prefuse: A Toolkit for Interactive Information Visualization

Jeffrey Heer	**Stuart K. Card**	**James A. Landay**
Group for User Interface Research	User Interface Research Group	DUB Group
Computer Science Division	Palo Alto Research Center	Computer Science & Engineering
University of California, Berkeley	3333 Coyote Hill Road	University of Washington
Berkeley, CA 94720-1776, USA	Palo Alto, CA 94301, USA	Seattle, WA 98195-2350, USA
jheer@cs.berkeley.edu	card@parc.com	landay@cs.washington.edu

ABSTRACT

Although information visualization (*infovis*) technologies have proven indispensable tools for making sense of complex data, wide-spread deployment has yet to take hold, as successful infovis applications are often difficult to author and require domain-specific customization. To address these issues, we have created prefuse, a software framework for creating dynamic visualizations of both structured and unstructured data. prefuse provides theoretically-motivated abstractions for the design of a wide range of visualization applications, enabling programmers to string together desired components quickly to create and customize working visualizations. To evaluate prefuse we have built both existing and novel visualizations testing the toolkit's flexibility and performance, and have run usability studies and usage surveys finding that programmers find the toolkit usable and effective.

Keywords: information visualization, user interfaces, toolkits, graphs, trees, interaction, navigation, 2D graphics

ACM Classification Keywords

D.2.2 **[Software Engineering]**: Design Tools and Techniques. H.5.2. **[Information Interfaces]**: User Interfaces. I.3.6 **[Methodology and Techniques]**: Interaction Techniques.

INTRODUCTION

Since the introduction of data graphics in the late 1700's [46], visual representations of abstract information have been used to demystify data and reveal otherwise hidden patterns. The recent advent of graphical interfaces has enabled direct interaction with visualized information, giving rise to over a decade of information visualization research. Information visualization (or *infovis*) seeks to augment human cognition by leveraging human visual capabilities to make sense of abstract information [12], providing means by which humans with constant perceptual abilities can grapple with increasing hordes of data.

Still, as inexpensive processing and graphics capabilities continue to improve, there remains a dearth of information visualization applications on current systems. While some of the reasons are economic [20], there are technical roadblocks as well. One is that information visualization applications are difficult to build, requiring mathematical and programming skills to implement complex layout algorithms and dynamic graphics. Another reason is that infovis applications do not lend themselves to "one size fits all" solutions; while successful visualizations often reuse established techniques, they are also uniquely tailored to their application domain (*e.g.*, [31]), requiring customization. This suggests a toolkit approach, supporting a diversity of customized applications by providing high-level support for common, reusable visualization solutions. While infovis toolkits attempting to fill this gap have begun to emerge, current offerings [9,17] provide a library of existing visualizations rather than a set of reusable components for building customized or novel visualization designs.

To address these concerns and better support the design and implementation of novel visualizations, we have built prefuse[1], an extensible user interface toolkit for crafting interactive visualizations. Instead of providing only ready-made infovis "widgets" that can be applied much like buttons or checkboxes in traditional GUI tools, prefuse provides a set of finer-grained building blocks for constructing tailored visualizations. This approach simplifies the composition of established methods, such as layout or distortion algorithms, while providing an integrated structure in which to develop novel techniques and domain-specific designs. The formalism of a graph—a set of entities and relations between them — is used as the toolkit's fundamental data structure, enabling a broad class of visualizations comprising node-link diagrams, containment diagrams, and visualizations of unstructured (edge-free) data such as scatter plots and timelines (*e.g.,* Figure 1). prefuse includes a library of layout algorithms, navigation and interaction techniques, integrated search, and more. prefuse is written in the Java programming language using the Java2D graphics library.

---

[1] In line with the musical naming conventions of Java interface toolkits, the prefuse (pronounced "pref-use") name derives from Prefuse73, an electronic musician whose work fueled toolkit development. prefuse is intentionally spelled in the lower-case.

To provide a principled toolkit flexible enough to support yet-to-be-designed visualizations while providing ample coverage of the visualization design space, we based the design of prefuse on an existing theoretical framework for infovis [11,12,15]. This model decomposes design into a process of representing abstract data, mapping data into an intermediate, visualizable form, and then using these visual analogues to provide interactive displays (Figure 2). Prior work has validated the model's expressiveness, providing a comprehensive taxonomy of visualization techniques [15].

In particular, prefuse introduces abstractions for *filtering* source data into visualizable content, providing both scalability and representational flexibility, and using composable *actions* to perform batch processing of this content, for example data transformation, layout, or color assignment. Programmers craft visualizations by stringing together *actions* into executable chains that can then be run to manipulate visual data and perform animation. Interactive views are then created from this visual data through a highly-configurable rendering system, to which pre-built controls can be added to specify interactive behaviors. This separation of concerns provides a degree of flexibility unmatched by existing infovis toolkits [9,17], supporting multiple views, semantic zooming, data and visual transformations, and application extension and customization. prefuse further demonstrates that these generalized abstractions can be provided without unduly sacrificing performance.

In the next sections we survey related work, further describe the design of prefuse, present example applications, and discuss evaluations of the toolkit, including a usability study of prefuse's application programming interface (API).

MOTIVATION AND RELATED WORK

The goal of prefuse is to simplify the creation of visualizations akin to how GUI toolkits have facilitated the design of traditional WIMP user interfaces. As such, prefuse draws from pioneering work on input abstractions like the model-view-controller [29] and interactor [36] paradigms, and the rich history and lessons learned from toolkit development [37]. This includes early systems for graph layout and editing [23,26] and for including animation in user interface toolkits [24]. While cutting-edge 2D user interface toolkits such as Piccolo [7] and its predecessor Jazz [8] provide facilities useful for information visualization such as zooming and animation support, they are not focused on supporting common visualization techniques directly. Our goal is to construct a framework of higher-level abstractions for presentation, navigation, and batch processing of interactive objects that simplifies visualization creation while affording the freedom to explore new designs.

The past 15 years have witnessed a rich body of information visualization work, featuring the creation of novel visualization designs for both structured and unstructured data. Examples include TreeMaps [10,44], Cone Trees [42], Perspective Walls [34], StarField displays [1], Hyperbolic trees [30], DOITrees [13,22], SpaceTrees [39], and more. Advances also came in the form of selection, transformation and navigation techniques, including focus+context schemes [18], space distortion [32], point-of-interest navigation [33], and panning and zooming [25,38]. Perhaps the first integrated framework for infovis was the Information Visualizer (IV) [14], featuring many of the aforementioned techniques as well as a centralized "governor" to oversee animation and ensure smooth interactive frame rates.

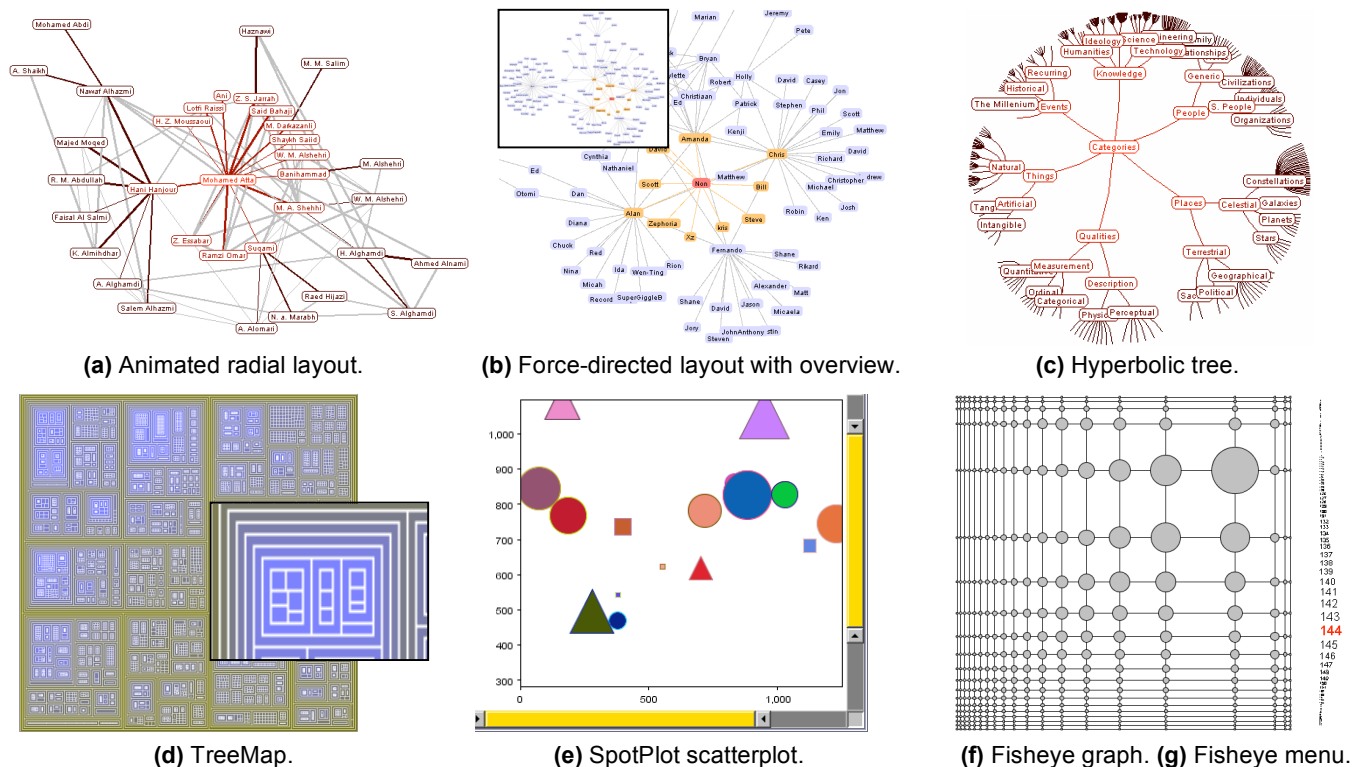
(a) Animated radial layout. **(b)** Force-directed layout with overview. **(c)** Hyperbolic tree.

(d) TreeMap. **(e)** SpotPlot scatterplot. **(f)** Fisheye graph. **(g)** Fisheye menu.

Figure 1. Sample prefuse visualizations.

Concurrently, the graph drawing community has devised algorithms for the aesthetic layout of graph structures. These are given thorough coverage by di Battista *et al.* in [4]. Perhaps the best known software for graph drawing is the excellent graphviz package from AT&T [19]. There are several other research and commercial graph drawing systems, including Marshall *et al.*'s Graph Visualization Framework (GVF) [35], the University of Ljubljana's Pajek [3], and products from Tom Sawyer and yWorks. These applications produce static visualizations and do not provide programming platforms for highly-interactive visualizations. In recent years the graph drawing community has begun moving towards more interactive solutions, signaling a convergence with the information visualization community.

While most information visualization research to date has consisted of exploring the space of successful designs and techniques, the field is now moving into a second phase in which this accumulated knowledge is applied in a principled manner. Polaris [45] applies infovis techniques to provide a powerful system for visualizing relational databases. ILOG Discovery [5] enables the declarative construction of data-linear visualizations such as plots, bar graphs, and histograms, but does not handle graph layout or animation.

The projects most similar in spirit to prefuse are infovis-specific toolkits such as Fekete's InfoVis toolkit [17] and Indiana's XML toolkit [9]. Both provide unified data models utilized by visualization "widgets" that encapsulate visualizations into monolithic units. With these toolkits, programmers can select from multiple existing visualizations such as TreeMaps or scatterplots and apply them in a straightforward manner.

Though these toolkits come a long way in making infovis techniques accessible, a finer-grained structure supporting deep customization and flexible composition of visualization methods—and thereby supporting novel approaches—is lacking. Within these existing toolkits modularity occurs primarily at the level of entire interactive visualizations rather than composable techniques, and generalized rendering and animation handling are lacking. Creating a new visualization requires either starting from scratch or subclassing a pre-existing visualization; one can not simply select and combine diverse techniques, nor craft visualization components that leverage techniques dynamically, such as orchestrating changes in item appearance (*e.g.*, semantic zooming) or providing various views and animated transitions within a single component (*e.g.*, switching between scatterplot and graph views of data). Introducing new functionality into existing visualizations without recoding can also prove difficult, as there is little decomposition of visualizations into reconfigurable parts. By abstracting visualization techniques, rendering, and interaction into composable, reusable units, we believe the state of the art can be advanced.

To meet this goal, we based the design of prefuse on existing theoretical models of information visualization. The information visualization reference model (or data state model) [11,12,15] serves as a conceptual framework for structuring infovis applications. The model decomposes design into a process of representing abstract data, mapping data into an intermediate, visualizable form, processing these visual analogues, and then mapping them into interactive displays (Figure 2). This model provides a sound base for characterizing a vast majority of infovis work (including the previous examples), providing a comprehensive taxonomy of visualization techniques [15]. Furthermore, Chi has shown that the model is functionally equivalent to the time-tested data flow model [16] used by 3D toolkits such as VTK [28]. We believe this makes the model a fit candidate as the basis for future, novel realizations. As discussed in successive sections, prefuse contributes a general implementation of this model to support a wide range of visualization designs.

DESIGN OF THE PREFUSE TOOLKIT

We now describe the toolkit design (illustrated in Figure 2), presenting the architecture, basic abstractions, and provided libraries for processing and visualizing information.
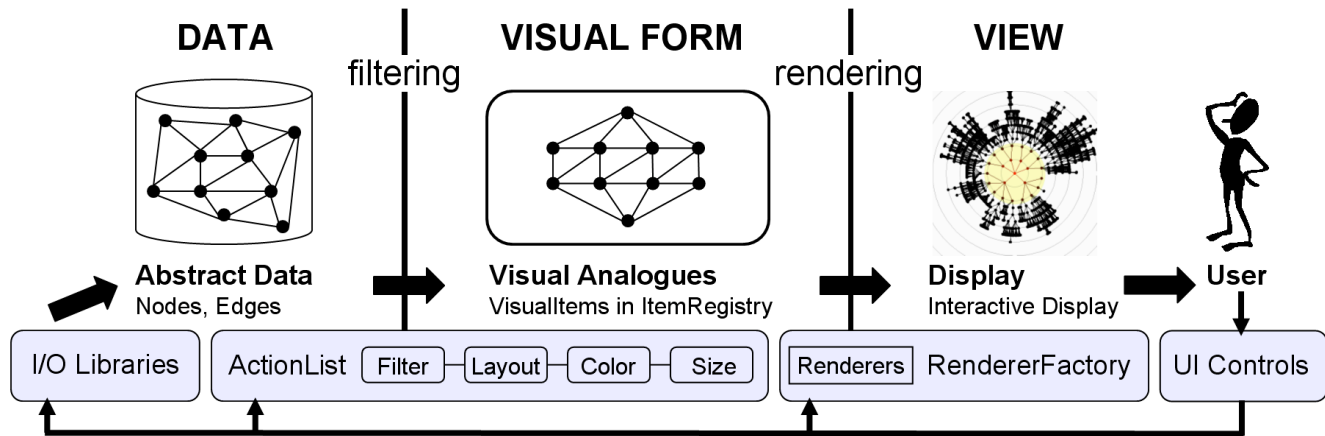
Abstract Data

The prefuse visualization process starts with abstract data to visualize, represented in some canonical form. prefuse provides interfaces and default implementations of data structures for unstructured, graph, and tree data. The basic data element type, an Entity, supports any number of named attributes (name-value pairs) and provides the base class from which structural types such as Node, TreeNode, and Edge descend. prefuse provides extensible interfaces for input and output of this data, and includes (currently read-only) support for incremental loading and caching from a database or other external store, supporting bounded visualizations of data collections too large to fit in memory.

Filtering

Filtering is the process of mapping abstract data to a representation suitable for visualization. First a set of abstract data elements are selected for visualization, such as a focal region of a graph [18] or a bounded range of values to show in a scatter plot. Next, corresponding visual analogues (called VisualItems) are generated, which, in addition to the attributes of the source data, record visual properties such as location, color, and size. Individual filters are provided in prefuse as Action modules, discussed later in this section.

In the data state model of [15], filtering constitutes the *Visualization Transformation*: reducing abstract data to visualizable content. Filtering can also be understood as implementing a tiered version of the model-view-controller pattern [29]. Abstract data provides a base model for any number of visualizations, while filtered data constitutes a visualization-specific model with its own set of view-controllers. This enables multiple visualizations of a shared data set by using separate filters, and different views of a specific visualization by reusing the same filtered items.

DATA | VISUAL FORM | VIEW

filtering   rendering

Abstract Data
Nodes, Edges

Visual Analogues
VisualItems in ItemRegistry

Display
Interactive Display

User

| I/O Libraries | ActionList | Filter | Layout | Color | Size | Renderers | RendererFactory | UI Controls |

Figure 2. The prefuse visualization framework. Lists of composable actions filter abstract data into visualizable content and assign visual properties (position, color, size, font, *etc*). Renderer modules, provided on a per-item basis by a RendererFactory, draw the VisualItems to construct interactive Displays. User interaction can then trigger changes at any point in the framework.

Managing Visual Items: The ItemRegistry

prefuse provides three types of VisualItem by default: NodeItems to visualize individual entities, EdgeItems to visualize relations between entities, and AggregateItems to visualize aggregated groups of entities. These items are arranged in a graph structure separate from the source data, maintaining a local version of the data topology and thereby enabling flexible representations of visualized content. If desired, additional VisualItem types can also be introduced.

VisualItems are created and stored in a centralized data structure called the ItemRegistry, which houses all the state for a specific visualization. Filter Actions request visual analogues from the registry, which returns the VisualItems, creating them as needed, and records the mapping between the abstract data and visualized content. The ItemRegistry also contains a FocusManager, overseeing FocusSets of items (*e.g.*, selected items and search results).

To support scalability, the ItemRegistry manages VisualItems using a caching approach, tracking item usage and performing garbage collection when previously visible items are no longer being filtered. This supports the constrained browsing of large data structures — including focus+context schemes such as generalized fisheye views [18] — by keeping only a working set of visualized items in the registry. To ensure performance, the ItemRegistry also recycles item instances when they are removed from the registry, avoiding object initialization costs that can cripple performance.

Actions

The basic components of application design in prefuse are Actions: composable processing modules that update the VisualItems in an ItemRegistry. Actions are the mechanism for selecting visualized data and setting visual properties, performing tasks such as filtering, layout, color assignment, and interpolation. To facilitate extensibility, Actions follow a simple API: a single run method that takes an ItemRegistry and an optional fraction indicating animation progress as input. In addition, base classes for specific Action types such as filters and layout algorithms are provided. While Actions

can perform arbitrary processing tasks, most fall into one of three types: filter, assignment, and animator actions.

Filter actions perform the filtering process discussed earlier, controlling what entities and relations are represented by VisualItems in the ItemRegistry. prefuse comes with filters for visualizing structures in their entirety, and for visualizing data subsets determined using degree-of-interest estimates [18,22]. By default, filters also initiate garbage collection of stale items in the registry, hiding these details from toolkit users. Advanced users can optionally disable default garbage collection and apply dedicated GarbageCollector actions.

Assignment actions set visual attributes, such as location, color, font, and size, for VisualItems. prefuse includes extensible color, font, and size assignment functions and a host of layout techniques for positioning items.

Finally, *animator* actions interpolate visual attributes between starting and ending values to achieve animation, using the animation fraction provided by the Action interface. prefuse includes animators for locations, colors, fonts, and sizes.

ActionLists and Activities

To perform data processing, Actions are composed into runnable ActionLists that sequentially execute these Actions. These lists form processing pipelines that are invoked in response to user or system events. ActionLists are Actions themselves, allowing lists to be used as sub-routines of other lists. ActionLists can be configured to run once, or to run periodically for a specified duration.

Consider the following example, in which an ActionList containing a force-directed layout and color function is applied to create an animated visualization that updates every 20ms. The ActionList parameters are the ItemRegistry to update, the duration over which to run (-1 being an infinite duration), and the rate at which to re-run the list.

```
ActionList forces = new ActionList(registry,-1,20);
forces.add(new ForceDirectedLayout());
forces.add(new ColorFunction());
forces.add(new RepaintAction());
forces.runNow(); // schedule the list to start now
```

The execution of ActionLists is managed by a general activity scheduler, implemented using the approach of [24]. The scheduler accepts Activity objects (a superclass of ActionList), parameterized by start time, duration, and step rate, and runs them accordingly. The scheduler runs in a dedicated thread and oversees all active prefuse visualizations, ensuring atomicity and helping avoid concurrency issues. A listener interface enables other objects to monitor activity progress, and pacing functions [24] can be applied to parameterize animation rates (*e.g.*, to provide slow-in slow-out animation).

Rendering and Display

VisualItems are drawn to the screen by Renderers, components that use the visual attributes of an item (*e.g.*, location, color) to determine its actual on-screen appearance. Renderers have a simple API consisting of three methods: one to draw an item, one to return a bounding box for an item, and one to indicate if a given point is contained within an item. prefuse includes Renderers for drawing basic shapes, straight and curved edges, text, and images (including image loading, scaling, and caching support). Custom rendering can be achieved by extending existing Renderers, or by implementing the Renderer interface.

Mappings between items and appearances are managed by a RendererFactory: given a VisualItem, the RendererFactory returns an appropriate Renderer. This layer of indirection affords a high level of flexibility, allowing many simple Renderers to be written and then doled out as needed. It also allows visual appearances to be easily changed, either by issuing different Renderers in response to data attributes, or by changing the RendererFactory for a given ItemRegistry. This also provides a clean mechanism for semantic zooming [38] – the RendererFactory can select Renderers appropriate for the current scale value of a given Display.

Presentation of visualized data is performed by a Display component, which acts as a camera onto the contents of an ItemRegistry. The Display subclasses Swing's top-level JComponent, and can be used in any Java Swing application. The Display takes an ordered enumeration of visible items from the registry, applies view transformations, computes the clipping region, and draws all visible items using appropriate Renderers. The Java2D library is used to support affine transformations of the view, including panning and zooming. In addition, an ItemRegistry can be tied to multiple Displays, enabling multiple views (*e.g.*, overview+detail [12]).

Displays support interaction with visualized items through a ControlListener interface, providing callbacks in response to mouse and keyboard events on items. Displays also provide direct manipulation text-editing of item content and allow arbitrary Swing components to be used as interactive tooltips.

The prefuse Library

The core prefuse architecture described above is leveraged by a library of significant components. These components simplify application design by providing advanced functions frequently used in visualizations.

Layout and Distortion. prefuse is bundled with a library of Action modules, including a host of layout and distortion techniques. Available layouts include random, circular, grid-based, force-directed, top-down [40], radial [48], indented outline, and tree map [10,44] algorithms. These layouts are parameterized and reusable, hence one can write new layouts by composing existing modules. In addition, prefuse supports space distortion of item location and size attributes, including graphical fisheye views [43] and bifocal distortion [32].

Force Simulation. prefuse includes an extensible and configurable library for force-based physics simulations. This consists of a set of force functions, including n-body forces (*e.g.*, gravity), spring forces, and drag forces. To support real-time interaction, n-body force calculations use the Barnes-Hut algorithm [2] to compute the otherwise quadratic calculation in log-linear time. The force simulation supports various numerical integration schemes, with trade-offs in efficiency and accuracy, to update velocity and position values. The provided modules abstract the mathematical details of these techniques (*e.g.*, 4th Order Runge-Kutta) from toolkit users. Users can also write custom force functions and add them to the simulator.

Interactive Controls. Inspired by the Interactor paradigm [36], prefuse includes parameterizable ControlListener instances for common interactions. Provided controls include drag controls for repositioning items (or groups of items), focus controls for updating focus and highlight settings in response to mouse actions, and navigation controls for panning and zooming, including both manual controls and speed-dependent automatic zooming [25].

Color Maps. To aid visualization, prefuse includes color maps for assigning colors to data elements. These maps can be configured directly, built using provided color schemes (*e.g.*, grayscale and color gradients, hue sampling), or automatically generated by analyzing attribute values.

Integrated Search. To simplify the addition of search to prefuse visualizations, the toolkit includes a FocusSet implementation to support efficient keyword search of large data sets. This component builds a trie (prefix tree) of requested data attributes, enabling searches that run in time proportional to the size of the query string. Search results matching a given query are then available for visualization as a FocusSet in the ItemRegistry's FocusManager.

Event Logging. prefuse includes an event logger for monitoring and recording events. This includes both user interface events (mouse movement, focus selection) and internal system events (addition and deletion of items from the registry). Although useful for debugging and performance monitoring, the primary motivation for this feature is to assist user studies, providing a unified framework for evaluating visualizations. Recorded logs can be used to review or replay a session. We have even synchronized the event logger with the output of an eye-tracker, enabling us to playback sessions annotated with subjects' fixation points.

WRITING APPLICATIONS WITH PREFUSE

In this section we demonstrate how prefuse can be used to craft and extend an interactive visualization by chaining together components, creating extensible applications while minimizing the need for tedious coding or mathematics.

Code Sample 1 presents 24 lines of code comprising a complete prefuse application for exploring graphs using animated radial layout (as in Figure 1(a) and [48]). The application first loads a graph data set from an XML file and creates a new ItemRegistry to house a visualization of that data. Next, individual Renderers for node and edge items are created and a default RendererFactory is created to assign these renderers to the appropriate items.

Two ActionLists are used to specify the visualization. The first filters the graph data into a tree structure, applies a radial tree layout, and then assigns colors to the nodes. The argument to the TreeFilter specifies that the current focus node should be used as the root of the filtered tree. The default ColorFunction used provides custom colors for focused or highlighted items. The second ActionList specifies an animated transition for when the focus of the visualization changes. It is parameterized to run for 1.5 seconds, interpolating node positions in polar coordinates and interpolating color values. This list is set to run whenever the previous layout ActionList completes.

A Display is then created to present the visualization. Two interactive controls are added: a DragControl enabling users to reposition nodes, and a FocusControl enabling users to select a new focus by clicking on a node, initiating a recalculation of the layout and an animated transition. Finally, the Display is added to an enclosing frame, and the layout ActionList is run.

The prefuse architecture supports the addition of customizations and extensions by introducing new Actions, Renderers, or Controls. For example, if the underlying data set consists of a very large graph, the TreeFilter can be replaced with a WindowedTreeFilter to limit the visualization to a specified degree of separation (*e.g.*, 3 hops out from the focus). Code Samples 2 through 4 further exemplify the space of possible customizations.

Code Sample 2 illustrates how to use a force simulator to cause nodes to repel each other, enhancing the layout by adding jitter to improve readability. The force simulation animates for 1 second after the layout transition completes.

Code Sample 3 shows how to add an overview display to the visualization (*e.g.*, see Figure 1b) and enable panning and zooming. Panning is performed by holding down the left mouse button on the background and dragging, zooming is performed similarly using the right mouse button.

Finally, Code Sample 4 demonstrates the addition of fisheye distortion to the visualization (*e.g.*, Figure 1f). An ActionList containing a Distortion action is created and invoked by an AnchorUpdateControl control that monitors mouse movement to move the focus (or "anchor") of the distortion.

```
// create graph and registry
Graph g = new XMLGraphReader().loadGraph(datafile);
ItemRegistry registry = new ItemRegistry(g);

// intialize renderers
Renderer nodeR = new TextItemRenderer();
Renderer edgeR = new DefaultEdgeRenderer();
registry.setRendererFactory(
  new DefaultRendererFactory(nodeR, edgeR));

// initialize action lists
ActionList layout = new ActionList(registry);
layout.add(new TreeFilter(true));
layout.add(new RadialTreeLayout());
layout.add(new ColorFunction());

ActionList animate = new ActionList(registry,1500);
animate.setPacingFunction(new SlowInSlowOutPacer());
animate.add(new PolarLocationAnimator());
animate.add(new ColorAnimator());
animate.add(new RepaintAction());
animate.alwaysRunAfter(layout);

// initialize display
Display disp = new Display(registry);
disp.setSize(500,500);
disp.addControlListener(new DragControl());
disp.addControlListener(new FocusControl(layout));

// initialize enclosing window frame
JFrame frame = new JFrame("prefuse example");
frame.getContentPane().add(disp);
frame.pack(); frame.setVisible(true);

layout.runNow();
```

Code Sample 1: Radial Graph Explorer

```
ForceSimulator fsim = new ForceSimulator();
fsim.addForce(new NBodyForce(-0.1f, 15f, 0.9f));
fsim.addForce(new DragForce());

ActionList forces = new ActionList(registry, 1000);
forces.add(new ForceDirectedLayout(fsim, true));
forces.add(new RepaintAction());
forces.alwaysRunAfter(animate);
```

Code Sample 2: Adding Force-Based "Jitter"

```
Display overview = new Display(registry);
overview.setBorder(
  BorderFactory.createLineBorder(Color.BLACK, 1));
overview.setSize(50,50);
overview.zoom(new Point2D.Float(0,0),0.1);
display.add(overview);
display.addControlListener(new PanControl());
display.addControlListener(new ZoomControl());
```

Code Sample 3: Adding an Overview, Panning, and Zooming

```
Distortion feye = new FisheyeDistortion();
ActionList distort = new ActionList(registry);
distort.add(feye);
distort.add(new RepaintAction());

AnchorUpdateControl auc =
  new AnchorUpdateControl(feye,distort);
display.addMouseListener(auc);
display.addMouseMotionListener(auc);
```
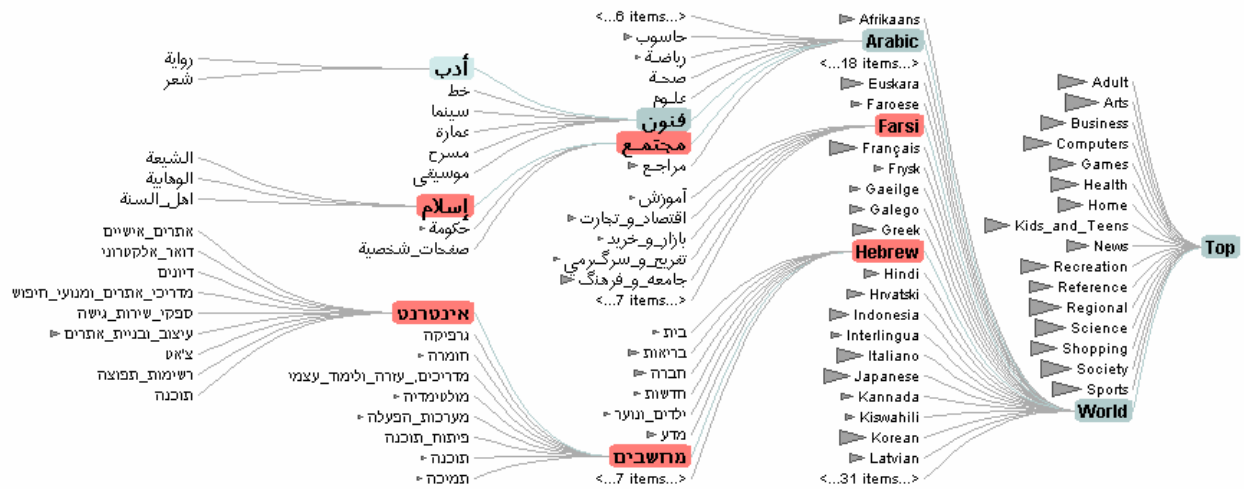
Code Sample 4: Adding Fisheye Distortion

Figure 3. Degree-of-Interest Tree visualizing a 600,000 node web directory.

EVALUATION – APPLICATION COVERAGE

Throughout the development of the toolkit, we both reimplemented existing visualizations and crafted novel designs to the test the expressiveness, effectiveness, and scalabilty of the toolkit. As shown in Figure 1, these existing visualizations include animated radial graphs [48], animated force-directed layout (similar to plumbdesign's Visual Thesaurus [47]), the hyperbolic tree browser [30], "squarified" tree maps [10], range-slider controlled starfield displays [1], fisheye graphs and fisheye menus [6], and the Data Mountain [41] (not pictured). prefuse greatly simplified the implementation of these visualizations, in some cases turning what might have been a matter of days or weeks into a matter of minutes. For example, using prefuse it took only 2 hours to implement the Data Mountain and a mere 20 minutes to create Fisheye Menus. Video demonstrations and implementation details of these applications are available at http://prefuse.sourceforge.net. We now describe in greater detail our experiences using prefuse to build two novel visualizations.

Degree-of-Interest Trees. We have used prefuse to create a novel hierarchy browser [22], an evolutionary step from Card and Nation's original Degree-of-Interest Tree (DOITree) browser [13]. DOITrees are tree visualizations featuring multiple focus+context techniques, including the use of degree-of-interest (DOI) functions [18] to determine which regions of the tree are visible, and the use of aggregates to represent unexpanded subtrees and to group lower-interest siblings in the face of limited space resources. Figure 3 shows a prefuse-built DOITree visualizing a web directory with over 600,000 nodes. Clicking a node in the visualization causes it to become a focus, initiating a recalculation of DOI values and layout followed by an animated transition. The visualization also supports multiple foci, selected through both manual selection and keyword search.

We implemented DOITrees using four ActionLists, all of which are sequentially scheduled in response to changes of focus node. The first list performs filtering, computes layout, and assigns initial colors. The second ActionList interpolates positions and colors to provide animated transitions. The third and fourth lists assign and then animate highlighting changes designed to make newly visible nodes easier to track. Additionally, an ActionSwitch (similar to a multiplexer) is used in the first list to select from one of three filters: a standard fisheye calculation [18], a custom filter showing only focus nodes (*e.g.*, search results) and their ancestors, and another filter showing only focus nodes and their least common ancestors. Each filter provides progressively more semantically "zoomed-out" views of the data, facilitating exploration of different foci quite far apart in the tree [22].

As we developed the DOITree browser, the toolkit enabled us to add animated behaviors (initial highlighting and fade-out for tracking newly visible items), design and incorporate a new layout algorithm [22], provide integrated handling of search results, and customize item appearances to specific application domains by crafting custom renderers. This application also demonstrates the toolkit's scalability, maintaining real-time interaction with data sets containing nearly a million items.

Vizster. Vizster [21] is a prefuse-built visualization of online social network services such as Friendster and Orkut (see Figure 4). It provides an ego-centric view of a person's social network, presented using a force-directed layout. We are currently using Vizster to visualize a 1.5 million person crawl of the popular Friendster service. Each node displays a person's name and image. Clicking a node causes a corresponding membership profile, containing information such as interests and relationship status, to appear in the panel on the right. Double-clicking a node makes the corresponding person the new center of the ego-centric network. The persons' friends are loaded from a backing database and displayed while the display automatically pans to center on the new focus. Manual panning and zooming are also supported; semantic zooming is used to switch to

higher resolution images of people when zoomed in. Typing in the search box immediately causes both matching nodes in the visualization and matching text in the profile to highlight.

In addition to the browsing mode described above, Vizster supports a comparison mode (see Figure 5), accessed by clicking the radio button next to the desired attribute in the profile panel. In response, node appearances simplify to using color to display the desired attribute of the data, such as age, gender, or relationship status. Alternative color maps can be used by selecting them from the application menu.

Underlying Vizster is a rather straightforward application of prefuse's built-in components, such as fisheye graph filtering, force-directed layout, image loading and rendering, panning, zooming, integrated search, and color maps. The application uses one primary ActionList, infinitely re-running the force simulation while also setting the node color values. An ActionSwitch is used to select the appropriate ColorFunction based on the state of the application. Furthermore, a custom RenderingFactory is used, overseeing semantic zooming and doling out image renders in browsing mode and text-only renderers in comparison mode. While the application consists of a total of 1541 lines of code, only 469 lines, or less than one-third, deal with specifying the visualization. The majority of the code deals with constructing traditional user interface components such as a login dialog and the profile panel. Using prefuse, we were able to construct the entire application in under a week.

Summary
The applications above showcase prefuse's support for component reuse and extensibility, using provided modules (*e.g.*, filters, layouts, renderers, interactive controls) across visualizations, while making it easy for both ourselves and others to introduce customized components. We also found that prefuse's highly-customizable rendering and animation support greatly accelerated implementation times and the exploration of various design ideas. Finally, the applications demonstrate that toolkit support did not unduly sacrifice performance, as applications maintained real-time interaction and animation rates with thousands of on-screen items and over a million data elements.

EVALUATION - QUALITATIVE USABILITY STUDY
While confident in the toolkit's expressiveness, we wanted to better understand the learnability and usability of prefuse's application programming interface (API) for other programmers. In particular, abstractions such as filtering and action lists might seem foreign to some programmers, constituting the *threshold* for toolkit use [37]. To investigate these concerns, we adopted the evaluation method of [28] and conducted a usability study of the prefuse API, observing 8 programmers using the toolkit to build applications and then interviewing them about their experiences.

The 8 participants were of varying background and expertise: 4 computer science students (2 undergrads, 2 grads), 3 professional programmers and/or user interface designers,
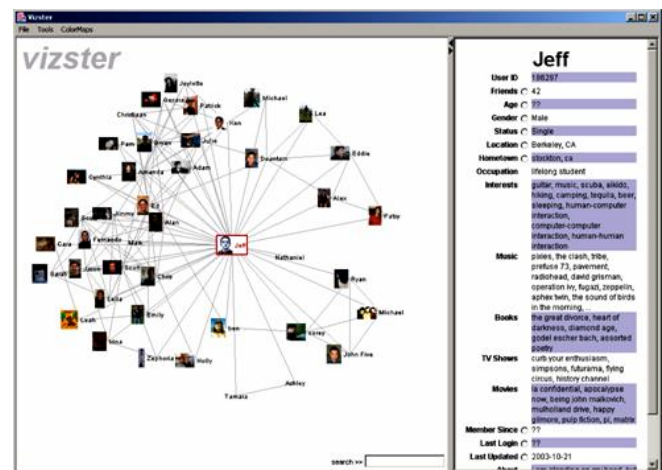

Figure 4. Vizster in browsing mode, showing an ego-centric network of friendship relations. The panel on the right displays profile data for a selected person.
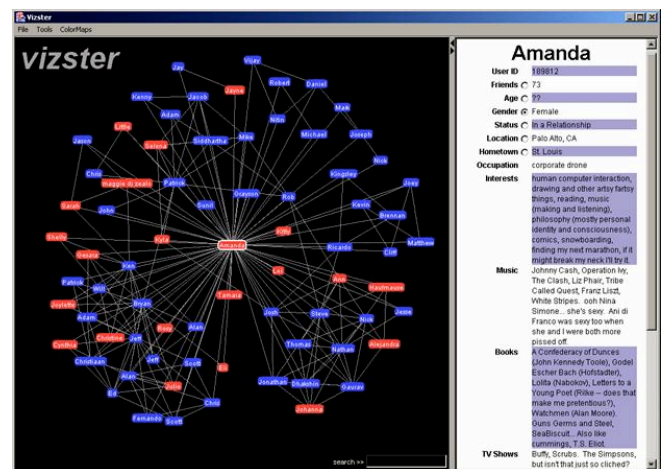

Figure 5. Vizster in comparison mode, using color to   display the genders of visualized friends.

and 1 information visualization expert. All were screened for familiarity with Java, the Swing UI toolkit, and the Eclipse integrated development environment.

Participants were first given a brief tutorial lasting about 20 minutes, including a code walkthrough of some sample applications. Subjects were then given a social network data file and asked to perform three programming tasks. The first was to create a static (non-animated) visualization of the data set using a random layout. The second task asked subjects to refine their visualization by applying a layout technique of their choice and using color to convey information about one or more data attributes. Finally, subjects were asked to add interactivity and animation, supporting a change of focus or other means of exploring the data. Tasks were performed on a Windows PC pre-loaded with the Eclipse IDE and prefuse source code, examples, and API documentation. Subjects were encouraged to "think-aloud" and were given up to an hour to complete the tasks. The tasks were videotaped and subject's code samples were saved for later analysis. The tasks were followed by a short, open-ended interview in

which subjects were asked about their experiences and their understanding of various toolkit abstractions. Interviews typically lasted 15-20 minutes and were audio recorded.

Results

Every subject successfully built a working visualization, and 7 of the 8 subjects completed every task. Subjects did not necessarily complete tasks in the order presented (they were told this was fine) and half encountered trouble at some point in the study. The most common difficulty was structuring dataflow appropriately, making sure that filtered structures worked with downstream components such as layout algorithms. For example, four subjects wanted to apply a radial layout in their design, but ran into troubles when they used a general graph filter and the radial layout algorithm, expecting a tree, threw an exception. In response to these and similar issues, we subsequently redesigned the filtering system to better align with user's expectations. For example, TreeFilters now automatically overlay a tree structure on filtered items when the source data is a general graph.

The study also proved useful for unearthing naming issues. Most notably, VisualItems had originally been called GraphItems, an obvious (in hindsight) blunder that fueled confusion as to which data was abstract and which was visual content. ActionLists were initially called ActionPipelines, but were renamed to avoid association with the streaming nature of traditional pipeline architectures.

Participant reaction to the toolkit, even from those who had difficulty, was encouraging. Many appreciated the toolkit design, saying "I'm surprised I needed as little code as I did!" and "[It's] shockingly easy to use." Four of the subjects wanted to use prefuse in their own work, and have downloaded the toolkit. One subject, who had been searching for tools to build visualizations of software execution, stated "This is the first thing I have found that can do what I want."

In addition to the findings directly related to prefuse, a couple of usage patterns emerged that are relevant to the study of software toolkits in general. One result was the rather minimal usage of the provided API documentation. Only one participant referred to documentation early on (exclaiming "I'm a javadoc fan!"); all others worked on tasks for at least 30 minutes before opening the documentation. When asked about this behavior in the post-study interview, subjects offered a number of explanations. Many said that they preferred to work directly with the code and explore problems as they arose, resorting to documentation only when a problem offers continued difficulty. One subject intimated that he preferred to stay within the Eclipse environment, as he felt switching between different applications (the documentation is read in a web browser) would slow him down.

Furthermore, all eight subjects at least initially used a "cut and paste" method of application building, reusing existing sample code while performing tasks. Many subjects commented negatively on this as they did it, saying it was "bad" or "embarrassing" (one subject even asked for permission!). When asked about this, subjects were about evenly split in describing their reasons for this perceived "shame." One camp maintained that they had been taught (largely in school) that "blindly" copying code was bad software engineering practice, for reasons too numerous to list here. Others felt that by copying and pasting they were not learning the toolkit deeply enough, and thus somehow not participating fully in the study. Despite this unease, all subjects disclosed in the post-task interviews that this was their typical approach to learning unfamiliar APIs. All subjects expressed the belief that sample code was the best way to learn new programming environments, making it clear that a toolkit's "user interface" is not just an API, but the associated materials (code samples, documentation) as well, all of which should be the subject of design.

Summary

Through the evaluation process, the toolkit has made great strides. Both the application building process and user study have validated the goals of our toolkit while revealing needed functionality and suboptimal design decisions. The filtering abstraction, while setting the learning curve for the system, was understood by user study participants and has enabled an array of scalable visualizations. Using prefuse, study subjects built useful visualizations in under an hour, and toolkit users expressed an appreciation of the accompanying extensibility.

We have found that iterative design, a proven method for developing user interfaces, has also proven a valuable design method for software toolkits. Since the study, an alpha release of prefuse has been downloaded over 1300 times and is being used in research projects, course assignments, and commercial products. We are following this usage in a longitudinal study of toolkit use, including a recent survey of 20 programmers. This has unearthed additional requirements, from bug fixes to the need for improved documentation. Overall, reaction to prefuse has been overwhelmingly positive, enabling programmers of varying skill levels to create new visualizations of their own.

CONCLUSION

In this paper we have introduced prefuse, a user interface toolkit for crafting interactive visualizations of structured and unstructured data. prefuse supports the design of 2D visualizations of any data consisting of discrete data entities, such as graphs, trees, scatter plots, collections, and timelines. prefuse implements existing theoretical models of information visualization to provide a flexible framework for simplifying application design and enabling reuse and composition of visualization and interaction techniques. In particular, prefuse contributes scalable abstractions for *filtering* abstract data into visual content and using lists of composable *actions* to manipulate data in aggregate.

Applications built with the toolkit demonstrate the flexibility and performance of the prefuse architecture. Both a user study and real-world usage has shown that programmers can use the toolkit to quickly build and tailor their own interactive visualizations.

prefuse is part of a larger move to systematize information visualization research and bring more interactivity into data analysis and exploration. In future work, we plan to introduce more powerful operations for manipulating source data, provide additional components, and potentially develop a visual environment for application authoring. Most importantly, both we and others are now using the toolkit to build and evaluate new interactive visualizations for a variety of application domains.

prefuse is open-source software. The toolkit, source code, and both interactive and video demonstrations are available at `http://prefuse.sourceforge.net`.

ACKNOWLEDGMENTS

REFERENCES

1. Ahlberg, C. and B. Shneiderman. Visual Information Seeking: Tight Coupling of Dynamic Query Filters with Starfield Displays. *CHI'94*. pp. 313-317, April 1994.
2. Barnes, J. and P. Hut, A Hierarchical O(N Log N) Force Calculation Algorithm. *Nature*, 1986. **324**(4).
3. Batagelj, V. and A. Mrvar, Pajek: Analysis and Visualization of Large Networks, in *Graph Drawing Software*, Springer. p. 77-103, 2003.
4. Battista, G.D., P. Eades, R. Tamassia, and I.G. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs*. Upper Saddle River, NJ: Prentice Hall, 1999.
5. Baudel, Thomas. Canonical Representation of Data-Linear Visualization Algorithms and its Applications. ILOG Report. http://www2.ilog.com/preview/Discovery/technology/DiscoveryResearchPaper.pdf
6. Bederson, B.B. Fisheye Menus. *UIST'00*. pp. 217-225, 2000.
7. Bederson, B. B., J. Grosjean, & J. Meyer. Toolkit Design for Interactive Structured Graphics, *IEEE Transactions on Software Engineering*, 30 (8), pp. 535-546, 2004.
8. Bederson, B.B., J. Meyer, and L. Good. Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java. *UIST'00*. pp. 171-180 2000.
9. Borner, K. *et al*. The XML Toolkit. Project Webpage. 2003. http://iv.slis.indiana..edu/sw/toolkit/.html
10. Bruls, M., K. Huizing, and J.J. van Wijk. Squarified TreeMaps. In Proceedings of *Joint Eurographics and IEEE TCVG Symp. on Visualization (TCVG 2000)*: IEEE Press. pp. 33-42, 2000.
11. Card, S.K., Information Visualization, in *The Human-Computer Interaction Handbook*. Lawrence Erlbaum Associates, 2002.
12. Card, S.K., J.D. Mackinlay, and B. Shneiderman, *Readings in Information Visualization: Using Vision to Think*. San Francisco, California: Morgan-Kaufmann, 1999.
13. Card, S.K. and D. Nation. Degree-of-Interest Trees: A Component of an Attention-Reactive User Interface. *Advanced Visual Interfaces*. 2002.
14. Card, S.K., G.G. Robertson, and J.D. Mackinlay. The Information Visualizer, an Information Workspace. *CHI'91*. pp. 181-188 1991.
15. Chi, E.H. A Taxonomy of Visualization Techniques Using the Data State Reference Model. *InfoVis '00*. pp. 69-75 2000.
16. Chi, E.H. Expressiveness of the Data Flow and Data State Models in Visualization Systems. *Advanced Visual Interfaces*. Trento, Italy, May 2002.
17. Fekete, J.-D. The InfoVis Toolkit, *InfoVis '04*, pp. 167-174, 2004.
18. Furnas, G.W., The Fisheye View: A New Look at Structured Files, in *Readings in Information Visualization: Using Vision to Think*, S.K. Card, *et al*, Editors. Morgan Kaufmann: San Francisco, 1981.
19. Graphviz. http://www.research.att.com/sw/tools/graphviz/
20. Grokking the Infoviz, *Economist Technology Quarterly*, June 2003.
21. Heer, J. Vizster: Visualizing Online Social Networks. April 2004. http://www.cs.berkeley.edu/~jheer/infovis/final
22. Heer, J. and S.K. Card. DOITrees Revisited: Scalable, Space-Constrained Visualization of Hierarchical Data. *Advanced Visual Interfaces*. Gallipoli, Italy, May 2004.
23. Henry, T.R. and S.E. Hudson. Interactive Graph Layout. *UIST'91*. pp. 55-64, November 1991.
24. Hudson, S. and J.T. Stasko. Animation Support in a User Interface Toolkit: Flexible, Robust, and Reusable Abstractions. *UIST'93*. pp. 57-67, 1993.
25. Igarashi, T. and K. Hinckley. Speed-Dependent Automatic Zooming for Browsing Large Documents. *UIST'00*. pp. 139-148, 2000.
26. Karrer, A. and W. Scacchi. Requirements for an Extensible Object-Oriented Tree/Graph Editor. *UIST'90*. pp. 84-91, October 1990.
27. *The Visualization Toolkit User's Guide*: Kitware, Inc., 2003.
28. Klemmer, S.R., J. Li, J. Lin, and J.A. Landay. Papier-Mâché: Toolkit Support for Tangible Input. *CHI'04,* Vienna, Austria 2004.
29. Krasner, G.E. and S.T. Pope, A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. *Journal of Object-Oriented Programming*, 1988. **1**(3): p. 26-49.
30. Lamping, J. and R. Rao, The Hyperbolic Browser: A Focus + Context Technique for Visualizing Large Hierarchies. *Journal of Visual Languages and Computing*, 1996. **7**(1): p. 33-55.
31. Lee, B., C.S. Parr, D. Campbell, and B. Bederson. How Users Interact with Biodiversity Information Using Taxontree. *Advanced Visual Interfaces*. Gallipoli, Italy 2004.
32. Leung, Y.K. and M.D. Apperley, A Review and Taxonomy of Distortion-Oriented Presentation Techniques. *ACM Transactions on Computer-Human Interaction*, 1994. **1**(2): p. 126-160.
33. Mackinlay, J.D., S.K. Card, and G.G. Robertson, Rapid, Controlled Movement through a Virtual 3d Workspace. *Computer Graphics*, 1990. **42**(4): p. 1971-1976.
34. Mackinlay, J.D., G. Robertson, and S.K. Card. The Perspective Wall: Detail and Context Smoothly Integrated. *CHI91*. pp. 173-179 1991.
35. Marshall, M.S., I. Herman, and G. Melancon, An Object-Oriented Design for Graph Visualization. *Software: Practice and Experience*, 2001. **31**(8): p. 739-756.
36. Myers, B.A., A New Model for Handling Input. *ACM Transactions on Information Systems*, 1990. **8**(3): p. 289-320.
37. Myers, B.A., S.E. Hudson, and R.F. Pausch, Past, Present, and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction*, 2000. **7**(1): p. 3-28.
38. Perlin, K. and D. Fox. Pad: An Alternative Approach to the Computer Interface. *SIGGRAPH'93*. pp. 57-64, 1993.
39. Plaisant, C., J. Grosjean, and B. Bederson. Spacetree: Supporting Exploration in Large Node Link Tree, Design Evolution and Empirical Evaluation. *InfoVis '02*. Boston, MA. pp. 57-64, October 2002.
40. Reingold, E.M. and J.S. Tilford, Tidier Drawings of Trees. *IEEE Transactions of Software Engineering*, 1981. **SE-7**: p. 21-28.
41. Robertson, G.G., M. Czerwinski, K. Larson, D.C. Robbins, D. Thiel, and M.v. Dantzich. Data Mountain: Using Spatial Memory for Document Management. *UIST'98*. pp. 153-162 1998.
42. Robertson, G.G., J.D. Mackinlay, and S.K. Card. Cone Trees: Animated 3D Visualizations of Hierarchical Information. *CHI'91*. pp. 189-194, 1991.
43. Sarkar, M. and M.H. Brown. Graphical Fisheye Views of Graphs. *CHI'92*. pp. 83-91, May 1992.
44. Treemaps for Space-Constrained Visualization of Hierarchies. 1998. http://www.cs.umd.edu/hcil/treemap-history/
45. Stolte, C., D. Tang, and P. Hanrahan, Polaris: A System for Query, Analysis and Visualization of Multi-Dimensional Relational Databases. *IEEE Transactions on Visualization and Computer Graphics*, 2002. **8**(1).
46. Tufte, *The Visual Display of Quantitative Information*. Graphics Press, 1983.
47. Visual Thesaurus. http://www.visualthesaurus.com
48. Yee, K.-P., D. Fisher, R. Dhamija, and M.A. Hearst. Animated Exploration of Dynamic Graphs with Radial Layout. *InfoVis'01*. pp. 43-50 2001.