

Breaking the Window Hierarchy to Visualize UI Interconnections

David F. Huynh, Robert C. Miller, David R. Karger
MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, MA, USA
{dfhuynh, rcm, karger}@csail.mit.edu

ABSTRACT

Among three traditionally fundamental characteristics of contemporary windowing systems—opacity, rectangularity, and hierarchy—the first two have been broken to give way to more expressive UI designs while hierarchy remains unchallenged. We propose a new kind of UI element called links that breaks the hierarchy of graphical user interfaces for the purpose of showing relationships between disparate UI elements. We present a small user study indicating that links are desirable visual elements that are currently under-used in graphical user interfaces. We propose a taxonomy of how links can be used, present a toolkit for representing and displaying links in Java/Swing, and show how links can enhance existing applications' UIs.

INTRODUCTION

Windowed user interfaces currently dominate if not monopolize all existing software applications. Users, UI programmers, and even UI researchers have come to accept that an application's graphical user interface is a tree of large and small, nested, mostly rectangular and opaque windows, overlapping and hierarchically clipped. These characteristics of contemporary windowing systems—rectangularity, opacity, and hierarchy—made sense back in the early days of graphical user interfaces when they bought us performance optimizations that allowed for smooth user interactions. Now, after many years of hardware advances, these optimizations become less and less justified against the demand for expressiveness in UI design.

Although the traditional windowing paradigm, consisting of opaque, rectangular windows with impenetrable borders, is conceptually simple for both programmers and users, there is mounting evidence that it is too confining. UIST has seen continuing research interest in translucency (e.g., [1], [6], and [8]) as researchers recognize the human ability to parse layered visual content. Non-rectangular UIs also have arrived in the form of pie menus and arbitrarily shaped media player skins. A recent piece of work [11] explores a circular radar-like interface for showing notifications.

Support for translucency and non-rectangularity has made its way into some windowing systems and is gaining traction.



We have witnessed the introduction of translucent windows in popular media players and instant messaging clients. However, hierarchy remains mostly untouched. It is this third characteristic of the traditional windowing paradigm that we wish to address. This paper explores the concept and usage of *links*—a new kind of UI element that crosses from one part of the hierarchical window tree to another part, expressing connections between disparate elements on screen.

We start by redesigning an existing UI to show new possibilities in UI designs when rigid windows are not used. In particular, the redesign illustrates the replacement of selection synchronization by the use of links crossing from one window to another.

Next, we show the results of a user study pilot in which test subjects drew links abundantly when told to design posters but hardly any when designing GUIs for the same purpose. This is evidence for the desirability of links and the difficulty in which they can be programmed in existing UI systems.

We propose a taxonomy of links to understand the nature of links. We then build a prototypical library of link UI elements that can be integrated into Java/Swing applications. We use this library to implement a version of the aforementioned redesign as well as an enhancement to an existing application through the use of links.

Finally, we discuss related work and future work.

SNAP: A REDESIGN EXAMPLE

In order to illustrate how the aforementioned three fundamental characteristics of the traditional windowing paradigm can be usefully broken, we consider SNAP, a recently developed research user interface [5]. This research system called SNAP provides users with a UI mechanism for coordinating several visualizations that have been rendered from queries to a relational database.

Figure 1 shows a sample UI composed using SNAP. A user loads each of the five windows with data and specifies the formats for rendering them (e.g., plot, map, table, outline). Then, the selections of the windows are tied together using the Snap buttons so that selecting a state in the top left window highlights the corresponding data point in the plot, underlines the state's code in the map, and displays that state's county data in the table and the treemap on the right.

We chose SNAP as an example because of its attempt to let users smash together several visualizations and explore

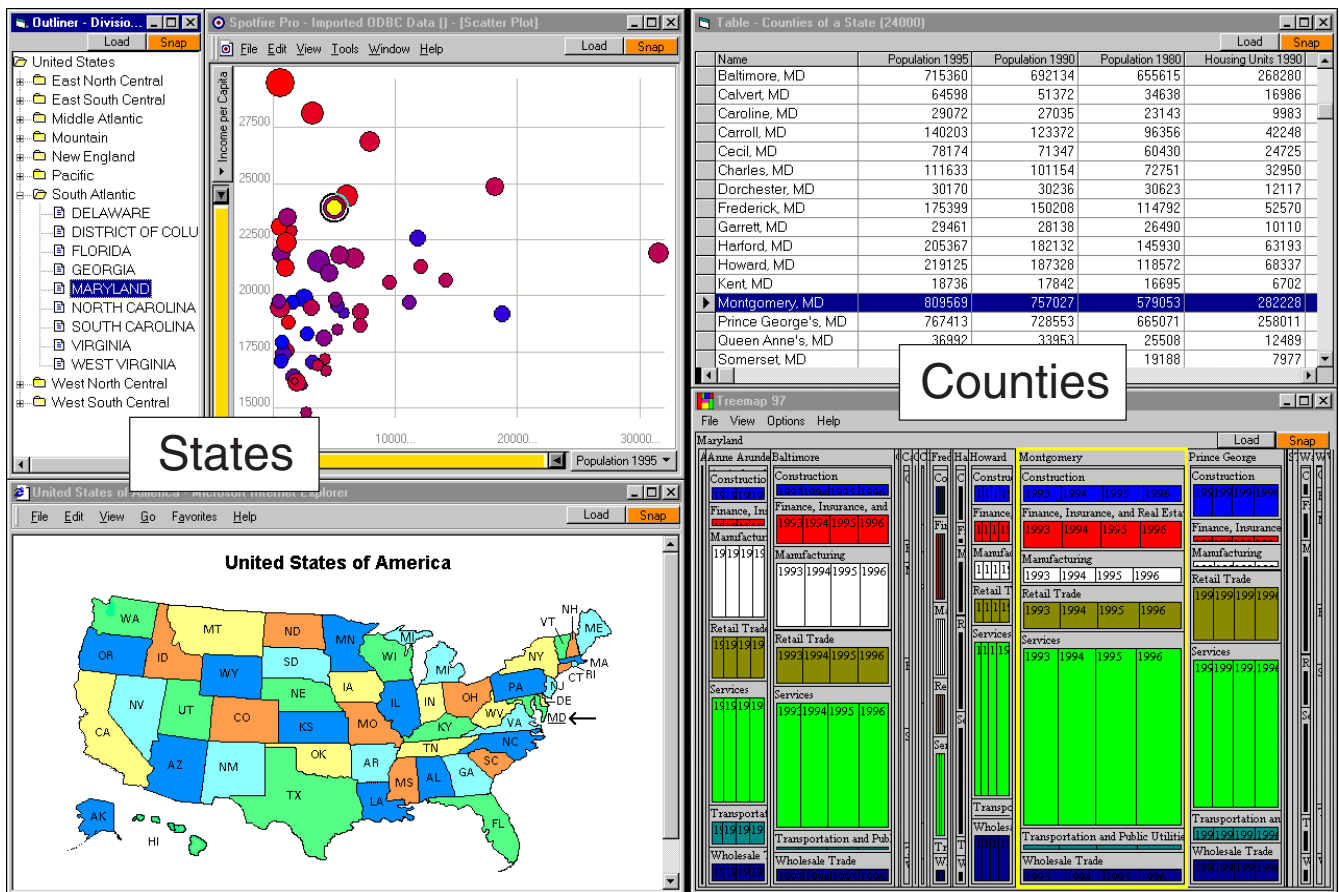


Figure 1. Original SNAP user interface from [North], reprinted with permission from authors

relationships among them. However, the “smashing” is not as effective as it could have been if windowing toolkits were less confining. Figure 2 shows a rough redesign of the UI. We focus primarily on the information being explored rather than on the UI mechanisms provided to the user to specify the visualizations. Note the following differences in Figure 2:

- We use lines to connect the selected state name (e.g., Maryland) to the corresponding state on the map as well as the data point on the plot. We believe that this use of connecting lines, or “links,” shows correspondence more effectively than just highlighting corresponding items in the different visualizations in synchrony.
- Furthermore, since links are used to indicate selection, visual variables formerly used for highlighting are now freed up for other purposes. For example, highlighting can be used to indicate data points on the plot corresponding to other states in the same region as the selected state. This is useful for an overview of how the states in a region are distributed.
- We also use a link to connect the selected state to the county data table. This connection between the state Maryland and its counties eliminates a need to suffix every county name with “MD” as in the original design.

[TODO: improve this point] Using most existing UI toolkits, it is difficult to draw lines stretching from within one window to another window. Even if it is possible, the resulting links are not first-class widgets and cannot be easily manipulated programmatically.

This paper focuses on links, but we note in passing that another UI characteristic—rectangularity—has been usefully broken in our redesign:

- Most borders are removed. This change allows the plot to be pushed into the map (crossing over its invisible rectangular boundary) without obscuring parts of the map or creating a busy look due to intersecting borders. The two pieces of information are fitted together more snugly. This fitting effectively communicates the relationship between the map and the plot in Figure 2. In contrast, two labels “States” and “Counties” had to be added to the screenshot in Figure 1 to avoid confusion: the plot could have been of county data rather than of state data.
- The column header labels in the table are inclined so that the columns can be pushed closer together. The ability to incline text, difficult in most UI toolkits, adds flexibility to the design of visualizations. Furthermore, note that the rectilinear bounding boxes of the column header labels intersect one another. As a result, providing a custom header label widget to a standard table widget is not enough—the table widget needs to be able to nudge the header labels closer than a conventional rectilinear layout would allow.

USER STUDY

These interesting axes of the design space—rectangularity, opacity, and hierarchy—remain largely under-explored because of limitations imposed by UI toolkits. In order to get a sense for how the creativity of UI programmers is affected by their perception of what’s easy or hard to implement, we

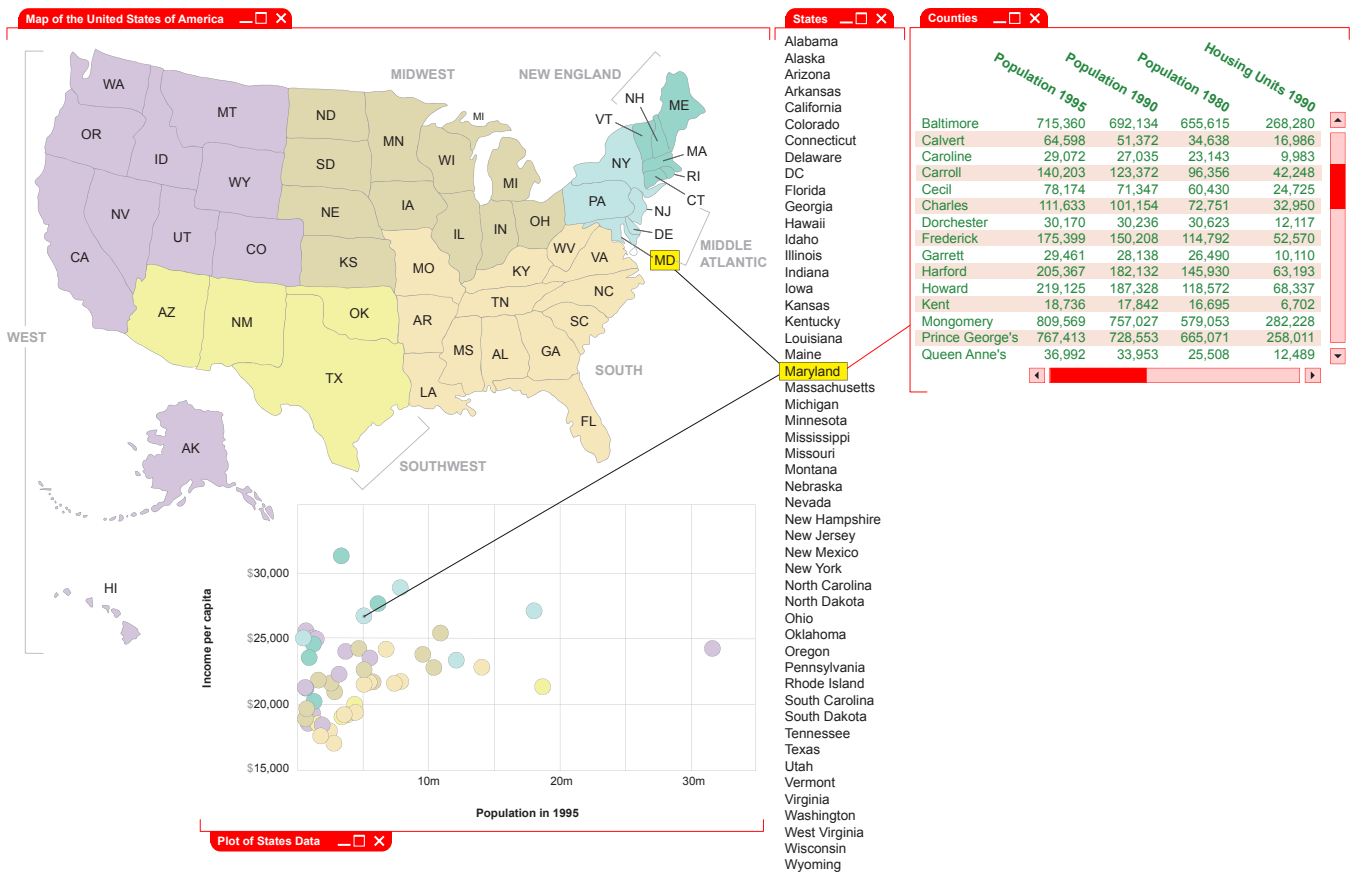


Figure 2. A redesign of the SNAP user interface

conducted a small study with 10 computer science graduate students in our lab. Each student was asked to design an information visualization, either in poster form or computer form, for the following scenario:

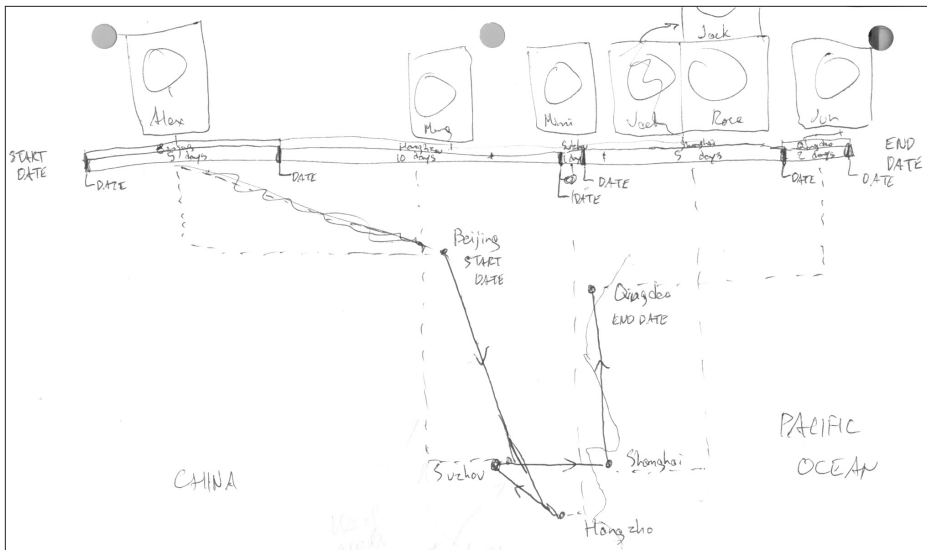
“You have recently gone on a trip to China. During your trip, you visited five cities and stayed a different number of days in each city, and you also made new friends at each city. [Here appeared the list of cities, durations of stay, and pictures of one or two friends made in each city.] As an assignment in your Chinese language class, you are to make a [poster or computer program] to recount your trip to your classmates, which shows (1) your route through China, (2) the friends you met, and (3) the fact that you spent a lot more time in Hangzhou than in any other city. Using paper and pencil, roughly sketch out what your [poster or computer program] would look like. Assume that you have access to a map of China and photos of your friends in any form you’d find convenient, and that the hypothetical class assignment is due in one week.”

We chose this scenario because its solution requires displaying several different visualizations—geographical, chronological, and pictorial—that are closely related. The Hangzhou requirement was particularly intended to motivate creative ways of showing the relationships between visualizations (like Minard’s famous visualization of Napoleon’s Russia campaign [9]). We wanted to see whether there were any systematic differences in how these relationships would be shown in a

paper display (a poster) compared to an interactive display (a computer program).

For 4 of the 10 designers (randomly chosen), the requested visualization was a poster; the remaining 6 designers were asked to design a computer program. All 10 designers had designed and implemented at least one substantial user interface, and had experience with a wide range of UI tools (chiefly Java Swing, HTML, SWT, and Visual Basic). Designers in the computer-program condition were told that they could imagine designing for any UI programming environment that they felt comfortable with. Designers were given as much time as they wanted to produce a design; in the end, all designers took less than 30 minutes. Three designers (2 in the poster group and 1 in the computer group) iterated their designs, throwing away a partial design and presenting a second sketch instead. The other designers created only one design.

The final designs were varied, but some general conclusions can be drawn. First, lines connecting different visualizations were commonly found in poster designs (3 out of 4) but rarely in computer designs (only 1 out of 6). The person who used lines in her computer program design admitted not thinking ahead about what tools to use and then claimed that she could implement her design in Java/Swing. Lines were used to connect cities on the map to friends’ photos (in 2 designs), and cities to points on a time line (2 designs). Several solutions used two kinds of lines, one kind to show the route around the map (a chronology within the geographic visualization), and another kind to make links between visualizations.



###Unreferenced### Figure 3. One of the poster de-



###Unreferenced### Figure 4. One of the computer program designs from the user study

Again, although this paper focuses primarily on links, we would like to point out that rectangularity is observed to be usefully broken in the user study. Most of the poster designs (the same 3 out of 4) were unified in the sense demonstrated by Figure 2: different visualizations freely overlapped without respecting rectangular boundaries. In most of the computer designs (5 out of 6), by contrast, the visualizations were walled off from each other by strongly drawn rectangular boundaries. In fact, half the computer designs (3 out of 6) put different visualizations on different pages, giving up any hope of unifying them visually.

Certainly, there are many differences between the poster medium and the computer medium that might influence how designers perceive the solution space. For example, a typical poster has more display area than a typical computer screen, while a typical computer program is more interactive and more dynamic (i.e., displaying changing information). But

it's worth noting that effective design idioms that were perfectly natural in poster designs—such as connecting lines and nonrectangular information elements—largely failed to appear in the computer designs.

TAXONOMY OF LINKS

Our user study's results indicate frequent use of links on paper but not in computer programs. We attribute this difference to the difficulty with which links can be programmed in current UI systems. In order to understand how links can be adapted to visualizations on the computer, we first explore the general diversity of links in information visualizations by proposing a taxonomy for line and link usage (Table 1). At the top

level, we distinguish between the use of lines for showing associations and for showing information applicable globally to a visualization, such as the scale of a map. Lines used associatively are what we have termed "links" previously. Links are the topic of our discussion here.

We propose two main categories of links:

Elaborative Links

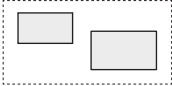

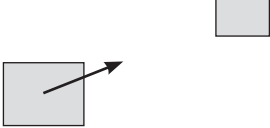
These are links that elaborate on the information being visualized by adding more information which is not previously evident from the visualization; they connect multiple UI elements so that the user is informed of the relationships between those elements. There are several types of elaborative links: correspondence links, equivalence links, succession links, grouping links, and links for showing general relationships. Note that grouping links are an exception in that they may not have ends terminating at the objects being grouped.

Emphasis Links

When the connections between objects are already presented but can be enhanced visually, we use emphasis links. Alignments of several objects are usually indicated by dashed lines. Distances between objects can be indicated with a ruler stretching from one object to the other. An arrow shows the direction from one object to another without touching the destination.

This is not a dichotomy. Rather, these two categories are more like two axes of characteristics onto which a link can be classified.

Note that a link can connect more than two objects to show an n-ary relationship. A link can also connect to another link: Figure 5 shows such a link being used to show the mutual friends of two persons A and B. In addition to this taxonomy, we also consider degenerated links attached to only one object: these act like tooltips in annotating the objects being pointed to (see accompanying videos).

Associative <i>showing association between different information items</i>	
Elaborative <i>the association is not already present in the visualization and is elaborated by the lines</i>	
Correspondence	<i>links between two or more views of a single logical object (e.g., three views of a state are linked together in our redesign example)</i>
Equivalence	<i>links connecting elements having common or similar attributes, e.g., a contour line indicating points of equal elevation</i>
Succession	<i>e.g., arrows that show transitions through a flowchart or connect points in a time series</i>
Grouping	<i>e.g., contour enclosing grouped items</i>
	
General Relationship	<i>links between an object or a relationship to several related objects (e.g., a state is linked to its county data in our redesign example)</i>
Emphasis <i>the association is already in the visualization and is emphasized by the lines</i>	
Alignment	<i>e.g., a dashed line on a form designer shows that a UI component being dragged is “snapped” to align with another UI component</i>
	
Length	<i>e.g., a line indicating some distance between two visual elements</i>
Direction	<i>e.g., an arrow pointing from one visual element in the direction of another visual element</i>
	
Non-associative <i>showing information applicable globally to the entire visualization</i>	
Length	<i>e.g., the scale ruler on a map</i>
Direction	<i>e.g., the compass on a map</i>

###Unreferenced### Table 1. Taxonomy of Lines

RENDERING ISSUES

Our initial taxonomy for lines, and links, helps us determine the types of links we will explore through actual implementation. In this section, we dive only into elaborative links as emphasis links are generally more useful in graphics editing programs rather than in generic information visualizations. We believe that emphasis links have been explored extensively in the graphics design software domain. We will discuss various challenges to be addressed in order to render links amid the traditional windowing paradigm.

Stems, Ends, and Anchors

Every link (perhaps with the exception of some grouping links) consists of one or more *stems* (e.g. straight or curved lines) and two or more *ends* (e.g., arrowheads). The ends are visually bound to potentially moving or movable *anchors*—UI elements, or parts of, that are identifiable and visually self-

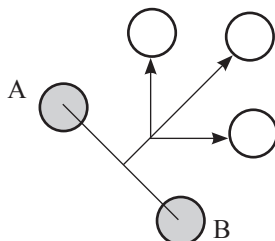


Figure 5. Link showing mutual friends of A and B

cohesive to the user. In other words, the user must be able to understand that, say, the link is attached to the border of a label and that the link will move if the label is moved when its containing window is dragged.

Extending The 2½D Space

Anchors are situated in a 2½D space (i.e., x and y coordinates plus a z-order index). In order to communicate visually the bindings between link ends and anchors, we must also visually situate link ends and link stems in this 2½D space. That is, we have to indicate through the renderings of links where their stems and ends are with respect to other UI elements in this space. Because the many anchors to which a link attaches lie at different z-orders, visually situating the various parts of the link in the 2½D space to present a visually cohesive UI element is non-trivial. This is the main challenge.

Consider the link from x to y in Figure 6. The two ends of the link are attached to anchors at different z-orders. If we keep the 2½D space, we have to break the stem into two segments, one belonging to the same z-order as the UI element C, and one as the UI element D. It is not clear where to break the stem. In fact, it is not even clear that the two segments can share the same z-orders as the elements C and D since no two elements can have the same z-order in this 2½D space. Where, then, should the two segments be in the z-order stack, relative to the intervening UI element B?

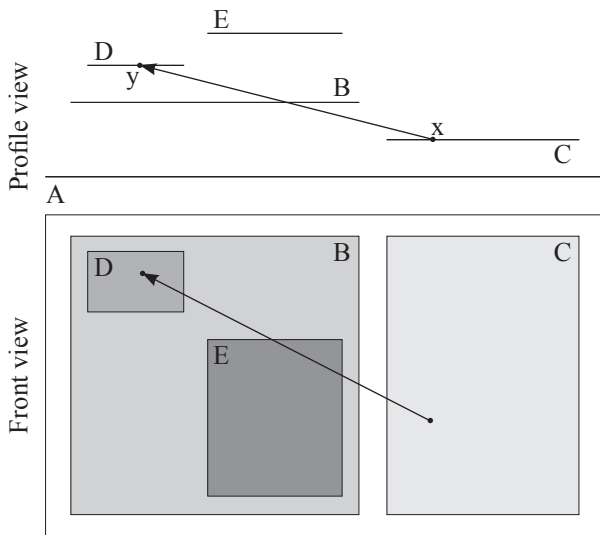


Figure 6. Showing link end obscurity through clipping and ghosting

It would be easier to conceptualize where the link's parts are situated with respect to other UI elements if we usher links into a 3D space in which other UI elements can only take on different, integral z coordinates. In this 3D space, link stems are continuous lines as already shown in the profile view in Figure 6.

Obscurity Problems

The next challenge is to communicate visually the location and distribution of the various parts of the link in the 3D space. A straightforward opaque 2D projection of the link (Figure 7) is not satisfactory: only the originating link end and two unconnected segments of the link stem remain unobscured by other opaque UI elements. In fact, because the two anchors are always at different z-orders, if the link stem is straight, one end

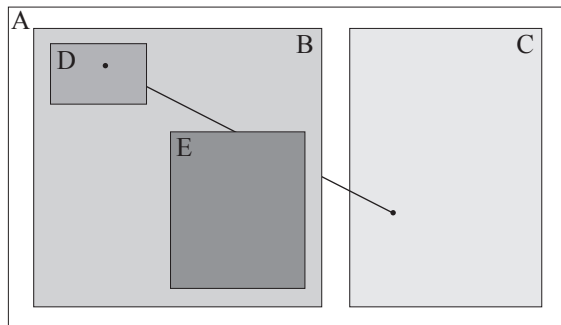


Figure 7. Opaque 2D projection

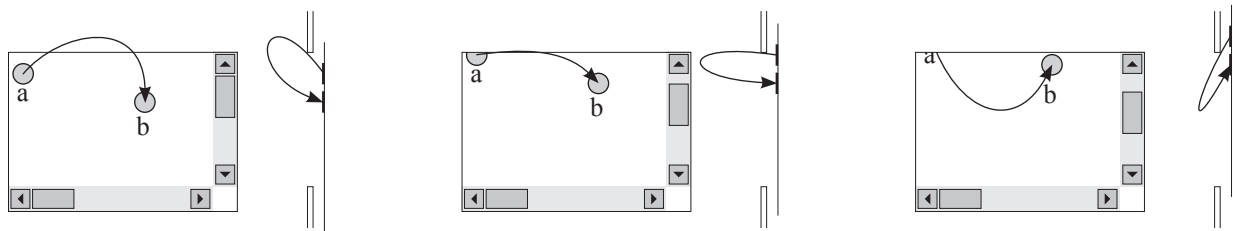


Figure 9. Physical model for link stems protruding from view ports: the link stem is constrained by the view port's upper edge as the view port is scrolled downward

of the link is guaranteed to be hidden behind the upper anchor. This method of rendering links is obviously not useful.

Before we dive into techniques for countering obscurity, we first distinguish two sources of obscurity.

Overlapping A UI element is obscured partially or entirely because another UI element of small z-order overlaps with its region.

Viewport Cropping A UI element can also be obscured because it is being viewed through a view port and it falls outside the viewport's opening partially or entirely.

In order to better present links—countering the obscurity effect resulted from the newly adopted 3D space while maintaining some consistent physical model comprehensible to the user—we consider two techniques: bending and translucency.

Countering Obscurity by Bending

We bend the link in Figure 6 up along a vertical plane as shown in Figure 8 so that it is no longer obscured by B, D, and E.

Note that the user might not want the link to go over E. For example, E might contain information not related at all to the contents of C and D, and the user wants to focus on interacting solely with E. This is usually the case when C, D, and E are separate windows and the user is given the ability to explicitly bring E to the front of the z-order stack. When C, D, and E are child widgets of a common dialog, closely interrelated in their functions, and the user is not allowed to change their z-orders, it makes more sense to bend the link over E.

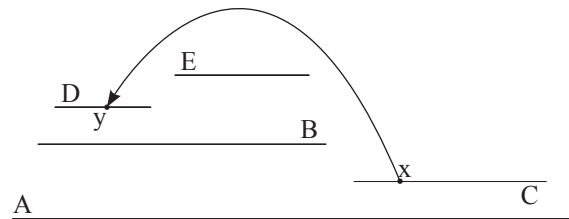


Figure 8. Bending link to counter obscurity

If the link is already curved, it is still beneficial to use the bending technique for, say, providing a smooth transition as the link's anchors are scrolled out of view. Consider the link in Figure 9: as the viewport's opening is scrolled downward, the link stem is pushed against the top border of the viewport. The link stem is bent and its curvature as visible to the user changes, maintaining as much visibility as it can manage.

Countering Obscurity using Translucency

There are cases where bending alone cannot effectively counter obscurity. Consider the link from A to B in Figure 10a: no matter how the link stem is bent, some part of the link is obscured. Even in the adverse bending in Figure 10b, the link end on A is still obscured. In such a case, we resort to making some part of B translucent so that the link end, part of the link stem, and the anchor on A can show through.

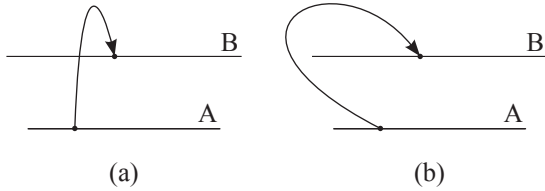


Figure 10. Bending ineffective to counter obscurity

Consider also the case in Figure 11a: B intervenes between A and C, cutting a link between them. The anchor y is entirely obscured by B. However, since C is topmost, it might be desirable to show all the links attached to it regardless of whether they are obscured partially. As in the previous case, no bending can reveal the link end on A. Still, truncating the link stem where it punctures through B might not be satisfactory: the link might seem to the user to terminate on B, or if not, the user might not be able to visualize where the link terminates on A if there were more links between C and A.

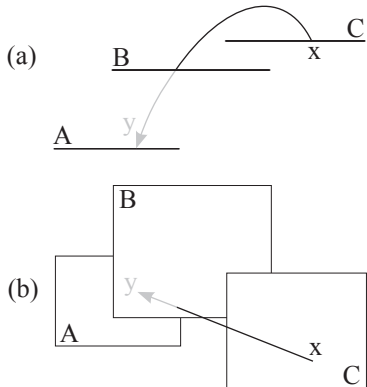


Figure 11. Bending ineffective to counter obscurity by intervening sibling

Note that a UI element obscuring a link (e.g., B in Figure 11a) is not aware of the link that it cuts. Hence, it cannot “volunteer” to be translucent for the purpose of revealing the link. It has to be forced to become partially translucent. Involuntary translucency is one example of how links have demanded more intervention from the UI framework in the rendering of unrelated UI elements. Clipping alone is no longer the only coordination between the renderings of various UI elements.

There are many ways to make a UI element obscuring a link “translucent.”

Alpha Composition We can make it translucent in the conventional way using alpha composition. The challenge here is to isolate a small region of the obscuring UI element to make

translucent as it is not desirable to make the entire element translucent.

Structural Projection We can also select the important structural features of the obscured part of the link and its context—borders of large UI elements around it, visually dominant background patches, identifying text segments, etc.—and show them in a ghosted mode on top of the obscuring UI element.

When Obscurity is Desirable

When part of a link is obscured by a viewport, the viewport’s space might be large and the obscured part of the link might extend far away from the viewport’s opening. Since the viewport’s primary purpose is to crop its contents, any use of translucency, which spills the viewport’s content outside its opening, might not be desirable. In such a case, although we might have to be content with obscurity, we can still use bending to indicate where the obscured part of the link is with respect to the viewport’s opening. Figure 12a shows the physical model of a link bent because one of its end is scrolled out-of-view in a viewport and Figure 12b shows the rendering of the link as it appears to the user.

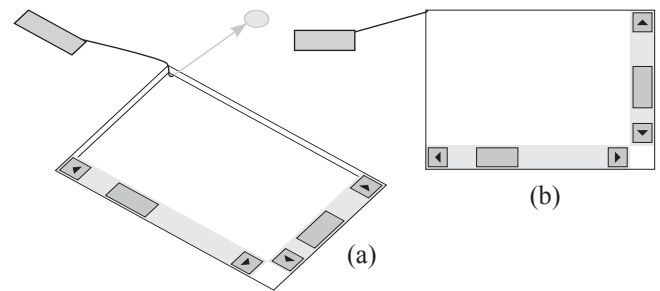


Figure 12. Bending a link by a viewport’s borders

Note that obscurity is not always desirable in the context of a viewport. Figure 13a illustrates a case in which cropping two links originating and ending at UI elements inside a viewport can result in visual confusion. This is fixed in Figure 13b when the links are allowed to spill outside the viewport’s opening.

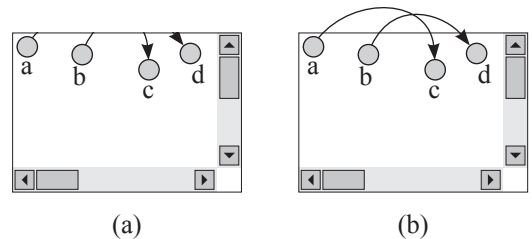


Figure 13. Stem invisibility creates confusion

Discussion

In our discussion of the bending technique, we have focused solely on bending in the z dimension in order to pull links over obscuring UI elements. However, links themselves can obscure other pieces of valuable information being visualized.

In such cases, it is desirable to bend links in the XY plane, rerouting them to avoid important screen regions.

In order for the software component responsible for rendering links to know which screen regions are considered important, the renditions of UI elements need to be annotated with, or lends itself to the computation of, some metric of “information value”—a measure of how much the user values seeing the information in a particular UI element. This metric is useful not only for routing links, but also for implementing involuntary translucency on other UI elements: when the information shown in a particular UI element is so important, we might make it “shine through” other less important UI elements that are obscuring it.

In addition to deriving information value metric, the rendering of links also needs to inspect the visual compositions of UI elements (e.g., lines and shapes that make up their renditions) in order to determine their structural essence for the purpose of structural projection as discussed previously. This inspection is more effectively done on the stroke model than on the pixel model. 2D scenegraph-based UI frameworks [cite??] expose the stroke model readily and this feature of theirs would facilitate the implementation of links.

PROGRAMMING ISSUES

Since links are a whole new type of UI element, they need new programming abstractions. First, while all other UI elements expose writable XY coordinates relative to their parents, links do not—their XY coordinates are managed and cannot be changed by the programmer. Similarly, the z coordinate of a link is also managed: it does not make sense to “bring a link to front” as one can with a window.

Since every link is attached to some anchors and when the anchors move, the link moves, it makes sense to allow the programmer to declare the bindings between the link and the anchors and have a constraint system maintain those bindings. In order to specify the bindings, the programmer must first be able to identify the anchors. This can be done by having UI elements expose a programmatic interface for retrieving anchors by name. Next, the programmer needs some declarative syntax for specifying how the link should be attached to the anchors—from center to center, or between nearest borders, etc. The programmer also needs to place restrictions on the automatic bending of the link as well as on how translucency should be used to counter obscurity. With regard to translucency, the provision of information value metric and structural essence metric is beyond the scope of this paper. [any citation?]

That links are managed UI elements makes them prone to interfere with the rest of the code of the programs in which they are used. For instance, if a link, implemented as a non-rectangular window, is blindly injected into an MDI window to show a connection between two children of the MDI window, it might break a piece of code that iterates over the MDI window’s children to, say, locate the topmost child and consider it the child window in focus. A possible solution to this problem introduces a semantic window hierarchy in addition to the existing syntactic window hierarchy: although the link is a syntactic child of the MDI window (for the purpose of ordering the renderings), it is not a semantic child of the MDI window.

MAGPIE: A TOOLKIT OF LINK UI ELEMENTS

Based on our exploration of various issues on rendering links, we have designed and implemented a toolkit called Magpie for drawing links between different Java/Swing components.

Implementation Technologies

We chose to implement Magpie in Java/Swing as its light-weight components all paint on a common canvas and it is easier to overlay links on them than it is on operating system-native UI widgets. Swing is built on top of the Java 2D Graphics toolkit which offers sophisticated rendering capabilities including alpha composition.

However, at the top level of all Java/Swing UIs are `javax.swing.JFrame`s—operating system-native windows that are independent of one another and do not paint on a common canvas. Consequently, we limit the toolkit to render links spanning only within individual `JFrame`s, not across them. This is a limitation that we will attempt to remove in future versions of the library. Nevertheless, it is still valuable to be able to use such links on, say, `JInternalFrame`s that can be moved about by the user.

Magpie was developed in parallel with our SNAP redesign demo. As such, it was designed to be integrated with an application that uses a `JDesktopPane`. One must be able to incorporate Magpie with minimal alteration to the original program. For this reason, links cannot be Swing components belonging in the same component hierarchy of the original program, as the program’s code might make assumptions about its component hierarchy and may be broken due to the intrusion of links. Furthermore, links can assume arbitrary z-order depending on the z-order of their ends and a link can even have several z-orders. Our solution renders links on the `JGlassPane` of the containing `JFrame` and performs all the necessary clipping to give the illusion of appropriate z-orders.

We needed to augment the painting of the `JGlassPane` of the containing `JFrame`, but in Swing, components are rendered by calls to their `paint()` methods and this mechanism disallows augmenting. Consequently, we had to wrap any existing `JGlassPane` with our own. This is a somewhat disruptive intrusion to client applications, but since `JGlassPanes` are not used often, it might be an acceptable solution.

Rendering links ourselves has a few drawbacks. Links are not first class components and much work will be needed to make them behave like first class components, responding to user inputs. A lot of clipping must also be done that can otherwise be handled by the underlying windowing system.

Magpie API

Following are the interfaces of the Magpie library:

- `edu.mit.csail.magpie.IAnchor`: An anchor is something that can be attached to. There are two kinds of anchors: point anchors and shape anchors. For example, a plot can provide point anchors corresponding to the centers of its data points and a map can provide shape anchors corresponding to the boundaries of its regions. When an anchor is moved or reshaped, it fires event to its listeners and its listeners (e.g., links) move appropriately.

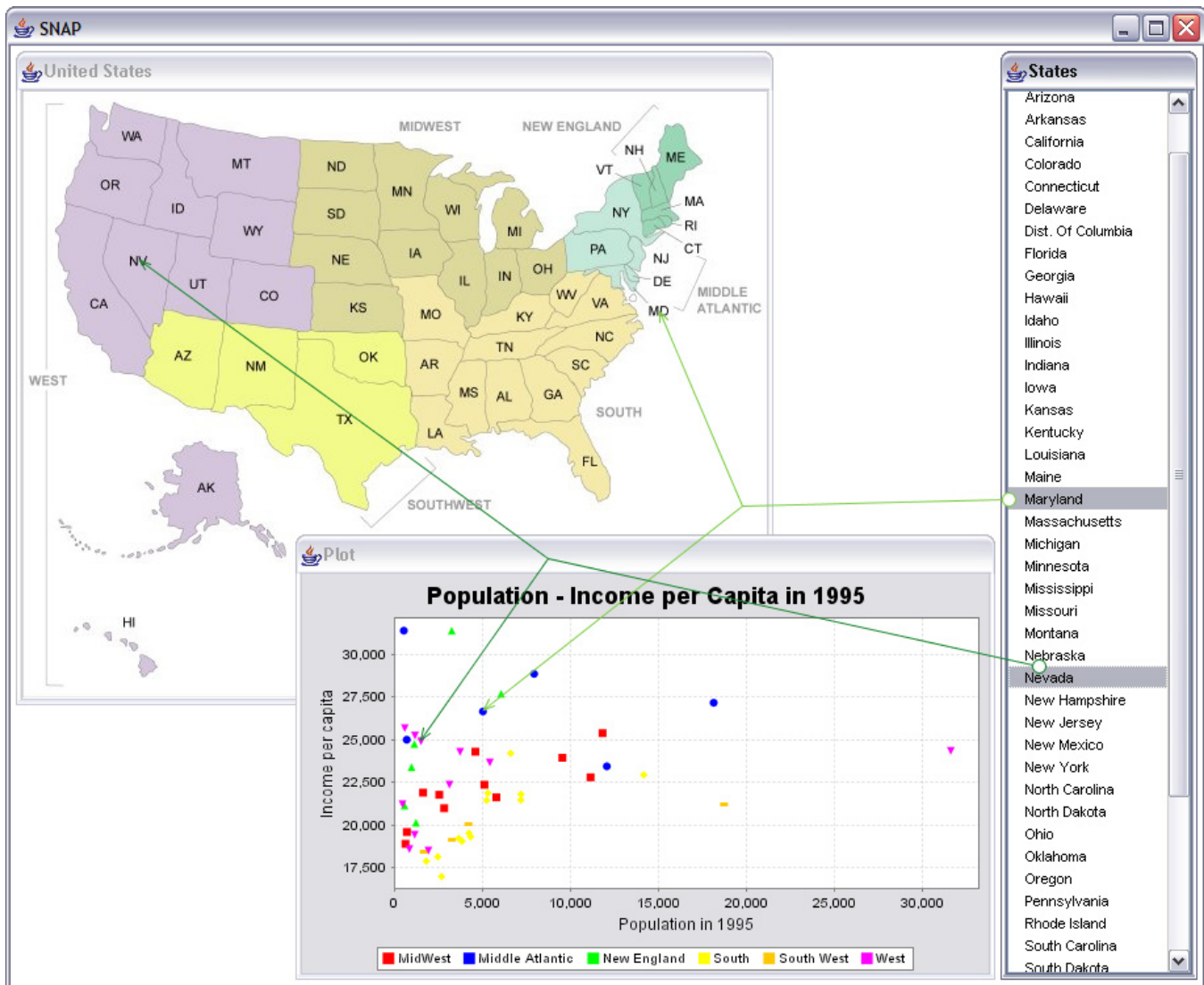


Figure 14. SNAP Redesign demo using the Magpie link toolkit

- `edu.mit.csail.magpie.IConnectable`: A connectable is a UI element that offers zero or more anchors and, hence, can be connected to. Magpie includes connectable wrappers for the several Swing widgets: `JLabel`, `JList`, and `JTable`. The label wrapper exposes a shape anchor for the label's bounding box, while the list and table wrappers expose shape anchors for the items and selections in the list or table. The custom plot and map widgets in the SNAP example (Figure 14) also implement `IConnectable` to provide their own custom anchors.
- `edu.mit.csail.magpie.IAnchorGroup`: Anchors are offered in groups. A connectable `JList` offers a group of shape anchors corresponding to the boundaries of its selected items. Each anchor group has a name, e.g., “`edu.mit.csail.magpie.selection`”. Anchor groups can be retrieved from a connectable. An anchor group fires events whenever anchors are added to or removed from it.
- `edu.mit.csail.magpie.ILink` is the interface for all links. We have implemented several link classes, each taking a different number of anchors and rendering the link differently. `StraightLineBinaryLink` connects exactly two anchors with a straight line. `StraightLineTertiaryLink` connects

three anchors (as shown in Figure 14). `GroupingLink` draws a contour around one or more anchors. Finally, `TooltipLink` takes exactly one anchor and renders a tooltip connected to that anchor.

- `edu.mit.csail.magpie.ILinkEnd`: Link classes are responsible for rendering link stems only and they make use of `ILinkEnd` classes to render link ends. This design choice allows customization of link ends for each type of link. Magpie currently offers circular link ends (seen attached to the state list in Figure 14), arrow heads (which point at the map and plot in Figure 14), and bare ends (seen in Figure 15).

APPLICATIONS

Figure 14 shows the SNAP redesign demo using the Magpie toolkit. This demo preserves the default windowing look and feel in order to illustrate the improvement that links bring about even in the absence of other alterations proposed in the mockup in Figure 2. Multiple selections are now useful because the user can tell exactly which selection in the plot corresponds to which selection in the list—this is not possible if simple highlighting were used.

We also explored adding Magpie links to an existing Java application, LAPIS [3]. LAPIS is a text editor that uses multiple selections for pattern matching, repetitive editing, and find-and-replace. Magpie links were used to address two known usability problems in the LAPIS user interface. First, LAPIS augments the scrollbar with marks showing where selections are located in the document, so that the user can find them more easily when scrolling around. In user studies, however, new users rarely notice the scrollbar marks or guess their purpose. Magpie links drawn from each scrollbar mark to the corresponding selection in the text makes this connection abundantly clear. (In Figure 15, these links point from the scrollbar to the left, into the text pane.)

Second, we used Magpie links to improve a new feature, cluster-based find & replace [4], which rearranges pattern matches into clusters based on similarity in order to reduce the chance of replacement errors. Clustered matches are shown in a separate pane (on the right in Figure 15), using small snapshots of context around each match. User studies showed that for some tasks, this snapshot provided too little information about a match for the user to decide whether it needed to be replaced. Unfortunately, it was hard for the user to find the corresponding match in the document. Magpie links make this simple: each selected match in the cluster pane is linked to a scrollbar mark, which in turn is linked to a selection in the text pane.

Adding these links to LAPIS required less than 100 lines of new code, which mainly exposes anchors representing scrollbar marks and text selections, and then creates links between them. Linking to cluster matches was easier, because the cluster pane used a JTree widget already, so anchors for the selections were immediately available after substituting Magpie's JTree wrapper.

RELATED WORK

One of the seven tasks of information visualization is to relate [7]. There are two ways to show associations: synchronizing visual attributes (e.g., color, shape, blinking) and drawing links. The former has been leveraged abundantly. The latter has also been used in numerous work on information visualization. But in most cases where links are used, links are part of the information being visualized, e.g., they are the relationships in a graph. On occasions, links are used to augment the presented information. For example, the Influence Explorer [10] shows histograms of several parameters collected from several experimental subjects and uses links to correspond data points in different histograms collected from a common subject. Augmentation has always taken place inside the same canvas (e.g., a graph view) as the information being visualized. There is one exception, LinkWinds [2], in which links are drawn between different visualizations. However, LinkWinds limits its links to point only between the linked

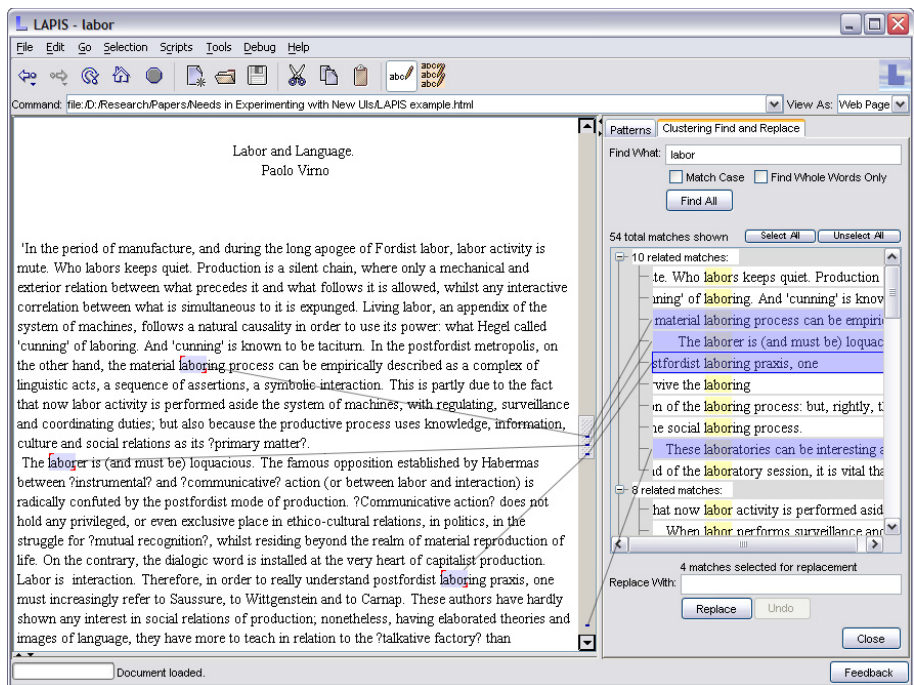


Figure 15. LAPIS with links

windows containing the visualizations, not between individual information objects like in our work.

DISCUSSION AND FUTURE WORK

When used within individual visualizations, links are parts of the information being visualized—they show relationships between data objects within the information. In contrast, links between separate visualizations show relationships between the visualizations. For example, in our SNAP redesign example, the list item “Maryland”, the map label “MD”, and the plot data point circle are essentially the same logical object but are presented differently in different visualizations. Hence, the links do not show relations between different data objects but rather reveal the fact that the three visualizations are cooperating to show three different views of the same object.

More generally, links can be used to expose to the user the internal wirings of the UI itself. In this manner, a UI can indicate how information flows through it, e.g., how checking a particular checkbox would change the content of a textbox not so nearby; what object a menu command would act upon.

As presented so far, links are rendered but not directly manipulable by the user. One can imagine allowing the user to reconnect links to request modifications to the information being visualized or to the UI’s internal wirings. The semantics and mechanism for reconnecting links need to be explored.

Like any UI mechanism, links have their niche and their limitations. In particular, it is obvious that links do not scale well: Figure 14 would be incomprehensible if all states were selected at once. However, links are still valuable even with their lack of scalability. When and how links should be used are topics for future work that demand in-depth usability evaluations.

With regards to implementation, we will attempt to incorporate links in a generic constraint-based UI management system because it seems logical to use constraints for tracking link ends. In addition, we will also explore the idea of z-order layout management.