a less detailed representation in the overview (even a bitmap image).

For all of these operations, the event handler provides a `viewChanged` method that gets called everytime the 2D view transform changes; a client visualization only needs to implement this method to respond to such viewport changes. Possible uses for this method is to support semantic zooming [38], where the visual representation changes depending on the magnification level, or to move the axes of a scatterplot to a fixed location inside the viewport as the user pans and zooms in the visualization.

VisDock also provides magnification lenses, in rectangular and circular shapes, that allow users to inspect the host visualization by focusing on a local region. A small rectangular or circular view will appear with a magnified view of the selected region. These features are shown in Figures 7, 8, 9, and 12 with a bounding shape with a grey stroke. The zoom levels of these lenses can be controlled by first activating the lens and then rotating the mouse wheel.

The above navigation operations are naturally common, and much more advanced and powerful techniques are possible for future extensions to VisDock. For example, pinching to zoom or rotate would be useful for when interacting with VisDock using a touch screen. While not currently supported, they may be implemented in the future.

4.7 Annotation

Annotations in VisDock use the same approach as Sense.us [7], where user annotations are simply represented by an overlay on top of the host visualization. This is akin to an acetate layer overlaid over a paper illustration, where drawing and handwriting on the acetate is independent of the underlying illustration, yet is ultimately composed into a single picture. Building on this metaphor, all annotations in VisDock become layers of their own and are added to the query management window. This gives access to the same operations that queries have, i.e., changing name (i.e., the annotation text), color (for reference points or regions), transparency, depth order, and deleting an annotation.

Annotations in VisDock are textual labels with a *reference line* and a *reference*. The mechanism is accessed using three separate tools: by point, by region, and by data. Annotations created by the first two methods have fixed reference points in the Euclidean space, while annotations by data let the user specify a single reference point and a single SVG element. Annotation by region lets the user draw a closed shape using a lasso to define a reference region. In all cases, the annotation label is attached to the reference using the reference line. The label can then be freely moved around the visualization to a place where it does not occlude important information. VisDock's annotations are navigation-aware, so they will stay fixed to the underlying visualization even when the user pans and zooms in the visual space (and are shown in the overview).

While annotation by point and annotation by region fix the reference at specific coordinates in the viewport, annotation by data space attaches the annotation to an SVG element. Therefore, if the underlying host visualization is interactive or changes over time, the original information contained in annotations with a fixed reference may become out of date. On the other hand, annotation by data gets updated to the appropriate location that reflects the change in the host visualization. Annotation by data space is showcased in the faux-3D shaded globe example[6] by Derek Watkins in Figure 6.

```
AnnotatedByData.layerTypes = ['.layers'];
```

Listing 6. Initialization for annotation By Data Space.

Implementing annotation by data space requires specifying the types of SVG elements to which annotations should be attached (Listing 6 specifies the SVG element types as elements with the 'layer' class identifier). These can be a SVG tag name, polygon, ellipse, rectangle, a class identifier, or a combination of these.

```
AnnotatedByData.update();
```

Listing 7. Updating annotation By Data Space.

Once the update event is called (Listing 7), all annotations created in this manner get updated. This, however, is a one-time update and may be invoked multiple times or continuously for a smooth transition. In addition, in case the host visualization undergoes a rotation event, whether it is due to semantic events or use of the navigation tools, all annotation text elements maintain the upright position so that the users can read the text conveniently at all times.

More advanced annotations than the ones described here are certainly possible. For example, on pen-based interfaces (or current multitouch surfaces that allow the use of a stylus), it may be useful to be able to draw and write directly on the visual representation using free-hand input. The VisDock design is extensible, and we anticipate adding this kind of annotation functionality in the future.

4.8 Supporting Collaboration

VisDock allows end-users to save their markings and annotations by exporting the cloned layers and annotations in a JavaScript Object Notation (JSON) format. The collaborative environment that VisDock offers can take place among (1) viewers with the same visualization (2) viewers with different visualizations that contain the same data contents or data with similar structures. The saved JSON data can be imported into the refreshed web visualization or a visualization that already contains other cloned layers created using VisDock.

4.9 Implementation

VisDock is implemented as a JavaScript library (`visdock.js`) using the jQuery[7] API for general JavaScript features. In addition, VisDock interface information for SVG and CSS components is contained in separate files. Finally, the VisDock utility library uses the KevLinDev computational geometry library[8] and another

6. http://http://bl.ocks.org/dwtkns/4686432
7. http://jquery.com/
8. http://www.kevlindev.com/geometry/

utility library to provide basic intersection testing and bookkeeping objects for use by client programmers.

VisDock is an Open Source project and is publicly available on GitHub at http://VisDockHub.github.io/NewVisDock. Besides the full JavaScript source, this website also contains documentation, tutorials, and examples.

5 Visualization Examples

We have successfully used VisDock to add cross-cutting interaction to a large number of web-based visualizations.[9] In this section we review some representative examples. Please note that we here in no way take credit for the visualizations themselves; all credit is due to the original authors (acknowledged below). Our contribution is merely the integration of VisDock with these existing tools.

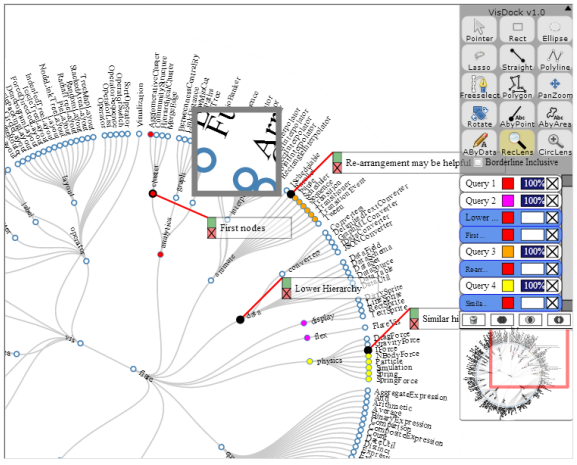


Fig. 7. VisDock-enabled rotating cluster layout [2].

5.1 Rotating Cluster Layout

The D3 website (http://www.d3js.org/) provides an example of a tree structure organized in a radial fashion, which has the root node in the center and the leaf nodes at the circumference. Built by Michael Bostock,[10] this example has a text and circular element for each node and supports interactivity where the visualization rotates with each mouse click and drag. Figure 7 showcases annotations by data space, panning and zooming, various selection methods, and a rectangular magnifying lens (shown as a grey box).

5.2 Force-Directed Layout Visualization

The force-directed layout example, also created by Michael Bostock,[11] contains nodes with an image and text that are loosely bound by interactive force [2]. As shown in Figure 8 this force can be perturbed with the mouse click and drag, which causes the nodes to realign. Our selection tools allow for querying these nodes, annotations by data space help

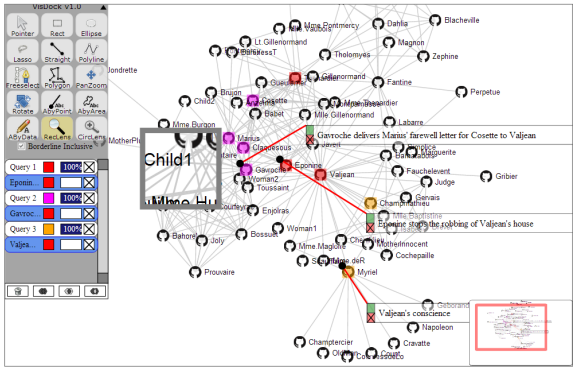


Fig. 8. Node-link diagram with force-directed layout.

users to tag the queries with meaningful information, the pan and zoom methods allow for navigation to optimize the view space, and the magnifying lenses let the user zoom in while leaving the main view intact.

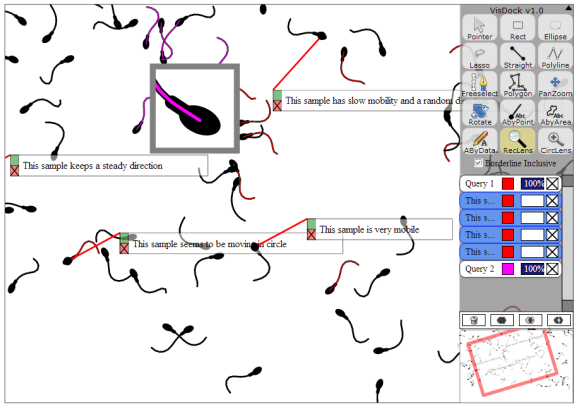


Fig. 9. VisDock-enabled dynamic SVG elements.

5.3 Animated Elements

The tadpole example, also created by Michael Bostock,[12] contains animated objects that continuously change trajectory and shape. Figure 9 shows VisDock integrated into this tadpole example, where the selection tools are used to highlight a number of the moving tails, annotations are used to tag the heads (all annotations will move around the visual space as the heads move), and navigation tools are used for closer inspection of these elements.

5.4 U.S. Unemployment Map

D3's Choropleth example[13] (also created by Michael Bostock) shows U.S. unemployment rates for 2008 down to a county level. The example uses more than 3,000 SVG elements; one for every county. However, the visualization is entirely static and supports no interaction.

Figure 1 shows the visualization with VisDock integrated. Geographical selections can now be anything ranging from an arbitrary rectangular shape in the Midwest to

9. http://bl.ocks.org/VisDockHub
10. http://mbostock.github.io/d3/talk/20111018/cluster.html
11. http://bl.ocks.org/mbostock/950642
12. http://bl.ocks.org/mbostock/1136236
13. http://bl.ocks.org/mbostock/4060606

a path following the West Coast. In addition, geometric selection can simplify tasks such as querying specific patterns such as: counties along the US-Canada border, and counties that are placed within a circular area centered at major U.S. cities. Furthermore, users can now zoom in on specific areas of the country without the original visualization having to be changed. As before, the VisDock integration is only a few lines of code.

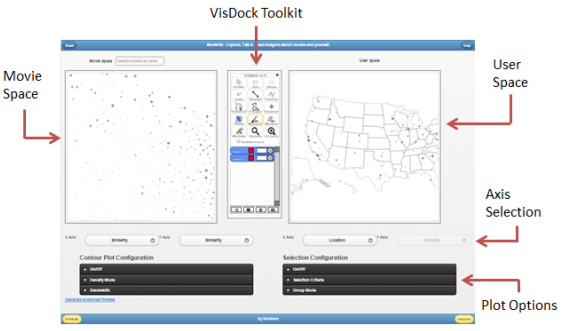


Fig. 10. MovieVis tool with its user interface elements.

5.5 MovieVis and YelpVis

The MovieVis[14] visualization (Figure 10) lets users explore movie preference patterns, whereas YelpVis[15] shows the relationship between local businesses, such as restaurants, and the words that used to describe them on Yelp (Figure 10). Viewers may explore the data space by changing data axes, navigating the data space, and selecting and annotating their findings. Both visualizations use VisDock to enable sophisticated query selections. The system consists of two canvases, and various tools for data exploration.

Based on our experiences integrating VisDock into these two tools, we extended the library to also include support for multi-view visualizations such as these ones. Activating multiple canvases requires the user to initialize VisDock with specifications (dimensions and layout) for each individual viewport. The client programmer can then access each individual viewport `ndx` using the call `VisDock.getViewport(ndx)`.

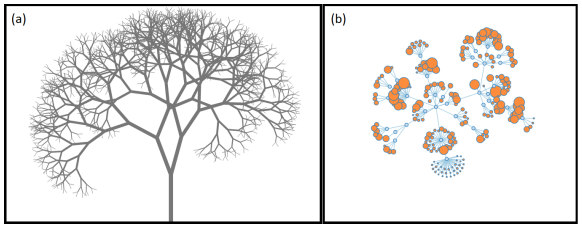


Fig. 11. (left) Binary tree visualization by Peter Cook, and (right) collapsible force layout by Michael Bostock.

14. http://vistalk3.herokuapp.com/movievis
15. http://vistalk3.herokuapp.com/yelpvis

6 Using VisDock for Visualization

To best demonstrate VisDock's utility and ease of use, we here present two detailed examples of how to integrate the toolkit with existing implementations of both static and dynamic visualizations. Figure 11 shows the initial visualizations: a static tree, and a force-directed graph.

6.1 Static Tree Visualization

The binary tree visualization[16] is a static visualization generated using a recursive algorithm with nodes branching in random yet controlled directions. This visualization offers some interactivity when the mouse pointer hovers over tree nodes, which highlights the path from the node to the root. Written using the D3 library [2], this beautiful example contains over a thousand SVG line elements, each representing a tree node. To add a realistic feel to the tree structure, the algorithm generates nodes (branches) in higher parts of the hierarchy in thicker strokes, while those in lower in the hierarchy use thinner strokes (Figure 12). The initialization of the VisDock toolkit can be invoked in a straightforward manner as discussed in Section 4.

We use very simple intersection testing against line elements. The VisDock utilities library is able to handle any polygon shape (even non-convex). Listing 8 gives an example of a concise JavaScript source code for implementing the VisDock event handler.

```
visdock.eventHandler = {
  getHits: function(points, inc) {
    var shapebound = new createShape(points);
    return shapebound.intersectLine
      (viewport.selectAll("line"), inc);
  },
  setColor: function(items, queryId) {
    for (var i = 0; i < items.length; i++) {
      VisDock.utils.addLineLayer(items[i]);
    }
  },
  changeColor: function(queryId, color) {
    VisDock.utils
      .changeColor(color, queryId, "stroke");
  },
  changeVisibility: function(style, queryId) {
    VisDock.utils.changeVisibility(style, queryId)
  },
  removeColor: function(queryId, color) {
    visdock.getViewport().select("#" + queryId)
      .remove();
  }
};
```

Listing 8. An example Event handler for binary tree visualization (refer to the GitHub page for more details).

Because host visualizations integrating VisDock can expect their visual elements to be part of several concurrent selections, it is not sufficient for the visualization to merely change the color of the underlying elements. Instead, for every selection, we clone the SVG line element representing nodes (branches) on top of the original nodes with a user-defined opacity. The cloned elements are tagged with a unique ID given by VisDock, allowing the elements to be easily retrieved for modification or removal.

16. http://prcweb.co.uk/lab/d3-tree/

The finished binary tree visualization with VisDock integrated and in action is shown in Figure 12(left).

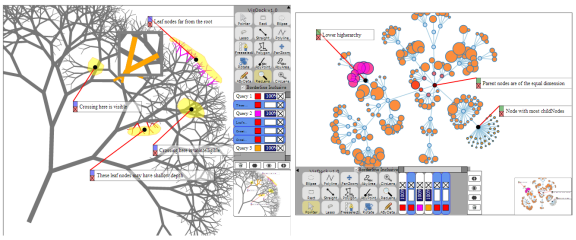


Fig. 12. VisDock-integrated binary tree visualization (left) and collapsible force layout (right).

6.2 Dynamic Graph Visualization

Figure 12(right) shows the collapsible force layout example built by Michael Bostock.[17] This force-directed graph layout example, like many other visualizations using force-directed layout, allows users to drag nodes around as well as hide the children of a node by double-clicking the node. Although nodes change their coordinates in response to the user's pointing, the selection layers that are cloned in a manner similar to those in the binary tree example can have their coordinates similarly updated with a simple command:

```
VisDock.updateLayers();
AnnotationsByData.update();
```

Listing 9. Updating cloned layers and annotations to new coordinates when the original elements animate.

Listing 9 may be invoked continuously to create smooth animations of the annotations and cloned layers over time.

7 Evaluation

We conducted two qualitative usability studies to assess the utility of VisDock. Below we discuss them in detail. The main difference between the studies was the participants:

- **End users:** In-depth data exploration for casual end users with the VisDock-enabled MovieVis tool; and
- **Client programmers:** Informal usability study with visualization programmers from a different research group integrating VisDock in their current projects.

In addition, we include here feedback on the usability and affordability of the tookit posted on the VisDock forum[18] by a number of developers who used VisDock.

7.1 Data Exploration with End Users

Method. We recruited 11 (8 male, 3 female, average age 26) paid participants from the student population at our university to use the MovieVis and YelpVis tools in 20-minute data exploration sessions. As described in Section 5 and shown in Figure 10, these visualizations consist of complicated data points. We imported VisDock into the visualizations to help the viewers to explore, navigate and search for meaningful patterns instead of designing the individual tools from scratch. Participants were given a 10-minute training session prior to starting these sessions, and was given a post-test survey after completing them. The sessions consisted of using VisDock to view movie ratings and business reviews, and to record insights from them using comments. The participants' comments were saved to the server and reviewed later.

Results. Not surprisingly, all participants were able to successfully use the VisDock interactions and gave positive feedback on the utility of the interface. Participants wrote a total of 71 comments for MovieVis and 52 comments for YelpVis during the exploration sessions (an average of 6.5 and 4.7 comments per participant). All participants used the VisDock selection tools to create queries consisting of multiple items, and the VisDock navigation tools to zoom and pan around in the two scatterplot views. In addition, a majority (9 out of 11) used the annotation tools to link their comments with points or regions on the visual representation. Informal observations and post-test interviews indicated no major issues with the usability or functionality of the VisDock interface. While designing these individual tools may be a daunting task in complex visualizations like MovieVis and YelpVis, the fact that the participants were able to explore the data and gather insights in VisDock-enabled MovieVis and YelpVis reinforces VisDock's ease-of-use and effectiveness. Furthermore, several participants requested more advanced selection and query management tools, including multiple selection, sophisticated set operations, and improved navigation techniques.

7.2 Developing with Client Programmers

Method. We report the evaluation of VisDock and the comments and feedback we gathered from developers who used VisDock and participated in the discussion of the system on the online forum. For the evaluation, we conducted an informal focus group evaluation of VisDock inspired by the methodology previously used by Klemmer et al. [39]. Six graduate students from a neighboring visualization research group at our university participated in the study. The participants all had varying experience in web-based visualization using JavaScript (self-reported); one was an advanced developer, whereas the others had novice to intermediate skills. All had built basic charts such as scatterplots, bubble charts, and pie charts.

The study consisted of a two-hour group session with five of our participants (the advanced-level student participated remotely). The session started with an initial 20-minute presentation on the VisDock toolkit. The participants were then given access to the VisDock GitHub website and were asked to use the tutorials, examples, and documentation to integrate the toolkit in their own projects. They were given a time limit of 90 minutes. Two experimenters participated in the evaluation session, taking notes, and answering questions. After the session, participants were asked to give feedback on the focus group session and the toolkit.

Results. Figure 13 shows four screenshots of outcomes from the study. The advanced participant successfully de-

17. http://bl.ocks.org/mbostock/1062288
18. http://groups.google.com/forum/?hl=en#!forum/visdock_group

veloped an interactive visualization using only the online VisDock source repository and tutorial without face-to-face communication with the researchers. The tool visualizes the converted frequency domain signal of a signal in the time domain. This example indicates that the VisDock library is accessible and understandable for expert programmers.

For the less experienced participants, all were able to import the VisDock toolkit in their project (i.e., so that the VisDock interface showed up in their visualization), but only three out of five were able to properly implement the VisDock event handler. Two of the participants were unable to implement the event handler; we found that one of these participants had an existing bug in their code.

All user study participants as well as independent developers noted that they were impressed with the interactions that VisDock provides without coding. One participant mentioned that VisDock can be useful in collaborative working environment because any VisDock-enabled visualization can allow viewers to explore and mark their findings and share the annotated visualization with others. She even suggested a sophisticated strategy to generate a URL to save the annotations and markings. One developer remarked that although VisDock saved him a lot of coding effort in building navigation/annotation tools in his visualization, he thinks VisDock would be more user-friendly if the toolkit were totally separate from the canvas viewport.

On the other hand, although they had positive sentiment toward the ease of use of importing VisDock, many thought that the VisDock event handler could be made simpler. For instance, one developer remarked that selection functions can be made much more condensed so that the developer's coding responsibility becomes less complicated. Another developer remarked that integrating VisDock into someone else's visualization is difficult because implementing the event handler interface required knowing all details of the visualization. In fact, implementing of Listings 1 through 6 all requires good knowledge of the structure of the host visualization. This is not totally unexpected since adding a new set of customized interactions would require knowledge of the structure of the original visualizations. We will work toward minimizing developers' burden in implementing VisDock while allowing for flexibly customizing these interactions. One solution to this problem would be a default event handler that developers can employ with a single line of code. However, this would limit developers' ability to customize mapping of interactions to certain events. We also plan to add more illustrative examples to the VisDock GitHub as guides to new developers.

8 Discussion

The fact that our interactions are cross-cutting also means that they cannot be specialized for particular datasets and visual representations. While the scope of cross-cutting interaction for visualization numbers more techniques than the selection, navigation, annotation, and query management techniques we currently include in the VisDock toolkit, we found that these were a good and well-rounded

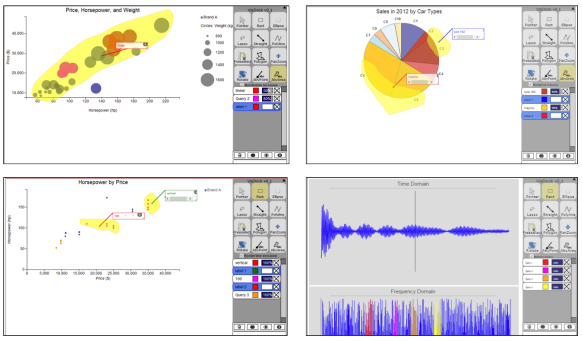


Fig. 13. Visualizations built in the developer study.

subset to start with. Future versions will likely incorporate additional cross-cutting techniques, such as general filtering, surrogate interaction [35], and interactive legends [36].

Furthermore, while integrating VisDock requires a minimum of coding on behalf of the client programmer, it is not possible to use VisDock without any coding at all. One solution to avoid coding altogether may have been to let VisDock operate on the pure SVG node hierarchy, i.e., selecting, grouping, and annotating individual shapes and vector elements. However, virtually all non-trivial visualizations use composite graphics as visual marks, and some graphical elements—such as tick marks, labels, and backgrounds—are not strictly part of the visual representation itself. Off-loading some of this responsibility to the client programmer allows for controlling these higher-level aspects at the cost of some code integration. VisDock currently supports multi-canvas visualizations, and efforts to implement brushing and linking and other features are on the way. In addition, we plan to make the VisDock design more customizable so that developers can select specific tools and change the appearance of the graphical interface.

VisDock is currently designed for manipulating the geometric representation of a visualization (in the form of the SVG scene graph), not the underlying data. Unlike Satyanarayan's data-level interactors [34], this limits the type of cross-cutting interactions that can be supported using the library to those restricted to geometric features of the visualization. Other types of interactions, such as filtering, computing derived values, and changing layout, require knowledge of the data representation as well. Cross-cutting interactions in data space would allow for easily implementing filtering, searching, and drilling down into the data. However, this is also left for future work.

Nevertheless, we overcome a few of these limitations by providing callbacks to the underlying visualization. For example, annotations and cloned layers can reference data-driven elements with little explicit coding on the developer's part. However, even though annotations can be attached to SVG elements so that continuous animation is possible, this method does not currently support annotations marking a region. One way to create an annotation referencing an area is a convex hull [3], where a polygon is automatically created to contain the elements that fall in the area, or using Bubble Sets [40], where elements can be

grouped into a few sets with minimal overlapping.

Checking intersections requires algorithms of high complexity if the objects have non-trivial geometric shape. We regard simple shapes as those mathematically easy to describe, such as circles or ellipses as opposed to those composed of numerous vertices and lines. Since VisDock relies heavily on the algorithm that checks for intersection based on geometric alignment of objects, subsets of SVG elements like polylines and polygons may compromise VisDock performance. In this regard, making VisDock more data-centered is part of our future plans.

Finally, our argument for creating VisDock was to provide a basic set of interaction techniques for web-based visualization. However, a potential pitfall with this approach might be to suggest that a consensus has been reached for "standard" interaction techniques for visualization, which is far from true [1], [41]. This, in turn, may discourage further invention in this area. To avoid this pitfall, our ambition is to continue developing and improving the VisDock toolkit, which is aided by the fact that it is an Open Source project on GitHub. Our goal is to make the toolkit responsive, modular, and customizable to any form of cross-cutting interaction technique conceivable, now or in the future.

9 Conclusion and Future Work

We have proposed the concept of cross-cutting interaction techniques for visualization: general yet concrete direct manipulation interactions that apply to a wide range of datasets and visualization techniques. In doing so, we also presented VisDock, a practical JavaScript/SVG library for providing cross-cutting interactions to any web-based visualization. VisDock currently provides advanced selection, query management, navigation, and annotation support with a minimum of additional coding on behalf of the client programmer. We have evaluated the toolkit in two complementary settings: with a group of end-users exploring data in a VisDock-enabled visualization, and with a group of developers integrating the toolkit into their own projects.

Future VisDock features include an improved and streamlined user interface, more advanced navigation techniques, and the use of hotbox-like [42] interface widgets. In particular, we will work toward implementing a new form of annotations allowing VisDock users, both end-users and developers, to scribble notes and marks that are not constrained to a typed medium. The future scope of VisDock implementation will go beyond SVG interfaces to any web-based structures such as CSS to support a wider range of web visualizations. A more data-centered VisDock interface would be implemented to increase the efficiency of the system. Finally, additional new features for the toolkit include linking, brushing, and data-level interactions.

Acknowledgments

We would like to thank S. Karthik Badam, Dennis M. Snell, D. S. Elliott, and M. Y. Shalaginov for helpful discussions and Ji Soo Yi, Sukwon, Lee and Sung-Hee Kim for advice on the VisDock evaluation. Also, we would like to acknowledge all the help we received from communicating with a number of visualization creators, some of whose works are cited in the paper.

References

[1] J. Yi, Y. Kang, J. T. Stasko, and J. A. Jacko, "Toward a deeper understanding of the role of interaction in information visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1224–1231, 2007.

[2] M. Bostock, V. Ogievetsky, and J. Heer, "D3: Data-driven documents," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 6, pp. 2301–2309, 2011.

[3] N. Elmqvist, P. Dragicevic, and J.-D. Fekete, "Rolling the dice: Multidimensional visual exploration using scatterplot matrix navigation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 6, pp. 1141–1148, 2008.

[4] G. W. Furnas and B. B. Bederson, "Space-scale diagrams: Understanding multiscale interfaces," in *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 1995, pp. 234–241.

[5] T. Moscovich, F. Chevalier, N. Henry, E. Pietriga, and J.-D. Fekete, "Topology-aware navigation in large networks," in *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2009, pp. 2319–2328.

[6] J. S. Yi, N. Elmqvist, and S. Lee, "TimeMatrix: Analyzing temporal social networks using interactive matrix-based visualizations," *International Journal of Human Computer Interaction*, vol. 26, no. 11-12, pp. 1031–1051, 2010.

[7] J. Heer, F. B. Viégas, and M. Wattenberg, "Voyagers and voyeurs: supporting asynchronous collaborative information visualization," in *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2007, pp. 1029–1038.

[8] B. Shneiderman, "Direct manipulation: A step beyond programming languages," *Computer*, vol. 16, no. 8, pp. 57–69, 1983.

[9] C. Ahlberg, "Spotfire: an information exploration environment," *SIGMOD Record*, vol. 25, no. 4, pp. 25–29, 1996.

[10] C. Stolte, D. Tang, and P. Hanrahan, "Polaris: A system for query, analysis, and visualization of multidimensional relational databases," *IEEE Transactions on Visualization and Computer Graphics*, vol. 8, no. 1, pp. 52–65, 2002.

[11] A. Bezerianos, F. Chevalier, P. Dragicevic, N. Elmqvist, and J.-D. Fekete, "GraphDice: A system for exploring multivariate social networks," *Computer Graphics Forum*, vol. 29, no. 3, pp. 863–872, 2010.

[12] M. J. McGuffin and I. Jurisica, "Interaction techniques for selecting and manipulating subgraphs in network visualizations," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 937–944, 2009.

[13] J. Heer, M. Agrawala, and W. Willett, "Generalized selection via interactive query relaxation," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '08. New York, NY, USA: ACM, 2008, pp. 959–968.

[14] B. Shneiderman, "The eyes have it: A task by data type taxonomy for information visualizations," in *Proceedings of the IEEE Symposium on Visual Languages*, 1996, pp. 336–343.

[15] S. Jul and G. W. Furnas, "Critical zones in desert fog: Aids to multiscale navigation," in *Proceedings of the ACM Symposium on User Interface Software and Technology*, 1998, pp. 97–106.

[16] C. Appert and J.-D. Fekete, "OrthoZoom scroller: 1D multi-scale navigation," in *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2006, pp. 21–30.

[17] W. Javed, S. Ghani, and N. Elmqvist, "GravNav: using a gravity model for multi-scale navigation," in *Proceedings of the ACM Conference on Advanced Visual Interfaces*, 2012, pp. 217–224.

[18] J. Abello and F. van Ham, "Matrix zoom: A visual interface to semi-external graphs," in *Proceedings of the IEEE Symposium on Information Visualization*, 2004, pp. 183–190.

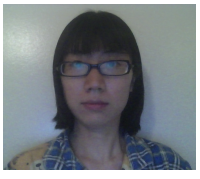
[19] W. Javed, S. Ghani, and N. Elmqvist, "PolyZoom: multiscale and multifocus exploration in 2D visual spaces," in *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2012, pp. 287–296.

[20] N. Elmqvist, N. Henry, Y. Riche, and J.-D. Fekete, "Mélange: Space folding for multi-focus interaction," in *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2008, pp. 1333–1342.

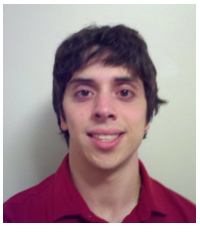
[21] S. Ghani, N. Riche, and N. Elmqvist, "Dynamic insets for context-aware graph navigation," *Computer Graphics Forum*, vol. 30, no. 3, pp. 861–870, 2011.
[22] N. Elmqvist, J. Stasko, and P. Tsigas, "DataMeadow: a visual canvas for analysis of large-scale multivariate data," in *Proceedings of the IEEE Symposium on Visual Analytics Science and Technology*, 2007, pp. 187–194.
[23] W. Wright, D. Schroh, P. Proulx, A. Skaburskis, and B. Cort, "The sandbox for analysis: Concepts and evaluation," in *Proceedings of ACM Conference on Human Factors in Computing Systems*, 2006, pp. 801–810.
[24] J. T. Stasko, C. Görg, and Z. Liu, "Jigsaw: supporting investigative analysis through interactive visualization," *Information Visualization*, vol. 7, no. 2, pp. 118–132, 2008.
[25] Y. Shrinivasan and J. van Wijk, "Supporting the analytical reasoning process in information visualization," in *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2008, pp. 1237–1246.
[26] J. Heer and M. Agrawala, "Design considerations for collaborative visual analytics," *Information Visualization*, vol. 7, no. 1, pp. 49–62, 2008.
[27] J. Heer, S. K. Card, and J. A. Landay, "prefuse: a toolkit for interactive information visualization," in *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2005, pp. 421–430.
[28] J.-D. Fekete, "The InfoVis Toolkit," in *Proceedings of the IEEE Symposium on Information Visualization*, 2004, pp. 167–174.
[29] C. Weaver, "Building highly-coordinated visualizations in Improvise," in *Proceedings of the IEEE Symposium on Information Visualization*, 2004, pp. 159–166.
[30] A. Satyanarayan and J. Heer, "Lyra: An interactive visualization design environment," *Computer Graphics Forum*, vol. 33, no. 3, pp. 351–360, 2014.
[31] D. Ren, T. Höllerer, and X. Yuan, "iVisDesigner: Expressive interactive design of information visualizations," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2092–2101, 2014.
[32] A. Satyanarayan and J. Heer, "Authoring narrative visualizations with ellipsis," *Computer Graphics Forum*, vol. 33, no. 3, pp. 361–370, 2014.
[33] J. Harper and M. Agrawala, "Deconstructing and restyling D3 visualizations," in *Proceedings of the ACM Symposium on User Interface Software and Technology*, 2014, pp. 253–262.
[34] A. Satyanarayan, K. Wongsuphasawat, and J. Heer, "Declarative interaction design for data visualization," in *Proceedings of the ACM Symposium on User Interface Software and Technology*, 2014, pp. 669–678.
[35] B. Kwon, W. Javed, N. Elmqvist, and J. S. Yi, "Direct manipulation through surrogate objects," in *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2011, pp. 627–636.
[36] N. Riche, B. Lee, and C. Plaisant, "Understanding interactive legends: a comparative study with standard widgets," *Computer Graphics Forum*, vol. 29, no. 3, pp. 1193–1202, 2010.
[37] C. Viau, M. J. McGuffin, Y. Chiricota, and I. Jurisica, "The FlowVizMenu and parallel scatterplot matrix: Hybrid multidimensional visualizations for network exploration," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 6, pp. 1100–1108, 2010.
[38] K. Perlin and D. Fox, "Pad: An alternative approach to the computer interface," in *Computer Graphics*, 1993, pp. 57–64.
[39] S. R. Klemmer, J. Li, J. Lin, and J. A. Landay, "Papier-Mache: toolkit support for tangible input," in *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2004.
[40] C. Collins, G. Penn, and S. Carpendale, "Bubble Sets: Revealing set relations with isocontours over existing visualizations," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1009–1016, 2009.
[41] W. A. Pike, J. T. Stasko, R. Chang, and T. A. O'Connell, "The science of interaction," *Information Visualization*, vol. 8, no. 4, pp. 263–274, 2009.
[42] G. Kurtenbach, G. W. Fitzmaurice, R. N. Owen, and T. Baudel, "The hotbox: Efficient access to a large number of menu-items," in *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 1999, pp. 231–237.

Jungu "Joe" Choi received a bachelor and master in electrical and computer engineering in 2010 and 2012, respectively, from Purdue University in West Lafayette, IN, USA. He is currently a Ph.D. student at the same institution working on Atomic, Molecular, and Optical (AMO) physics and engineering.

Deok Gun Park received a bachelor in electrical engineering in 2000 and a master's degree in biomedical engineering in 2002 at Seoul National University, Seoul, South Korea. Currently, he is pursuing a Ph.D. degree in computer science at University of Maryland, College Park, MD, USA.



Yuetling Wong received a bachelor and master in computer science and technology in 2007 and 2010, respectively, from Tsinghua University in Beijing, China. She is currently a Ph.D. student in the School of Electrical and Computer Engineering at Purdue University in West Lafayette, IN, USA.



Eli Fisher received a bachelor of computer engineering degree in 2014 from Purdue University in West Lafayette, IN, USA. He now works for Microsoft Corporation in Redmond, WA, USA.

Niklas Elmqvist received the Ph.D. degree in 2006 from Chalmers University of Technology in Göteborg, Sweden. He is an associate professor in the College of Information Studies at University of Maryland, College Park, MD, USA. He was previously an assistant professor in the School of Electrical & Computer Engineering at Purdue University in West Lafayette, IN. He is a senior member of the IEEE and the IEEE Computer Society.