Getting Portals to Behave

Chris Olston[*]
Stanford University
olston@db.stanford.edu

Allison Woodruff
Xerox PARC
woodruff@parc.xerox.com

Abstract

Data visualization environments help users understand and analyze their data by permitting interactive browsing of graphical representations of the data. To further facilitate understanding and analysis, many visualization environments have special features known as portals, *which are sub-windows of a data canvas. Portals provide a way to display multiple graphical representations simultaneously, in a nested fashion. This makes portals an extremely powerful and flexible paradigm for data visualization. Unfortunately, with this flexibility comes complexity. There are over a hundred possible ways each portal can be configured to exhibit different behaviors. Many of these behaviors are confusing and certain behaviors can be inappropriate for a particular setting. It is desirable to eliminate confusing and inappropriate behaviors. In this paper, we construct a taxonomy of portal behaviors and give recommendations to help designers of visualization systems decide which behaviors are intuitive and appropriate for a particular setting. We apply these recommendations to an example setting that is fully visually programmable and analyze the resulting reduced set of behaviors. Finally, we consider a real visualization environment and demonstrate some problems associated with behaviors that do not follow our recommendations.*

Keywords: *Portals, Multiple Views, Data Visualization.*

1 Introduction

Recently, much attention has been devoted to data visualization environments that permit interactive browsing of graphical representations of large data sets [1, 2, 3, 4, 8, 9, 12, 13]. Many of these environments present a two-dimensional infinite canvas[1] of graphical data through which the user can "navigate" to interactively browse the data. Interactive browsing can be a powerful way to understand and analyze data. To further facilitate browsing and analysis, many visualization environments have special features known as *portals* [2, 3, 12, 13],[2] which are sub-windows of a data canvas. Portals provide a way to display multiple graphical representations simultaneously, in a nested fashion. This makes portals an extremely powerful and flexible paradigm for data visualization.

Unfortunately, with this flexibility comes complexity. There are over a hundred possible ways each portal can be configured to exhibit different behaviors. Many of these behaviors have confusing effects. Furthermore, many behaviors are inappropriate for a particular setting. For example, certain behaviors have visually programmable aspects and thus are not appropriate in a browse only setting. It is desirable to eliminate confusing and inappropriate behaviors. In this paper, we construct a taxonomy of portal behaviors and give recommendations to help designers of visualization systems decide which behaviors are intuitive and appropriate for a particular setting.

So that this analysis can apply as broadly as possible, we model the environment as follows. A *visualization environment*[3] is any system that displays (a portion of) a two-dimensional canvas, called the *parent canvas*, which contains a set of objects. Some of these objects may be portals, which are special objects that show a portion of another canvas, called the *child canvas*. Note that the child canvas can be the same as the parent canvas. A concrete example is illustrated in Figure 1, which is a screenshot from the DataSplash visualization environment [12] showing a parent canvas of filled polygons and other objects. The polygons are U.S. states. The three square objects are portals, each located at the coordinates of a major city. Each portal is a sub-window in the parent canvas that shows a portion of a child canvas, which in this case contains a bar chart describing transportation data for the city. Note that, in general, portals can be of any shape.

In addition to showing a portion of the child canvas, portals can be used for navigation. Many visualization environments allow users to "enter" portals to instantly navigate to the child canvas. For example, entering one of the portals in Figure 1 causes the corresponding bar chart canvas to become the parent canvas and the bar chart to fill the entire screen.

[*]Supported by a National Science Foundation graduate research fellowship.

[1]Called a "surface" in Pad [13].

[2]Portals have many names, including "wormholes" in Tioga-2 [2], the predecessor to DataSplash.

[3]Although we use the term "visualization environment" throughout this paper, our analysis also applies more generally to multiple view systems that contain views that are related (usually via an underlying spatial model).

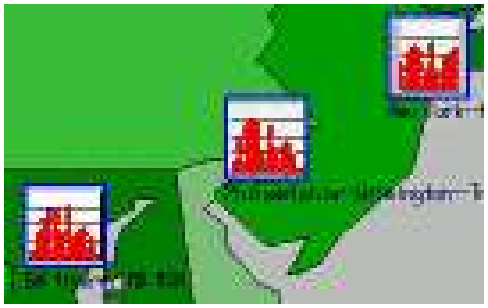

Figure 1: An example DataSplash visualization with portals.

Since portals allow users to instantly navigate to another location and/or another canvas, they can be used to create visual versions of hyperlinks and bookmarks. There are also many other uses for portals. Portals can be used to implement a variety of tools to help the user browse and understand the data being visualized by showing multiple views simultaneously. The views can show different representations of the data, or data at different levels of magnification. For example, *magnifying glasses* [2, 13] show a magnified view of the data below them and can in some cases be repositioned by the user.

Each portal tool comes in many varieties, depending on how the user is permitted to interact with the tool. For example, the user may or may not be permitted to reposition a magnifying glass. When the user pans and zooms, magnifying glasses can travel with the user or remain fixed relative to the canvas. Furthermore, the user may or may not be allowed to pan and zoom the child canvas inside the magnifying glass. If zooming inside the magnifying glass is allowed, it may change the magnification factor, zoom the entire parent canvas along with the child canvas inside the magnifying glass, automatically resize the magnifying glass, or apply some combination of these effects. Finally, if resizing the magnifying glass is allowed, enlarging it may increase the magnification, or alternatively cause more data to become visible with the same magnification.

In this paper, we concentrate on the properties that govern which user operations are allowed and which user operations trigger which other operations. Considering only these fundamental properties we have identified, there are 125 possible behaviors for each portal. Enabling all possible behaviors is probably not a good idea for several reasons. First, it is unlikely that the user can be expected to understand all 125 behaviors. Second, some of the behaviors may have confusing effects. Finally, many of the behaviors may simply not be useful or appropriate, depending on the setting.

This paper addresses this issue by formally defining the behavior space and proposing a two-step procedure for reducing the behavior space to a handful of intuitive and appropriate behaviors. The first step eliminates behaviors that are confusing. For this purpose we propose a set of rules that can be applied in all settings. The second step eliminates those behaviors that are not appropriate in a particular setting. To illustrate the process of eliminating inappropriate behaviors, we consider a specific example setting—a fully visually programmable environment—and describe rules to be applied. We then analyze the resulting reduced set of behaviors. We omit an exhaustive analysis of possible settings.

The analysis in this paper focuses uniquely on the binary choice of which user operations trigger which other operations. We do not consider the exact behavior when changes are propagated via triggering. For example, when the user enlarges a magnifying glass, either the same portion of the canvas remains visible but is shown at increased magnification (*i.e.*, no triggering), or a larger portion of the canvas is shown at the same magnification (*i.e.* triggering). Our analysis addresses which of these two scenarios may occur, but not how much more of the canvas becomes visible nor how much the magnification increases. These issues are orthogonal to the discussion in this paper.

Others have focused on modeling the way operations are correlated in specific behaviors [3, 5, 6, 10], often using a spatial model. However, we are not aware of any work on enumerating fundamental behaviors. Furthermore, to our knowledge no work has focused on making recommendations for choosing a reduced set of behaviors from the large space of possible behaviors we identify.

The remainder of this paper is structured as follows. After giving an overview of some useful portal tools in Section 2, we present our model for portals in Section 3, which focuses on a set of fundamental binary properties that determine the portal behavior. To reduce the space of allowed behaviors by eliminating confusing behaviors, we propose rules in Section 4. Next, in Section 5, we describe rules to further reduce the space for an example setting by eliminating inappropriate behaviors, and we analyze the remaining reduced set of behaviors. Section 6 considers a real visualization environment, and demonstrates some problems associated with behaviors that do not conform to the suggested rules. Finally, in Section 7, we discuss avenues for future work.

2 Portal Tools

In this section, we give examples of portal tools. We make no claim that this is an exhaustive list of useful tools. Each portal tool comes in many varieties, depending on which aspects are visually programmable by the user and which are not. Since there are so many variations on each tool, each requiring a different behavior, we omit a thorough discussion. Instead, in Section 3, we describe a set of fundamental properties that govern the behavior of a portal.

2.1 Visual Hyperlinks

Visual hyperlinks are analogous to hypertext hyperlinks (*e.g.*, between Web pages) in that they allow the user to instantly navigate between a location on one canvas to some location on another canvas. Additionally, visual hyperlinks display the contents of the destination in a sub-window, to provide a "preview" of the destination.

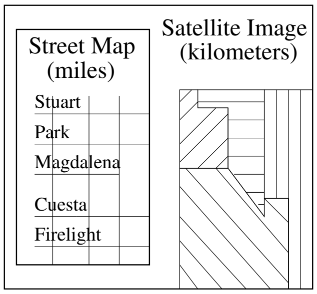

Figure 2: A visualization showing a satellite image in the parent canvas and a street map in a coordinated view tool.

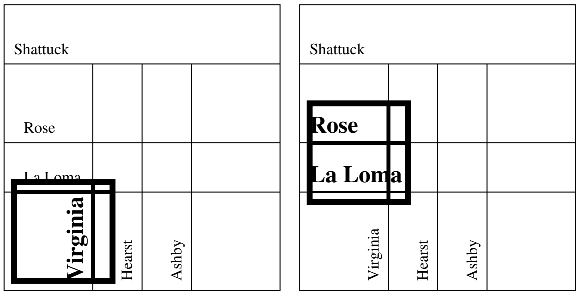

Figure 3: Two snapshots of a map visualization having a movable magnifying glass with the lens and lens^{-1} dependencies enabled. The user can go from the before image (left) to the after image (right) either by repositioning the magnifying glass or by panning the magnified image inside the magnifying glass.

2.2 Bookmarks

Bookmarks are visual hyperlinks that remain on the screen at all times and allow the user to instantly navigate to the location displayed in the bookmark portal. *Indexes* in Pad++ [3] are similar to bookmark tools.

2.3 Coordinated Views

A *coordinated view* [5, 15] is a portal tool that remains on the screen at all times and shows a different representation of the data in the main window. The data shown in the coordinated view corresponds to some region of the main window and is not related to the position of the portal. Coordinated views can be quite useful in applications such as astronomy, medical imaging, comparative cartography, and structural analysis, where data objects have several alternative representations showing different characteristics [15]. For example, Figure 2 illustrates a visualization displaying a satellite image in the parent canvas and a street map of the same geographic region in a coordinated view tool.

2.4 Overviews

An *overview* is a portal tool that shows a demagnified (zoomed out) copy of the canvas and is fixed on the screen to help orient the user during navigation. In the literature, the combination of overview portal and parent canvas is called an *overview and detail* view [7].

2.5 Filters

Filters [13] show a different graphical representation of the region of the canvas that is occluded by the portal object. Filters are useful for displaying two different representations of the same data.

Movable filters are filters that the user is allowed to reposition and resize. Afterward, whatever region of the parent canvas is newly occluded by the filter is instead displayed inside the filter, as the alternative representation. Movable filters allow the user to interactively adjust the region of the canvas being filtered. This can be a useful behavior for a number of reasons outlined in [6]. *Magic Lenses* in the See-Through Interface [6, 15] and Pad++ [3] are movable filter tools.

Filters and movable filters can either be *canvas-stationary* or *user-stationary*. Canvas-stationary filters are stationary relative to the parent canvas, and do not remain with the user during navigation unless the user explicitly moves them. Canvas-stationary filters are often kept positioned near a particular region of the canvas that is interesting to see through a filter. On the other hand, user-stationary filters remain fixed in position and size on the screen during navigation. User-stationary filters are useful when the user wishes to see a filtered view of every part of the canvas visited, possibly in conjunction with the regular, unfiltered view.

2.6 Magnifying Glasses

A *magnifying glass* [2, 13] is a portal tool that shows a magnified (zoomed in) view of the region of the canvas underneath.[4]

Movable magnifying glasses are magnifying glasses that the user is allowed to reposition (and resize). Movable magnifying glasses can help the user see different portions of the canvas in detail without navigating. For example, a movable magnifying glass might be useful when using a map visualization to view street names that are otherwise too small to read without zooming in, as illustrated in Figure 3 (the reader should ignore the caption for now, as it contains concepts that have not yet been introduced).

As with filters, magnifying glasses and movable magnifying glasses can either be canvas-stationary or user-stationary. Both types can be useful. For example, consider a map visualization used to convey driving directions between two points. When viewing the entire route in freeway-level detail on the screen, it may be useful to have one canvas-stationary magnifying glass over the starting location and another over the destination showing detailed street maps. Alternatively, consider a document visualization. A user-stationary magnifying glass

[4]We draw a distinction between filters (which show a different graphical representation of a given region of a canvas) from magnifying glasses (which show a larger or smaller region of a canvas). We discuss hybrid tools in the following subsection.

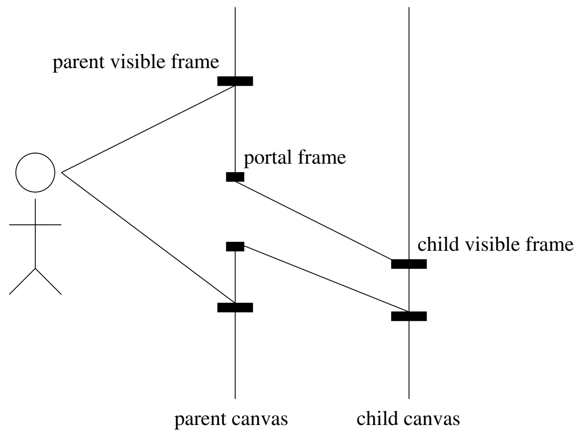

Figure 4: Portal model.

could be positioned in the center of the screen to facilitate reading text while still displaying the context in the periphery.

2.7 Hybrid Tools

Hybrid tools can be created that have properties taken from several portal tools. For example, a magnifying glass/filter hybrid would simultaneously magnify the data and change its representation. Such tools are particularly common in semantic zoom [13] environments that display different representations of data depending on the magnification level.

3 Portal Model

In this section we present the model used in this paper. Our model is simple enough to capture the semantics of many environments with portals. Figure 4 illustrates a side-view of our model, with the user on the left.

The vertical line immediately to the right of the user represents the parent canvas. The *parent visible frame*, or simply parent frame, is the two-dimensional region of the parent canvas displayed on the screen. This frame corresponds to the entire window in Figure 1. The parent canvas may contain one or more portals. Each portal has a *portal frame*, which is a two-dimensional region of the parent canvas. The border of each square in Figure 1 is a portal frame.

The vertical line on the right represents the child canvas associated with a portal. Note that each portal in the parent canvas can point to a different child canvas. The *child visible frame*, or simply child frame, is the two-dimensional region of the child canvas that is displayed inside the portal. This frame corresponds to one of the bar charts in Figure 1.

The position and size of each frame can be adjusted by the user. Editing the parent visible frame is accomplished via *parent navigation* operations, *i.e.*, pan and zoom. Similarly, many environments support *child navigation*, which allows the user to edit the child visible frame. For example, a user could potentially pan and zoom one of the child canvases in Figure 1 to enlarge a portion of the bar chart without affecting the parent canvas.

User Operation	*Frame Edited*
parent navigation (panning and zooming in the parent canvas)	parent visible frame
manipulating (resizing and repositioning) the portal object	portal frame
child navigation (panning and zooming in the child canvas)	child visible frame

Figure 5: Summary of which user operations edit which frames.

In addition to editing the parent and child visible frames via parent and child navigation, a user can edit the portal frame by manipulating the portal object. Recall that a portal can be thought of as a special object that displays another canvas. Some environments permit users to move and resize portal objects in the same manner as other non-portal objects. For example, a user could make one of the portals in Figure 1 smaller and reposition it. Figure 5 summarizes which user operations edit which frames.

While other models are possible, we believe these three frames represent the minimal set of entities needed to describe what appears on the screen. The parent frame describes what portion of the parent canvas is displayed, the portal frame specifies the position of the portal, and the child frame determines what portion of the child canvas appears inside the portal.

The three frames in this model can be thought of as having an ordering in terms of conceptual distance from the user. First, the parent visible frame is the closest to the user since it controls what the user sees of the parent canvas. Second, the portal frame is an element of the parent canvas, where it can be manipulated by the user. Third, the child visible frame is the farthest from the user, because editing it requires navigating the child canvas, which is conceptually located beneath the parent canvas. To indicate this order, we write *parent visible frame* $\succ$ *portal frame* $\succ$ *child visible frame*. The order of frames in terms of conceptual distance from the user is a useful concept that we will invoke later in the paper.

Some environments (*e.g.*, DataSplash [12]) support multiple levels of nesting, where portals can contain portals, and so on, and thus have additional frames beyond the three discussed. In DataSplash, the user is only permitted to edit the three closest frames (the parent visible, portal, and child visible frames), and cannot edit frames of grandchild portals without first entering the child portal. Therefore, in this paper we consider only the three closest frames. However, our model easily generalizes to environments that permit the user to edit frames of nested portals.

Now that we have presented our model for portals, we turn to a discussion of portal properties. Although there are a multitude of properties that portals can have, we wish our model to be basic enough to capture the semantics of as many environments with portals as possible. Therefore, our model considers only two types of binary properties, which we consider fundamental, called *frame editability*, which applies to frames, and *frame dependency*, which applies to ordered pairs of frames. Each property of either type can be either enabled or disabled. A set of enabled properties is called a *behavior*. Various instances of the tools discussed in Section 2 can be implemented as portals with different behaviors. We now discuss the editability and dependency properties in turn.

3.1 Frame Editability Properties

It is possible to disallow edits to one or more frames by disabling its *editability property*. Recall from Figure 5 that the parent and child frames are edited by parent and child navigation operations, respectively, and the portal frame is edited by repositioning and resizing the portal object. It is often desirable to disallow edits to certain frames in some behaviors. For example, some behaviors require portals to remain fixed and do not permit users to resize or reposition portals. Therefore, each portal has three binary editability properties, one for each frame. For convenience, we write $editable(A)$ to indicate that a frame A is editable.

3.2 Frame Dependency Properties

In the absence of any dependencies between frames, each frame is independent. In other words, editing one frame does not affect the other two frames. For example, making one of the portals in Figure 1 smaller does not affect the child visible frame, which remains the same size. Thus, the same child frame (the entire bar chart) will be displayed in the now smaller portal frame.[5]

Frame dependencies are properties that cause one frame to change automatically when the user edits another frame. Dependencies are one-way links between ordered pairs of frames, of which there are six. Each dependency can either be enabled or disabled. If a dependency from frame A to frame B (written $A \rightarrow B$) is enabled, whenever the user edits the position and/or size of frame A, the position and/or size of frame B changes automatically. Of course, a dependency $A \rightarrow B$ cannot be enabled unless frame A is editable. On the other hand, even if frame B is not editable, the dependency can be used. Editability only restricts direct editing, while still allowing indirect editing via frame dependencies. For notational convenience, we write $A \rightsquigarrow B$ if a (possibly empty) chain of dependencies is enabled from frame A to frame B. Note that since editing a frame always changes it, $A \rightarrow A$ and $A \rightsquigarrow A$ are trivially true for any frame A.

There are six possible frame dependencies in our three-frame model. We classify the dependencies into forward and reverse dependencies. A dependency $A \rightarrow B$ is forward if $A \succ B$, and reverse otherwise. Each forward dependency $A \rightarrow B$ has an inverse, $B \rightarrow A$ that is a reverse dependency. Conversely, the inverse of a reverse dependency is a forward dependency. It is useful to think of the frame dependency properties

[5]For the purposes of this paper, we assume that the shape of the child visible frame must be the same as the shape of the portal frame. When these frames are rectangles, the aspect ratios must be equal.

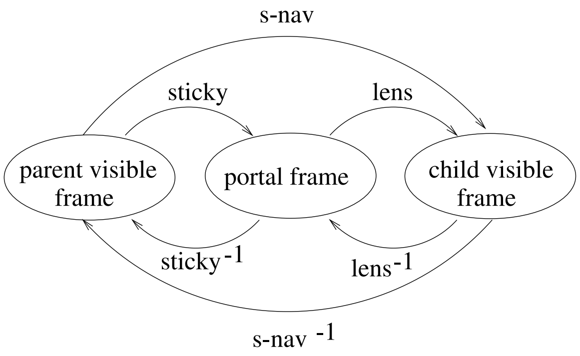

Figure 6: A graph of the behavior having all the frame dependencies enabled. The nodes are displayed in order of conceptual distance from the user, from left to right. The name of each dependency is shown next to its corresponding edge.

of a portal as a directed graph with three vertices, one for each frame (parent visible, portal, and child visible). An edge from frame A to frame B means that the frame dependency $A \rightarrow B$ is enabled. Figure 6 shows a graph for the behavior with all the frame dependencies enabled. The nodes are displayed in order of conceptual distance from the user, from left to right. Each edge in the graph is labeled with the name we give to the corresponding dependency.

Next, we describe the three forward dependencies (which we call *sticky*, *s-nav*, and *lens*) and their inverses (*sticky*$^{-1}$, *s-nav*$^{-1}$, and *lens*$^{-1}$) in Sections 3.2.1, 3.2.2, and 3.2.3. Then, in Section 3.3, we describe *dependency mappings*, which specify the exact way in which editing a frame automatically changes a dependent frame.

3.2.1 Sticky and Sticky^{-1} Dependencies

We refer to the dependency *parent visible frame* $\rightarrow$ *portal frame* as the *sticky dependency*, which can be used to make portals "stick to the screen" [2, 3]. In this scenario, when the user edits the parent frame (*e.g.*, by panning and zooming), the portal frame changes so that its size and position relative to the parent frame remain constant. The sticky dependency is useful for portals that are intended to remain on the screen at all times.[6]

The *sticky*$^{-1}$ *dependency* is the inverse of the sticky dependency: *portal frame* $\rightarrow$ *parent visible frame*. This dependency is useful when moving a portal should automatically change the parent frame. This dependency is often temporarily applied when the user drags an object off the edge of the screen, allowing the parent frame to follow the drag. For the sake of simplicity, in this paper we limit ourselves to behaviors that arise from permanently enabling the sticky^{-1} dependency.

[6]This property and its inverse can also be applied to objects other than portals. For example, shapes such as triangles or circles may be sticky.

3.2.2 S-nav and S-nav^{-1} Dependencies

We refer to the dependency *parent visible frame → child visible frame* as the *s-nav dependency* (for synchronous navigation), which can be used to apply navigation performed in the parent canvas to the child canvas (possibly with a transformation) [16]. In other words, navigating the parent canvas causes the child canvas inside the portal to automatically navigate. This dependency is useful for portals whose view is somehow linked to the view of the parent, as in coordinated views (Section 2.3) and certain varieties of other tools.

The *s-nav^{-1} dependency* is the inverse of the s-nav dependency: *child visible frame → parent visible frame*. In certain behaviors, navigation in the child canvas automatically navigates the parent. Some varieties of coordinated views and other tools use this effect.

3.2.3 Lens and Lens^{-1} Dependencies

We refer to the dependency *portal frame → child visible frame* as the *lens dependency*, which creates behaviors where editing the portal frame affects which part of the child canvas is displayed inside the portal. This dependency is useful for movable filters and magnifying glasses (see Sections 2.5 and 2.6).

The *lens^{-1} dependency* is the inverse of the lens dependency: *child visible frame → portal frame*. When the lens^{-1} dependency is enabled, navigating the child canvas inside the portal changes the portal's position and/or size on the surface of the parent canvas. This dependency can be used for many purposes, including to create a variety of movable filters in which panning the child canvas to change the region being filtered automatically repositions the filter above the corresponding region in the parent canvas.

3.3 Dependency Mappings

When a dependency $A \rightarrow B$ is enabled, it is useful to think of the relationship between frames A and B as a *dependency mapping* from edits by the user of frame A to changes automatically performed on frame B. Although a detailed discussion of dependency mappings is beyond the scope of this paper, which focuses instead on the binary choice of which dependencies are enabled, we give a short illustrative example.

Consider a magnifying glass with the lens dependency enabled. One interesting dependency mapping for the lens dependency might have, among other things, the following characteristic. If the user enlarges the magnifying glass (portal frame), then the region of data displayed inside the magnifying glass (child frame) also enlarges. Such a mapping could be used to maintain a fixed ratio between the portal frame and child frame sizes, to hold the magnification factor constant. In this situation, the magnification factor is a parameter in the dependency mapping.

Ideally, we would like our analysis in this paper to be orthogonal to the choice of dependency mappings, since we concentrate on the binary choice of enabling or disabling each dependency. However, we do make two minor restrictions on dependency mappings to simplify our analysis without loss of applicability. It is our belief that systems that do not meet these restrictions have unintuitive and undesirable behaviors. Specifically, we only consider environments that obey *mapping transitivity* and *mapping inversion*, which we discuss next.

3.3.1 Mapping Transitivity

In environments with *mapping transitivity*, if dependencies $X \rightarrow Y$, $Y \rightarrow Z$, and $X \rightarrow Z$ are enabled, then the mapping for the $X \rightarrow Z$ dependency is the composition of the mappings for $X \rightarrow Y$ and $Y \rightarrow Z$. In general, the composition of mappings along any path from X to Z must produce the same mapping as $X \rightarrow Z$. We say that a dependency $X \rightarrow Z$ is *derivable via transitivity* if there is a path $X \rightsquigarrow Z$ other than $X \rightarrow Z$. This means that the mapping for $X \rightarrow Z$ can be derived by composing a sequence of mappings for dependencies from X to Z.

Environments that do not employ mapping transitivity can allow confusing behaviors. Consider a behavior with the following three dependencies: $X \rightarrow Y$, $Y \rightarrow Z$, and $X \rightarrow Z$. Editing X to X' automatically changes Y to Y' and Z to Z'. If mapping transitivity is not employed, then manually editing Y to Y' can change Z to something other than Z'. This is counter intuitive.

As a concrete example, consider a behavior with dependencies *parent → portal* (sticky), *portal → child* (lens), and *parent → child* (s-nav). Say that moving the portal to the right by one unit automatically pans the child canvas to the right by some amount x. The user can also cause the portal to move one unit to the right indirectly via the sticky dependency by panning the parent. Without mapping transitivity, doing so pans the child canvas by some amount other than x. This effect is confusing because intuitively the user expects the child canvas to pan by x, which would occur if they had moved the portal directly.

On the other hand, with mapping transitivity, the s-nav mapping is derivable from the sticky and lens mappings. In this case, whenever the portal is moved (either directly, or indirectly by panning the parent), the same effect is manifest on the child frame. This effect is determined solely by the lens dependency mapping.

3.3.2 Mapping Inversion

In environments with *mapping inversion*, if a dependency $A \rightarrow B$ is enabled in conjunction with its inverse ($B \rightarrow A$), then the two mappings are inverses of each other. In other words, when two frames are co-dependent, the user can directly edit either frame to produce the same effect.

Having two frames that are co-dependent without mapping inversion is confusing. For example, consider the canvas with a coordinated view illustrated in Figure 2. Say the user pans the satellite image (parent canvas) to the right by one kilometer, which causes the street map (child canvas) inside the coordinated view to pan to the right by 0.625 miles. Without mapping inversion, manually panning the street map to the right by 0.625 miles does not pan the satellite image to the right by one kilometer, as would be expected when the parent and child frames are co-dependent.

Mapping inversion can be seen as a special case of mapping transitivity. Consider two co-dependent frames A and B. By mapping transitivity, the mapping along the path $A \rightarrow B \rightarrow A$ must equal the mapping for $A \rightarrow A$, which is the identity mapping.

We say that a dependency $A \rightarrow B$ is *derivable via inversion* if the inverse dependency $B \rightarrow A$ is enabled. This means that the mapping for $A \rightarrow B$ can be derived from the mapping for $B \rightarrow A$ by inverting it. Note that whenever a dependency and its inverse are both enabled, they are always mutually derivable via inversion.

4 Reducing the Number of Behaviors

The *behavior* of a portal is defined by the set of frame editability and frame dependency properties that are enabled. We count the number of possible behaviors, keeping in mind that a dependency $A \rightarrow B$ cannot be enabled unless frame A is editable. First, if none of the frames are editable, the only valid behavior has no dependencies. Second, if any one of the three frames is editable, it is possible to select any subset of the two outgoing dependencies from the enabled frame, for a subtotal of $3 \cdot 2^2$ behaviors with any one frame editable. Third, if any two of the three frames are editable, then any subset of the four outgoing dependencies from the two enabled frames can be selected, giving $3 \cdot 2^4$ behaviors. Finally, if all three frames are editable, there are 2^6 behaviors. Summing up the possibilities, there are $1 \cdot 2^0 + 3 \cdot 2^2 + 3 \cdot 2^4 + 1 \cdot 2^6 = 125$ different behaviors, considering only the frame editability and frame dependency properties.

Presenting the user with a choice of any possible behavior for each portal is probably not a good idea for several reasons. First, it is unlikely that the user can be expected to understand all 125 behaviors. Second, some of the behaviors may have confusing effects. Finally, many of the behaviors may not be appropriate, depending on the setting.

In the remainder of this section, we present a set of rules that eliminate confusing behaviors in all settings. Then, in Section 5, we focus on a specific example setting and discuss ways to further limit the space to behaviors that are appropriate for the particular setting we consider.

4.1 Dependency Transitivity

We recommend enforcing *dependency transitivity*. In a behavior that is not dependency transitive, when the user explicitly edits frame Y, frame Z automatically changes, but when the user edits some other frame X that automatically changes Y, frame Z does not change. This is counterintuitive.

For example, consider a behavior that is not dependency transitive with only dependencies *parent* $\rightarrow$ *portal* and *portal* $\rightarrow$ *child* enabled. Moving the portal automatically pans the child canvas, and panning the parent automatically moves the portal but does not pan the child canvas. Since behaviors that do not obey dependency transitive are unintuitive, we recommend the use of the following rule:

Dependency Transitivity Rule: $\forall X, Y, Z((X \rightarrow Y) \wedge (Y \rightarrow Z) \Rightarrow (X \rightarrow Z))$

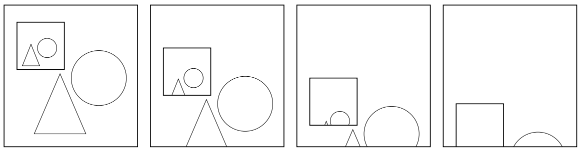

Figure 7: A behavior with bypassing as the user pans upward in the parent canvas.

4.2 No Bypassing

We strongly believe that *bypassing* can be confusing for the user. Recall from Section 3 that frames have an order based on conceptual distance from the user. Bypassing occurs when editing one frame automatically changes a closer or farther frame without changing an intermediate frame. The No Bypassing rule states that no dependency can bypass a frame in the conceptual distance order. In other words, if editing one frame automatically changes a closer or farther frame, then it must also change all frames in between.

For example, consider a behavior where the only enabled dependency is *parent* $\rightarrow$ *child* (s-nav). Figure 7 illustrates the effect of panning upward in the parent canvas. The portal frame remains stationary relative to the other objects in the parent canvas, yet the child canvas inside the portal pans. This behavior can be quite confusing, so we suggest the following rule:

No Bypassing Rule: $\forall X, Y, Z((X \rightarrow Z) \wedge ((X \succ Y \succ Z) \vee (Z \succ Y \succ X)) \Rightarrow (X \rightarrow Y))$

4.3 Only Forward Derivable Reverse Dependencies

Behaviors with reverse dependencies are usually confusing. In general, users expect edits of a frame to propagate to more distant frames via forward dependencies, but not the other way around.

However, reverse dependencies can make sense when they are derivable via transitivity or inversion (recall Sections 3.3.1 and 3.3.2) from some sequence of dependencies that includes a forward dependency. When this is the case, we say that a reverse dependency is *forward derivable*, written formally as: *forward-derivable*$(B \rightarrow A) \Leftrightarrow (A \rightarrow B) \vee \exists X_1, X_2, \ldots X_k((X_1 = B) \wedge (X_k = A) \wedge \forall 1 \leq i \leq k-1(X_i \rightarrow X_{(i+1)}) \wedge \forall 1 \leq i \leq k, 1 \leq j \leq k(X_i \neq X_j) \wedge \exists 1 \leq i \leq k-1(X_i \succ X_{(i+1)}))$.

Since the user is aware of the forward dependency, the reverse dependency from which it is derived does not come as a surprise. The notion that behaviors with non-forward derivable reverse dependencies are confusing is best illustrated through three examples. The first two

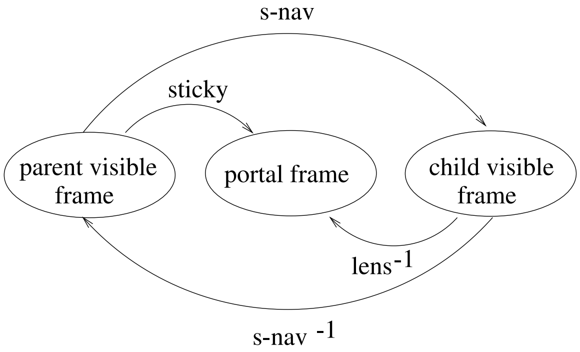

Figure 8: A graph of DataSplash Behavior 3, which has all reverse dependencies derivable.

examples describe intuitive behaviors with reverse dependencies that are forward derivable via inversion and transitivity, respectively. The third example illustrates a confusing behavior with a reverse dependency that is not forward derivable.

First, we describe an intuitive behavior with a reverse dependency that is forward derivable via inversion. Consider a behavior with both the lens and lens^{-1} dependencies enabled. Clearly, the lens^{-1} dependency is derivable from the lens dependency via inversion (Section 3.3.2). In this behavior, repositioning the portal automatically pans the child canvas inside (as with a movable magnifying glass). If the user manually pans the child canvas, the portal is automatically repositioned appropriately. Figure 3 illustrates an example display before and after performing either of these operations to a movable magnifying glass in a map visualization. This is an intuitive behavior because changing what region is displayed inside the magnifying glass automatically repositions the magnifying glass over the corresponding region of the map.

We now describe an intuitive behavior with a reverse dependency that is forward derivable via transitivity. Consider the behavior whose dependency graph is illustrated in Figure 8 having the sticky, s-nav, s-nav^{-1}, and lens^{-1} dependencies enabled. The lens^{-1} dependency is derivable from the s-nav^{-1} and sticky dependencies via transitivity (Section 3.3.1). In this behavior, panning the parent canvas automatically moves the portal frame and pans the child canvas (as with a coordinated view). In addition, panning the child canvas automatically pans the parent canvas and moves the portal frame in the same manner as if the parent had been panned manually. Intuitively, panning the child produces the same effect as panning the parent (via the s-nav^{-1} dependency), which among other things moves the portal frame (via the sticky dependency). Figure 9 illustrates the display before and after repositioning a coordinated view tool (top) and panning left in either the parent or child canvas (bottom).

Now that we have presented two examples of intuitive behaviors with forward derivable reverse dependencies, we give an example of a confusing behavior with a reverse dependency that is not forward derivable. Consider the behavior with only the lens^{-1} dependency enabled. Repositioning the portal has no effect on the child canvas inside (as with a visual bookmark). However, panning the child canvas automatically moves the portal. Figure 10 illustrates the display before and after repositioning the portal (top) and panning the child canvas to the right (bottom). This is effect is counter-intuitive. The user has no reason to expect panning the child canvas to have any reverse effects like moving the portal, because the lens^{-1} dependency is not forward derivable.

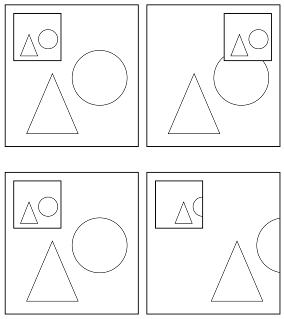

Figure 9: Two sequences of snapshots of a visualization having a coordinated view with the sticky, s-nav, s-nav^{-1}, and lens^{-1} dependencies enabled. The top two snapshots illustrate the display before and after repositioning the coordinated view portal. The bottom two snapshots illustrate the display before and after panning left in either the parent or the child canvas.

Since behaviors with underivable reverse dependencies are confusing, we propose the following rule:

Only Forward Derivable Reverse Dependencies Rule: $\forall X, Y((X \succ Y) \wedge \neg$***forward-derivable***$(Y \rightarrow X) \Rightarrow \neg(Y \rightarrow X))$

5 Example Setting

Applying the usability rules presented in Section 4 reduces the size of the behavior space from 125 potential behaviors to 32 intuitive behaviors. We omit a detailed discussion of all intuitive behaviors. Not all of these behaviors are appropriate in every setting. In this section we describe additional rules to further reduce the set of behaviors for a visualization environment that permits the user to edit any frame and to visually program any dependency mapping.

Given a mapping M, user edits that alter M can be thought of as *programming* M, and edits that do not alter M can be thought of as *browsing* with respect to M. Consider two frames A and B with the dependency $A \rightarrow B$ enabled using mapping M, and no other dependencies among frames. Editing frame B does not affect frame A, and thus programs the relationship between frames A and B: mapping M. On the other hand, editing frame A causes frame B to be changed according to the mapping and does not alter the mapping. Therefore, editing A is considered browsing with respect to M.

For example, consider a magnifying glass with the lens dependency enabled and the lens^{-1} dependency disabled. A dependency mapping for the lens dependency

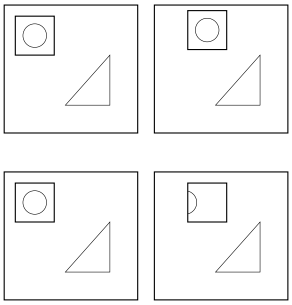

Figure 10: Two sequences of snapshots of a visualization having a portal with only the lens^{-1} dependencies enabled, which is not forward derivable. The top two snapshots illustrate the display before and after repositioning the coordinated view portal. The bottom two snapshots illustrate the display before and after panning the child canvas to the right.

might cause the child frame to be automatically enlarged when the user enlarges the portal frame. The ratio of the portal frame to parent frame sizes is a parameter in the lens dependency mapping that controls the magnification factor, as discussed in Section 3.3. Since the lens^{-1} dependency is disabled, editing the child frame does not affect the portal frame, and thus programs the magnification factor.

In the fully programmable setting, we would like every dependency to be *programmable*, meaning that there is a way for the user to program its dependency mapping. A dependency is programmable if the dependent frame is editable and the inverse dependency is disabled. Enabling the inverse dependency would cause the two frames to be co-dependent, so that no edits could alter the relationship between frames. Note that although inverse dependencies are undesirable in this setting, they are useful in other settings with browse-only characteristics.

It is also desirable for all dependencies to be *independently programmable*, meaning that programming its mapping does not also program any other mapping. In mapping transitive environments (which we consider in this paper as discussed in Section 3.3), a dependency $A \rightarrow B$ is independently programmable if all mappings for dependencies pointing to B are derivable via transitivity from each other. In this case, all dependency mappings pointing to B are really just incarnations of the same mapping, which can be programmed independently of other mappings that do not point to B. The formal rule for enforcing independent programmability of a dependency $A \rightarrow B$ is:

Independent Programmability Rule for $A \rightarrow B$: $(A \rightarrow B) \Rightarrow \mathit{editable}(B) \wedge \neg(B \rightarrow A) \wedge \forall X((X \rightarrow B) \Rightarrow (X \rightsquigarrow A) \vee (A \rightsquigarrow X))$

To obtain a reduced set of behaviors for a fully editable and independently programmable environment, we start by applying our usability rules suggested in Section 4, which reduces the size of the behavior space to 32 intuitive behaviors. To further reduce the set of allowed behaviors in this setting, we start by asserting that all frames are editable, which results in 11 behaviors. Then, we apply the Independent Programmability rule to each pair of frames, which leaves the following set of five behaviors that are intuitive and fully editable and have all dependencies independently programmable:

1. Dependencies: {}
   Editable: {parent, portal, child}
2. Dependencies: {sticky}
   Editable: {parent, portal, child}
3. Dependencies: {sticky, s-nav}
   Editable: {parent, portal, child}
4. Dependencies: {lens}
   Editable: {parent, portal, child}
5. Dependencies: {sticky, s-nav, lens}
   Editable: {parent, portal, child}

In this reduced set of behaviors, all dependencies are uni-directional since this setting requires independent programmability of all dependencies, and they are forward since our usability rules do not permit underivable reverse dependencies. Our usability rules also eliminate the behaviors {s-nav} and {s-nav, lens}, which both exhibit bypassing, and {sticky, lens}, which violates dependency transitivity. We now turn to a discussion of the five allowed behaviors.

Behavior 1 has no dependencies, and can be used to implement visual hyperlinks (Section 2.1). Behavior 2 can be used to construct bookmarks (Section 2.2).

Coordinated views (Section 2.3) and overview tools (Section 2.4) use Behavior 3. The sticky dependency ensures that the portal frame is always in the same position relative to the parent visible frame, *i.e.*, is fixed on the screen. Using the s-nav dependency with the identity mapping, the child visible frame always shows the same set of objects as the parent visible frame, giving rise to a coordinated view. Alternatively, using a different s-nav dependency mapping that causes the child visible frame to be larger than the parent visible frame, we have an overview tool.

Finally, Behaviors 4 and 5 can be used to construct movable filters (Section 2.5) and movable magnifying glasses (Section 2.6). The lens dependency maps changes to the portal frame to changes to the child visible frame so that moving or resizing the portal changes which part of the child canvas is filtered or magnified. Using Behavior 4, the filter or magnifying glass is canvas-stationary, since its size and position is static relative to the parent canvas, independent of navigation. Alternatively, using Behavior 5, the filter or magnifying glass is user-stationary, so it remains fixed on the screen.

Each of these behaviors is fully and independently programmable. To see this, consider the most restrictive case of Behavior 5, where all three forward dependencies are enabled. The sticky mapping is programmed by editing the portal frame, and the lens mapping is programmed by editing the child visible frame. Since mapping transitivity is employed, the s-nav mapping is defined as the composition of the sticky and lens mappings and is not a separate mapping available for programming. Therefore, Behavior 5 is fully and independently programmable, as are the other behaviors with fewer dependencies.

6 Case Study: DataSplash

The authors of this paper were involved in the DataSplash prototype implementation [12] as part of the Tioga project at UC Berkeley [14]. The DataSplash prototype was designed and implemented before the analysis of this paper was conducted. Therefore, it does not conform to all of the rules outlined here. We discuss the behaviors allowed in DataSplash, and show that in cases where the rules in the paper were not followed, behaviors can be undesirable.

DataSplash employs mapping transitivity and mapping inversion, and allows the following set of six behaviors, the last two of which violate dependency transitivity:

1. Dependencies: {}
   Editable: {parent, portal, child}
2. Dependencies: {sticky}
   Editable: {parent, portal, child}
3. Dependencies: {sticky, s-nav, s-nav^{-1}, lens^{-1}}
   Editable: {parent, portal, child}
4. Dependencies: {lens}
   Editable: {parent, portal, child}
5. Dependencies: {sticky, lens}
   Editable: {parent, portal, child}
6. Dependencies: {sticky, s-nav, s-nav^{-1}, lens, lens^{-1}}
   Editable: {parent, portal, child}

Behaviors 1, 2, and 4 are the same as Behaviors 1, 2, and 4 in the fully programmable setting (see Section 5).

Behavior 3, illustrated in Figure 8, was discussed in Section 4.3 and has two forward derivable reverse dependencies. It is similar to Behavior 3 in the fully programmable setting and can be used to construct coordinated views and overviews that have an independently programmable sticky dependency. However, the s-nav dependency mapping, *e.g.*, the demagnification factor of an overview tool, is not programmable since the parent and child frames are co-dependent.

Behavior 5 is inconsistent and confusing because it violates the Dependency Transitivity rule, which was recommended for usability in Section 4.1. The user can edit the portal, causing the child to automatically navigate. However, if the user pans or zooms the parent, the portal follows the user, but the child does not automatically navigate. Another way to see how this is problematic is that the dependency mapping initially set up between the portal frame and child frame (*e.g.*, making them the same for a movable filter) is destroyed when the user navigates the parent. Our experience with the DataSplash prototype confirms this lack of usability. Behavior 5 was never used.

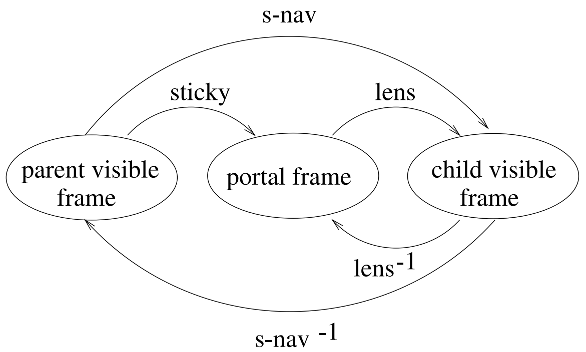

Figure 11: A graph of DataSplash Behavior 6, which violates dependency transitivity.

Behavior 6, illustrated in Figure 11, also violates dependency transitivity. Since editing the portal frame automatically changes the child frame but does not also change the parent frame, it does not make sense that editing the child frame should change the parent frame. For example, consider a magnifying glass. Moving the magnifying glass around has the expected effect: it causes a magnification of the region of the canvas underneath the repositioned magnifying glass to appear in the magnifying glass. Panning the magnified region not only automatically repositions the magnifying glass, but also pans the parent. The latter effect can be surprising and confusing.

One way to convert Behavior 6 into a less confusing behavior is to remove the s-nav^{-1} dependency. This more intuitive behavior can be used for movable magnifying glasses whose lens dependency is not programmable. When the user pans the child inside the magnifying glass, the magnifying glass moves so as to remain correctly positioned relative to the parent canvas. Similarly, when the user zooms the child inside the magnifying glass, the magnifying glass automatically resizes to maintain the same magnification factor.

In practice, Behavior 6 was effectively converted into a less confusing behavior in another way. While experimenting with the DataSplash prototype, we never edited the child frame directly for portals using Behavior 6 because it was confusing. Thus, child frame editability was effectively disabled. Notice that if we remove child editability from Behavior 6, the s-nav^{-1} and lens^{-1} dependencies become meaningless and are removed, and the behavior no longer violates transitivity and is similar to fully programmable Behavior 5 (see Section 5), but with child editability disabled.

7 Summary and Future Work

In this paper, we presented a model for portals and fundamental binary properties that govern their behavior and gave examples of useful tools that come in many varieties with different properties. Since the space of possible behaviors is quite large, we introduced rules that can be applied to eliminate confusing behaviors. We then suggested additional rules to apply to eliminate inappropriate behaviors for an example setting and examined the resulting reduced set of behaviors. Finally, we analyzed the set of behaviors available in a real visualization environment, focusing on behaviors that violate our rules and the resulting adverse effects.

There are numerous avenues for future work in this area. First, we plan to explore models for dependency mappings. Some models for displaying portals treat the portal frame as a physical window, where moving closer enlarges the visible area of the child. With other models, portals are like hanging pictures rather than windows in this respect.

In addition, we plan to consider environments that permit partial editability (*e.g.*, where panning is permitted but not zooming), and partial programming of dependencies (*e.g.*, where zooming the child of a magnifying glass changes the magnification, but panning does not change the magnified region). Furthermore, we plan to consider environments where frame editability and dependency programmability are temporary states, rather than static properties of a behavior.

Some environments support *replicated portals* [2, 5] (called "splash" portals in DataSplash [11, 12]). Portals can be automatically replicated, one for each data item in a data set. For example, Figure 1 shows a map visualization of the U.S. states (the filled polygons) in which a replicated portal displays a bar chart for each major city. In some cases, it may be desirable to have dependencies between child frames of replicated portals, so that navigating one portal automatically navigates others. We plan to study the interaction of dependencies between child frames and the dependencies discussed in this paper, such as the s-nav dependency.

Finally, since portals are so complex, more work remains to be done on intuitive ways to convey to the user the behavior and dependency mappings of each portal.

Acknowledgments

We thank the other members of the DataSplash research group at UC Berkeley: Alexander Aiken, Michael Chu, Vuk Ercegovac, Mark Lin, Mybrid Spalding, and Michael Stonebraker. They provided many useful discussions. We also thank Ed Chi and Jock Mackinlay for helpful feedback.

References

[1] C. Ahlberg and E. Wistrand. IVEE: An information visualization and exploration environment. In *Proceedings of the First Information Visualization Symposium (InfoVis '95)*, Atlanta, Georgia, October 1995.

[2] A. Aiken, J. Chen, M. Stonebraker, and A. Woodruff. Tioga-2: A direct manipulation database visualization environment. In *Proceedings of the 12th International Conference on Data Engineering*, pages 208–217, New Orleans, Louisiana, February 1996.

[3] B. B. Bederson, J. D. Hollan, K. Perlin, J. Meyer, D. Bacon, and G. W. Furnas. Pad++: A zoomable graphical sketchpad for exploring alternate interface physics. In *Journal of Visual Languages and Computing*, volume 7:1, pages 3–31, March 1996.

[4] B. B. Bederson and B. McAlister. Jazz: An extensible 2D+ zooming graphics toolkit in java. Technical report, University of Maryland, May 1999.

[5] Belmont. CrossGraphs: Multidimensional graphical reporting and data visualization. White paper, Belmont Research, Inc., 2000.

[6] E. A. Bier, M. C. Stone, K. Pier, W. Buxton, and T. DeRose. Toolglass and magic lenses: The see-through interface. In *Proceedings of the ACM SIGGRAPH Computer Graphics Annual Conference Series*, pages 73–80, Anaheim, California, August 1993.

[7] S. K. Card, J. D. Mackinlay, and B. Shneiderman. *Information Visualization: Using Vision to Think*, pages 285–286. Morgan-Kaufmann, San Francisco, California, 1998.

[8] M. Derthick, J. A. Kolojejchick, and S. F. Roth. An interactive visualization environment for data exploration. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, pages 2–9, Newport Beach, California, August 1997.

[9] M. Livny, R. Ramakrishnan, K. Beyer, G. Chen, D. Donjerkovic, S. Lawande, J. Myllymaki, and K. Wenger. DEVise: Integrated querying and visual exploration of large datasets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 301–312, Tucson, Arizona, May 1997.

[10] C. North and B. Shneiderman. Snap-together visualization: A user interface for coordinating visualizations of a relational database. In *Proceedings of the 5th International Working Conference on Advanced Visual Interfaces (AVI 2000)*, Palermo, Italy, May 2000.

[11] C. Olston, M. Stonebraker, A. Aiken, and J. M. Hellerstein. VIQING: Visual Interactive QueryING. In *Proceedings of the 14th IEEE Symposium on Visual Languages*, pages 162–169, Halifax, Canada, September 1998.

[12] C. Olston, A. Woodruff, A. Aiken, M. Chu, V. Ercegovac, M. Lin, M. Spalding, and M. Stonebraker. DataSplash. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 550–552, Seattle, Washington, June 1998.

[13] K. Perlin and D. Fox. Pad: An alternative approach to the computer interface. In *Proceedings of the 20th International Conference on Computer Graphics and Interactive Techniques*, pages 57–64, Anaheim, California, August 1993.

[14] M. Spalding and A. Woodruff. DataSplash: A database visualization environment developed by the UC Berkeley Tioga project, 1998. http://datasplash.cs.berkeley.edu.

[15] M. C. Stone, K. Fishkin, and E. A. Bier. The movable filter as a user interface tool. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 306–312, Boston, Massachusetts, April 1994.

[16] A. Woodruff, A. Su, M. Stonebraker, C. Paxson, J. Chen, A. Aiken, P. Wisnovsky, and C. Taylor. Navigation and coordination primitives for multidimensional browsers. In *Proceedings of the 3rd IFIP 2.6 Working Conference on Visual Database Systems*, pages 360–371, Lausanne, Switzerland, March 1995.