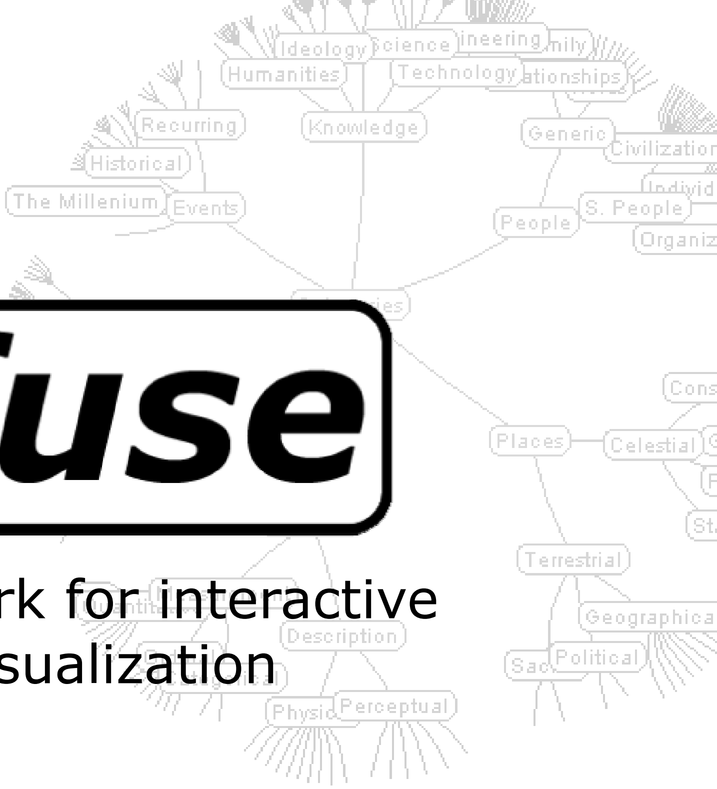
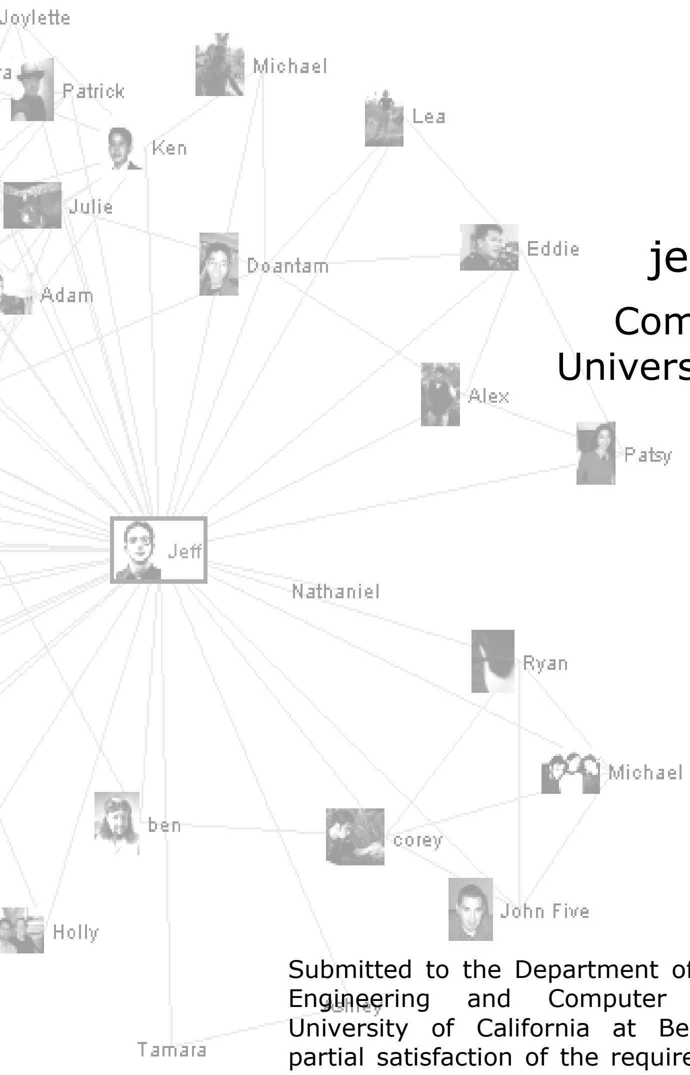


prefuse

a software framework for interactive information visualization



jeffrey michael heer

Computer Science Division
University of California, Berkeley
December, 2004

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of Master of Science, Plan II

Approval for the Report and Comprehensive Examination:

Professor James A. Landay
Research Advisor

Professor Marti Hearst
Second Reader

acknowledgements

this report represents the fruit of months of effort, built atop a foundation years in the making. none of this would have been possible without the tireless efforts of those who helped lay that foundation.

thanks to the American Society for Engineering Education for financial support of my studies.

thanks to the University of California, Berkeley Computer Science Division, for years of education, inspiration, and support.

thanks to both my advisor and reader, James Landay and Marti Hearst, for providing support and feedback while trusting me with the freedom to explore. thanks also to Anind Dey, Jennifer Mankoff, Marc Davis, and Peter Lyman for their guidance and help navigating the ever-changing landscapes of academia.

thanks to my esteemed graduate colleagues, both past and present: Alan Newberger, Scott Lederer, Scott Klemmer, Chris Beckmann, Jason Hong, Scott Carter, Tara Matthews, Nathan Good, Ana Ramirez, Guillermo Diez-Canas, and Tye Rattenbury.

thanks to my intellectual parents at Palo Alto Research Center, formerly Xerox PARC. i would not be where i am today without the mentorship of Stuart Card, Jock Mackinlay, Peter Pirolli, and the rest of the Information Sciences and Technology Laboratory. in particular, a very special thanks is due to Ed Chi, who years ago met one precocious undergrad and showed him how to be a researcher.

thanks to the friends who made it worth doing in the first place: Zohr, Fred, James, Bloke, Cones, Meezil, Rookie, Danny, Leah, Katrina, Sarah(x2), Corey, Ryan, Nathan, Kevin, Mirko, Amanda, Ken, Adam, Bryan, Zahra, Curtis, Gazelle, and members of the Berkeley Crew everywhere.

finally, thanks to my family for all their support, love, and encouragement throughout the years.

-jmh

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.

Interactive demonstrations, video demonstrations, and open-source software for the prefuse project are available at `http://prefuse.sourceforge.net`.

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 libraries of pre-built 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

[1] In line with the musical naming conventions of Java user interface toolkits, the prefuse (pronounced "pref-use") name derives from Prefuse 73, an electronic musician whose work fueled my own. prefuse is intentionally spelled in the lower-case.

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 (Figure 1 contains a sample visualization). 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.

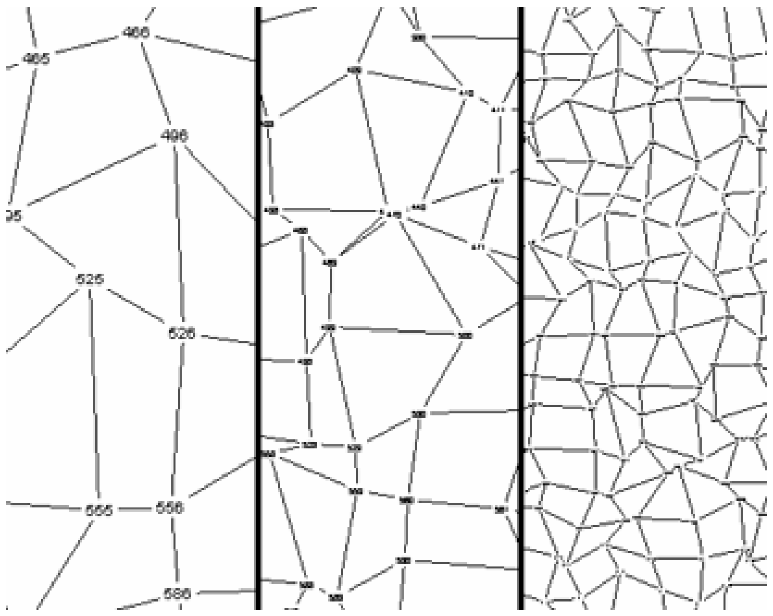


Figure 1. Consecutive snapshots of a prefuse-built graph viewer featuring automatic zooming.

Informed by a review of existing applications, our own years of experience designing novel visualizations, and earlier toolkit evaluations, the prefuse toolkit offers:

- Data structures and I/O libraries for unstructured, graph, and tree data
- Multiple visualizations of a single data source
- Multiple views of a single visualization
- Scalability to thousands of on-screen items, and to backing data sources with millions of elements
- Batch processing of data using composable actions
- A library of provided layout and distortion techniques
- Animation and time-based processing
- Graphics transforms, including panning and zooming
- A physical force simulator for layout and interaction

- Interactor components for common interactions
- Integrated color maps and search functionality
- Event logging to support visualization evaluation

To provide a principled toolkit flexible enough to support novel visualizations while providing ample coverage of the visualization design space, we based the design of prefuse on an existing theoretical framework for infovis, the "data state" or infovis reference model [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 section we survey related work, motivating the need for our toolkit. Next, we describe the design of prefuse and walk through an example prefuse-built visualization. We then present evaluations of the toolkit, including applications showcasing the toolkit's power and flexibility, and a qualitative study demonstrating the usability 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 (Windows-Icons-Menus-Pointing) 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 [14], featuring many of the aforementioned techniques as well as a centralized “governor” to oversee animation and ensure smooth interactive frame rates.

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 largely static visualizations and do not constitute programming platforms for highly-interactive visualizations. However, in recent years the graph drawing community has begun moving towards increasingly interactive solutions, signaling a possible 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. For example, Polaris [45] applies infovis techniques to provide a powerful system for visualizing relational databases. ILOG Discovery [5] allows for the declarative construction of data-linear visualizations such as plots, bar graphs, histograms, and containment diagrams, but does not handle graph layout or interactive 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 layout, rendering, and interaction in 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 (*e.g.,* from XML data files), 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. These filtered VisualItems also maintain their own version of the data topology. Though in

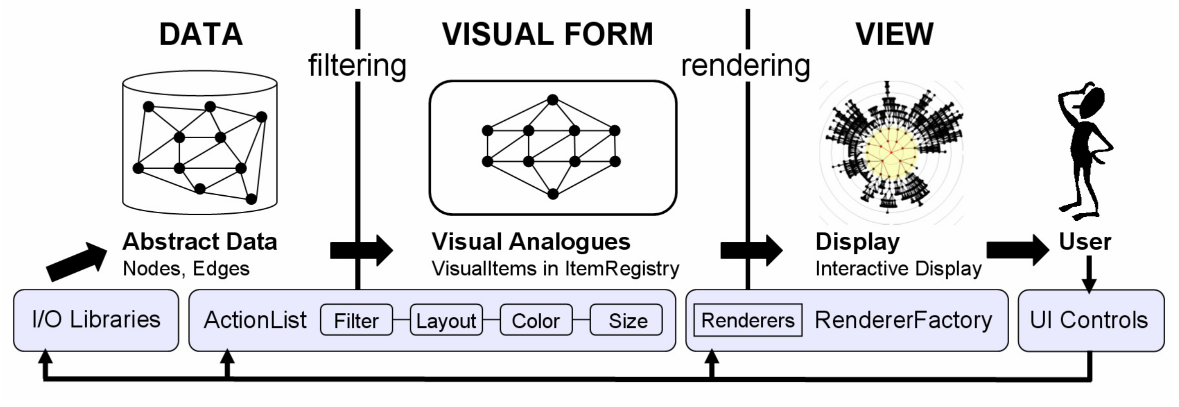


Figure 2. The prefuse visualization framework. Lists of composable actions filter abstract data into visualizable content and assign visual properties (position, color, size, font, *etc*). Renderer modules, provided on a per-item basis by a RendererFactory, draw the VisualItems to construct interactive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

many cases this may simply be a mirror of a subset of the abstract data's structure, this representational flexibility allows any number of transformations of the original topology. One example, demonstrated later in the Applications section, is to remove intermediate levels of a tree based on inferred user interest in the data items. 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.

Managing Visual Items: The Item Registry

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. As mentioned above, 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. This type system is extensible; if needed, additional VisualItem types can 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. Each VisualItem type is stored in a sorted queue, with an assignable Java Comparator instance overseeing the order of items both within and across queues. This sorting determines the rendering order of the items. Hashtables for each item type maintain the mapping between source Entity instances and VisualItems, while each individual VisualItem maintains a reference to the source Entity (or in the case of AggregateItems, to the collection of Entity instances) represented. The ItemRegistry also contains a FocusManager, overseeing FocusSets of items such as the current focus of interaction, collections of 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. Each VisualItem instance contains a counter. This counter is reset to zero if the item is requested by the filter, otherwise the counter is incremented. Once the counter reaches a threshold value (by default set to 1), the item will be removed from the registry.

When removed from the registry, a VisualItem is removed from the queue for its item class and any mappings between source data and the item are deleted. To ensure performance, the ItemRegistry recycles item instances when they are removed from the registry, clearing the state of the item and placing it in an object pool. When new VisualItems are requested, the ItemRegistry first checks this pool and reuses and reinitializes an existing instance if available. This pooling avoids memory allocation and 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.

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.

Finally, prefuse also includes an ActionSwitch, which chooses and runs a single Action from a collection. This provides a means for providing dynamic action invocation, for example by choosing from a selection of various filters in response to user actions.

ActionLists and Activities

To perform data processing, Actions are composed into runnable ActionLists that sequentially execute contained Actions. These lists form processing pipelines that are invoked in response to user or system events. ActionLists are Actions themselves, allowing nested lists to be used as sub-routines within 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, providing callbacks when activities are started, stepped, finished, or canceled. Time-based processing is controlled by uniformly moving an animation fraction, a value between 0 and 1, over the requested time span of the activity. Pacing functions [24], which map the animation fraction to a new value (which must still remain between 0 and 1), can be applied to parameterize animation rates. This allows for effects such as slow-in slow-out animation (by mapping the animation fraction through a sigmoid-shaped function) and there-and-back animation (by moving from 0 to 1 over the first half of the activity duration, then moving from 1 back to 0).

Rendering and Display

VisualItems are drawn to the screen by Renderers, components that use the visual attributes of items (*e.g.*, location, color) to determine their 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 on the fly, 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 components for application building. 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, force-directed, top-down (Reingold-Tilford) [40], radial [49], 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], which builds a quad-tree of items 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 [48]) 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.

These components, coupled with the underlying capabilities of the prefuse architecture, provide an expressive platform for crafting a range of highly-interactive visualization applications. The next sections provide a sample of this range by presenting various prefuse -built visualizations and illustrate how the architecture facilitates development while providing scalable and responsive visualization performance.

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.

```
// 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

Code Sample 1 presents 24 lines of code comprising a complete prefuse application for exploring graphs using animated radial layout (as in Figure 3 and [49]). 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.

```
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”

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.

```
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

Code Sample 3 shows how to add an overview display to the visualization (*e.g.*, see Figure 4) 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.

```
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

Finally, Code Sample 4 demonstrates the addition of fisheye distortion to the visualization (*e.g.*, Figure 6a). 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.

EVALUATION – APPLICATION COVERAGE

Throughout the development of the toolkit, we both reimplemented well-known visualizations and crafted novel designs to the test the expressiveness, effectiveness, and scalability of the toolkit. In this section we describe our experiences using prefuse to build this array of applications.

Existing Visualizations

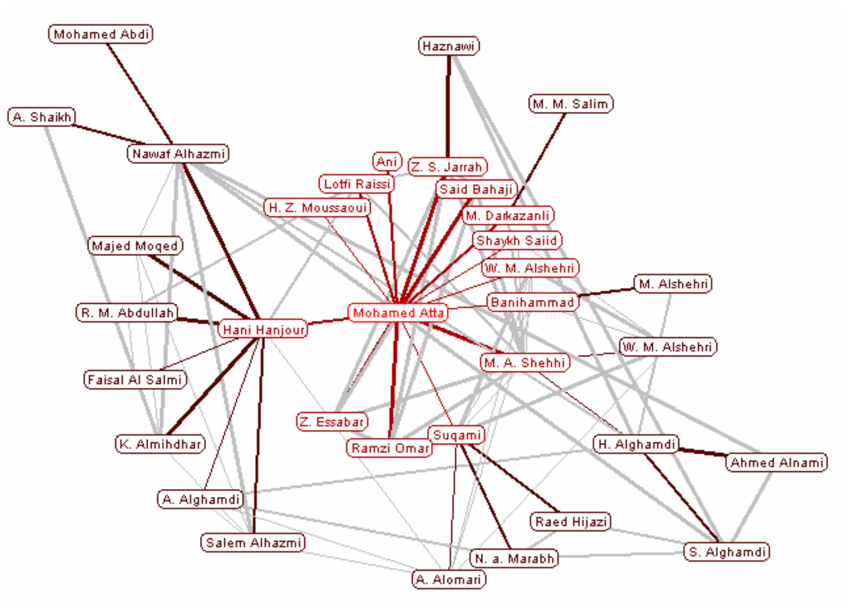


Figure 3. Animated radial layout of terrorist connections.

Animated Radial Graphs. The first prefuse application was a re-implementation of Yee *et al.*'s system for animated exploration of graphs using radial layout [49], shown visualizing a network of terrorists involved in the 9/11 attacks in Figure 3. Clicking a node in the visualization initiates an animated transition in which that node becomes the new center of the diagram. To avoid "clumping" during animation, nodes follow arced trajectories.

The application consists of 190 lines of code[2] and was built using three ActionLists. The first filters the graph data and computes a radial layout. The second animates between configurations in response to a focus change, updating colors and interpolating positions in polar coordinates. A third list updates color values, highlighting neighboring nodes in response to mouse-over events. As radial layout can suffer from occlusion when too many items are present, we added jitter by introducing an ActionList that briefly runs a force simulation using anti-gravity. Using prefuse's library components, it took 12 lines of code and 5 minutes of development time to add this novel customization.

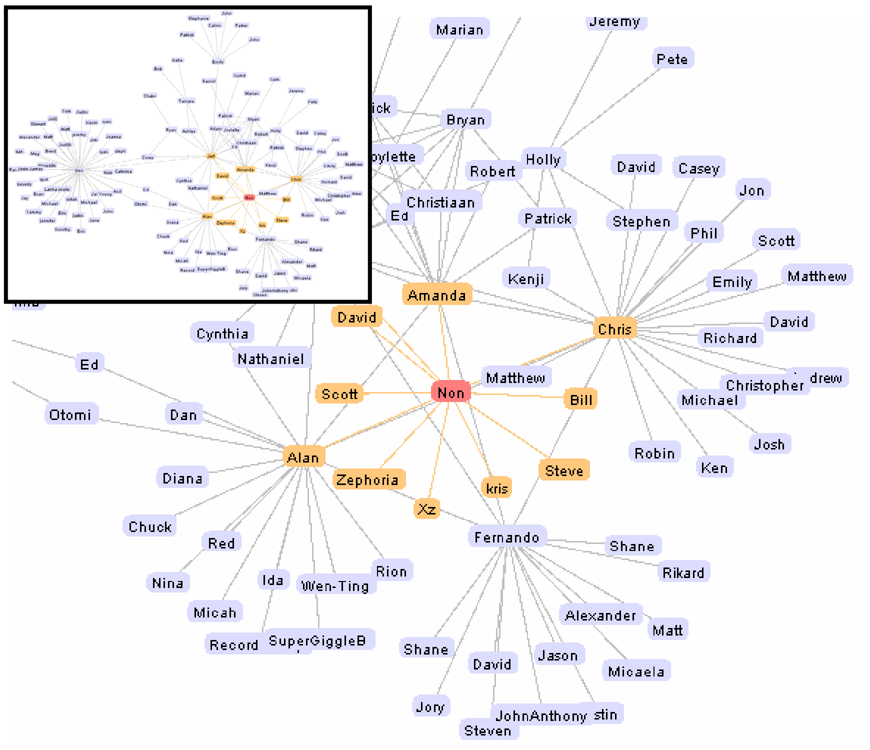


Figure 4. Zoomable force-directed layout of an online social network, including an overview display.

Force-Directed Layout. Force-based techniques are often used for graph layout, for example by creating a simulation in which nodes exert anti-gravity, edges act as springs, and friction or drag forces ensure that items settle. A well-known visualization utilizing these techniques is the Visual Thesaurus from plumbdesign [47]. We have built a similar application in prefuse, shown in Figure 4. The application consists of a single ActionList, parameterized to re-run every 20ms. The list consists of a filter, a force directed layout action, and a color function. In 3 lines of code we added controls for dragging nodes, panning, and zooming. With 5 more lines, we also added an overview display, bringing the total to 164 lines.

[2] All code line counts include import statements, which in some cases account for over one-third of the lines.

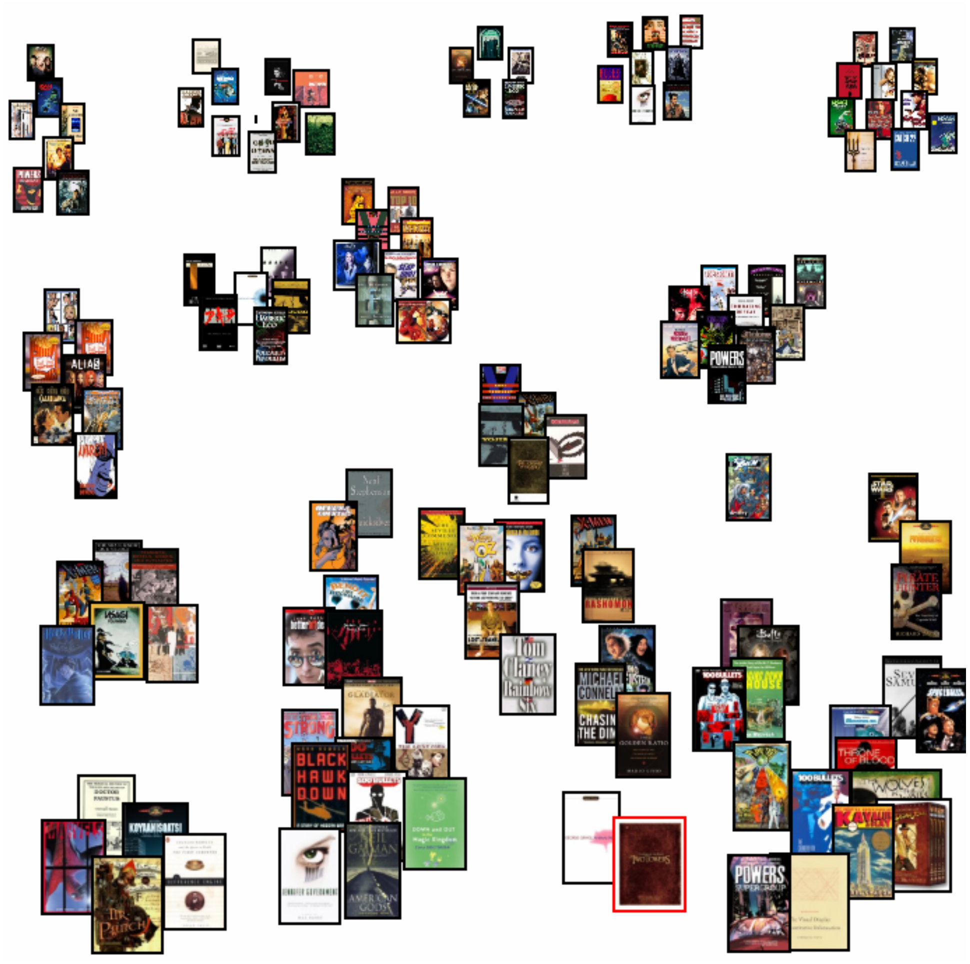
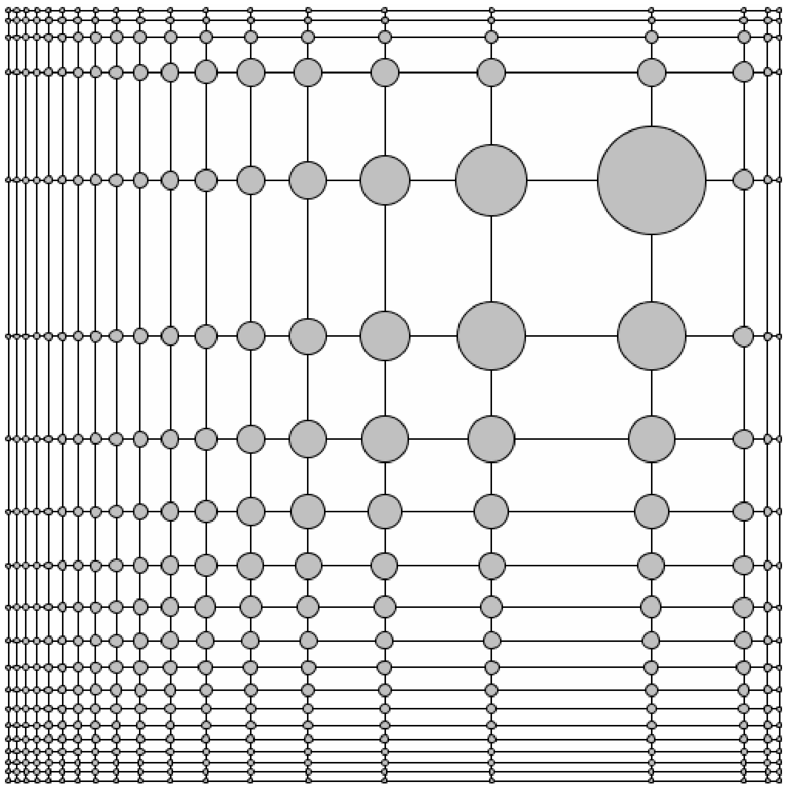


Figure 5. Data Mountain of a book and movie collection.

Data Mountain. We also used the force simulator in a re-implementation of the Data Mountain [41], which we used to visualize a collection of books and movies (shown in Figure 5). Images are automatically retrieved from the web by prefuse's image renderer and scaled according to an item's size value, which is assigned proportionally to an item's y-coordinate by a custom SizeFunction. Dragging a thumbnail moves it around the space, simultaneously initiating an ActionList containing a force-based layout and the aforementioned size function. Anti-gravity pushes nearby documents out of the way, while invisible springs anchor items near their original locations. The application was written in under 2 hours and consists of 211 lines of code.



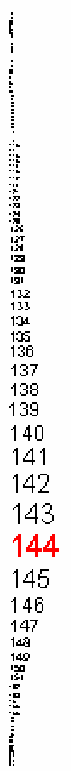


Figure 6a. Space Distortion demo. **6b.** A Fisheye Menu.

Fisheye Graphs and Menus. Figure 6a depicts a graph visualization using space distortion to present a focus+context view of a graph. Moving the mouse pointer causes the focus of the distortion to change accordingly. This was implemented using a run-once ActionList to filter the graph and compute the layout, and a second list containing a fisheye distortion action, run in response to a provided update control. The demo has 142 lines of code and was built in about an hour. Using a similar design, we also built a working prototype of fisheye menus [6], shown in Figure 6b. Using prefuse, we were able to build the prototype in just 20 minutes with 86 lines of code, the bulk of which consists of a simple layout that computes the item locations and scaling factor for the initial, undistorted view.

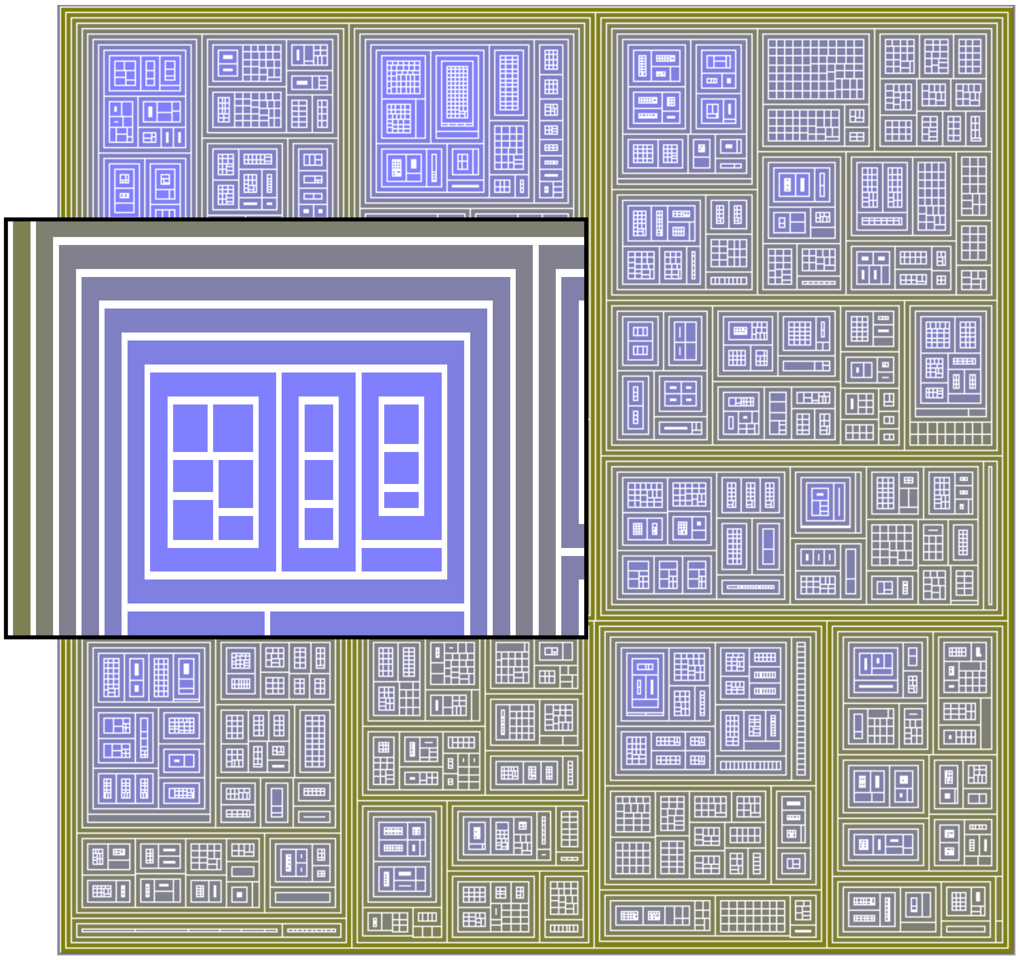


Figure 7. TreeMap of a nearly 8,000 node ontology. The callout shows a zoomed-in portion of the map.

TreeMaps. As an example of containment diagrams, we built a TreeMap browser using prefuse, shown visualizing an 8,000 node hierarchy in Figure 7. Each box represents a node in the tree and contains its descendants in nested boxes. The visualization is backed by a single ActionList containing a TreeFilter, a custom SizeFunction to assign node areas, a "squarified" tree map layout [10], and a ColorFunction that uses a color map to assign node color according to depth in the tree. The application was built in under a day, with most of the effort spent writing and testing the TreeMap layout for the prefuse library. The actual application consists of 133 lines of code.

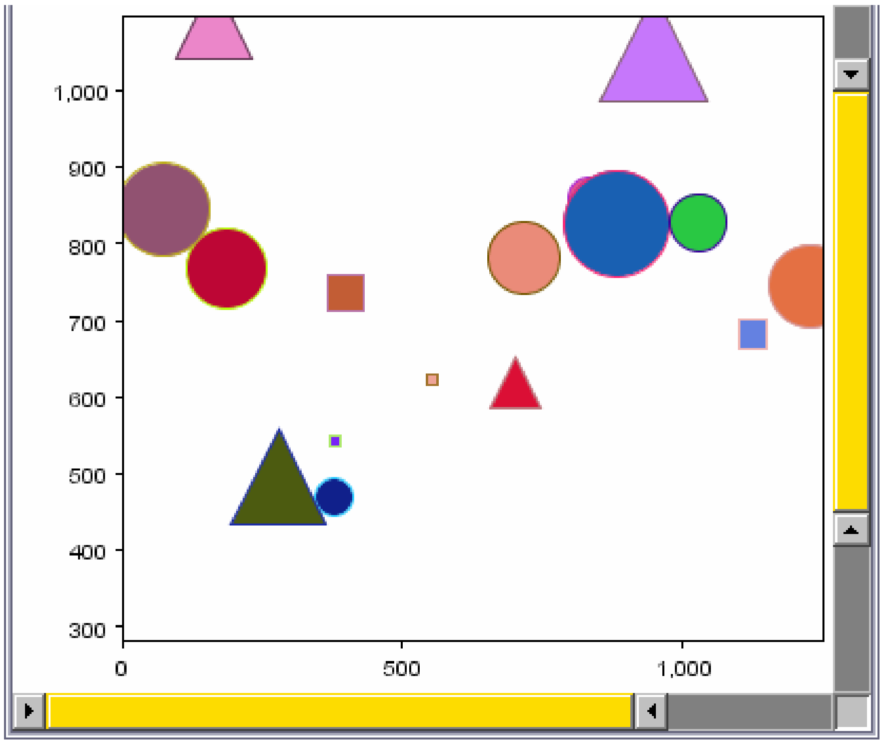


Figure 8. SpotPlot scatter plot. Range sliders control the scale and view of visualized data.

Starfield Displays. SpotPlot is a scatter plot viewer built by a colleague with whom we shared our toolkit. As shown in Figure 8, SpotPlot uses range sliders to control a filtered view of data—both the scatter plot display and the axis values update in response to the slider-specified ranges. SpotPlot uses a single ActionList with a custom filter, which uses the current range slider values to filter data elements, and a layout action that places items according to their (x,y) data values. A custom Renderer draws different shapes in response to node attributes. The app also uses a customized Display component, overriding the postPaint method from the Display class to draw the scatter plot axes. The application consists of 523 lines of code in 7 source files, written in under a week of part-time work.

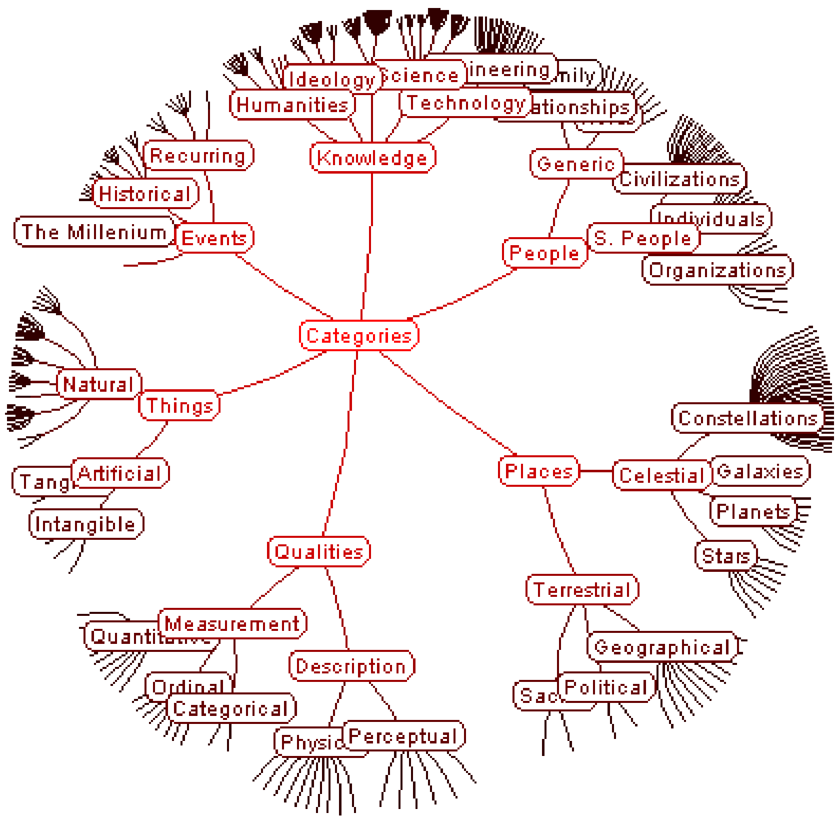


Figure 9. Hyperbolic Tree Browser.

Hyperbolic Tree. We also used prefuse to re-implement the popular hyperbolic tree browser [30], shown in Figure 9. Implementing the hyperbolic tree required writing a handful of new Action modules. The first was a hyperbolic layout routine that computes the coordinates of each data item in the complex plane, storing the coordinates as attributes of the visual items. Another Action was written to map these complex coordinates to actual screen locations, completing the layout. To add interactivity, a hyperbolic translation Action was added to compute coordinate translations in hyperbolic space, projecting the results back onto the complex plane. The translation module is run in response to individual mouse drags, but also doubles as an animator, interpolating between two positions in response to clicked items. Finally, we also introduced an Action to toggle the visibility of peripheral items, improving frame rates. In all, we wrote 631 lines of code in under three days, 372 in new Action modules and 259 in application code.

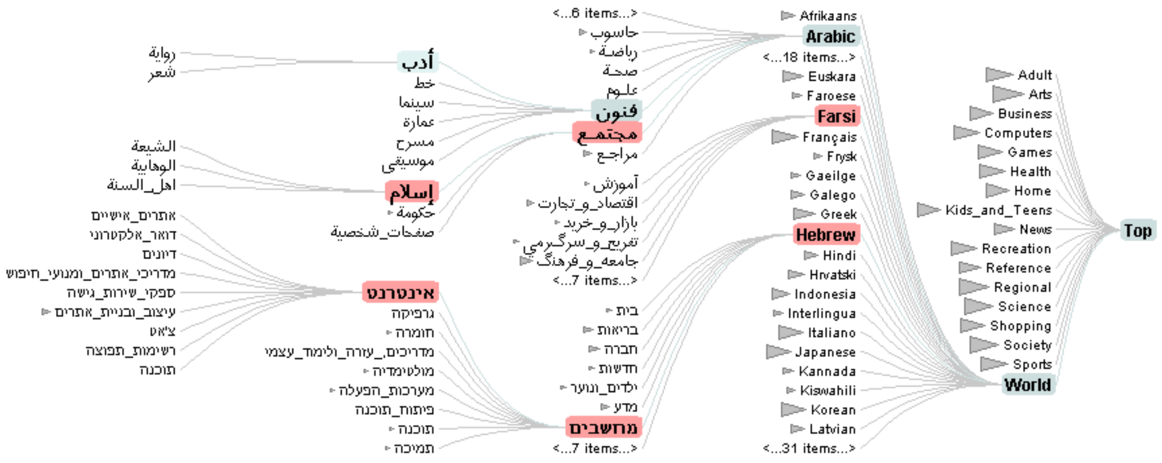


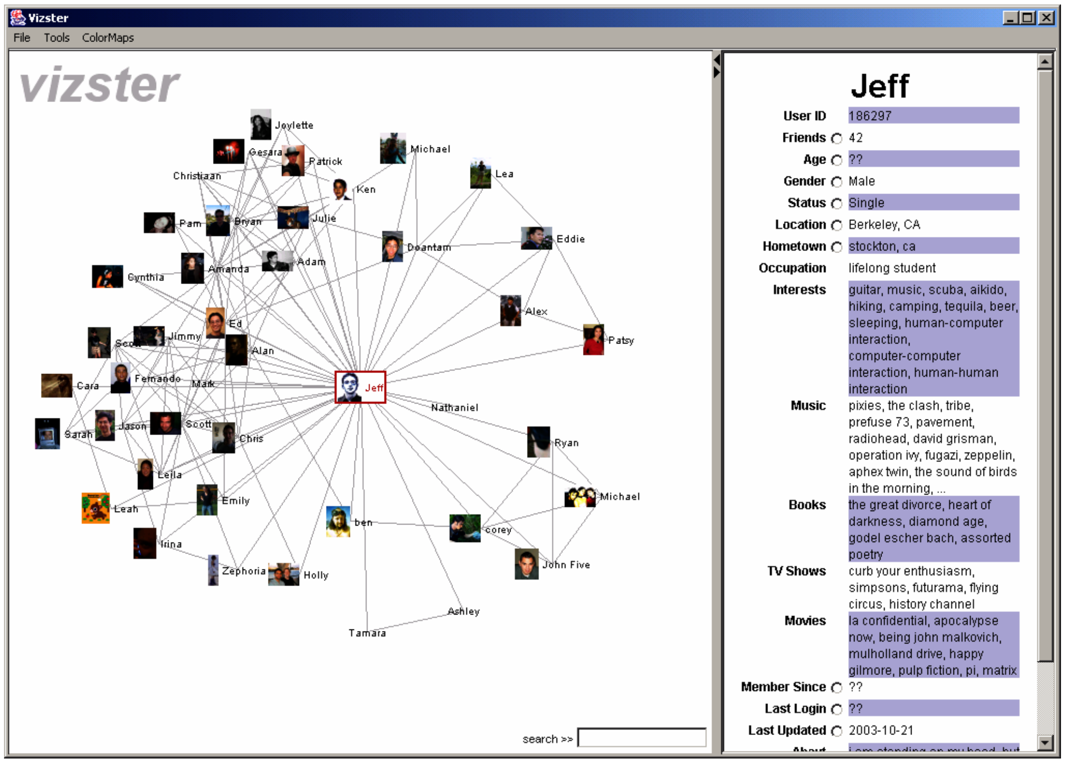
Figure 10. Degree-of-Interest Tree visualizing a 600,000 node web directory.

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 10 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, 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 that may be quite far apart in the tree [22].

The DOITree browser consists of 1929 lines of code, 1011 in reusable Action modules and 918 in application code. As we developed the app over a period of two months, 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 11). 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 the membership



Jeff

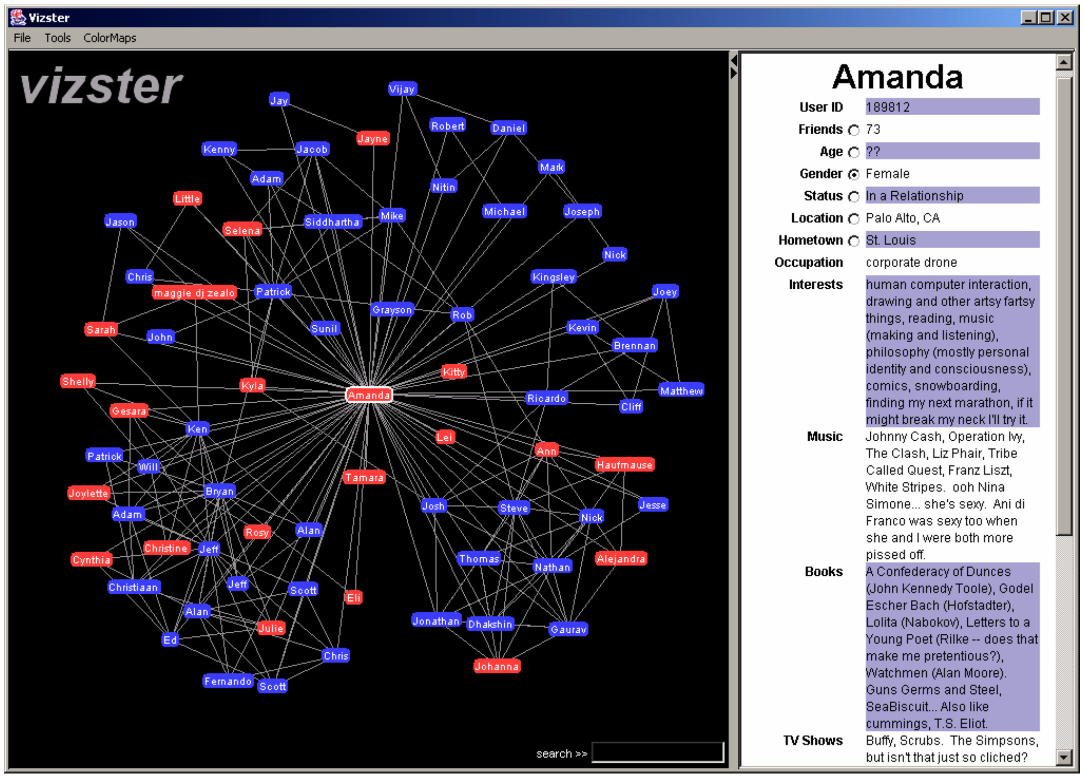
User ID	186297
Friends	42
Age	??
Gender	Male
Status	Single
Location	Berkeley, CA
Hometown	stockton, ca
Occupation	lifelong student
Interests	guitar, music, scuba, aikido, hiking, camping, tequila, beer, sleeping, human-computer interaction, computer-computer interaction, human-human interaction
Music	pixies, the clash, tribe, prefuse 73, pavement, radiohead, david grisman, operation ivy, fugazi, zeppelin, aphex twin, the sound of birds in the morning, ...
Books	the great divorce, heart of darkness, diamond age, godel escher bach, assorted poetry
TV Shows	curb your enthusiasm, simpsons, futurama, flying circus, history channel
Movies	la confidential, apocalypse now, being john malkovich, mulholland drive, happy gilmore, pulp fiction, pi, matrix
Member Since	??
Last Login	??
Last Updated	2003-10-21

Figure 11. Vizster in browsing mode, showing an ego-centric network of friendship relations. The panel on the right displays profile data for a selected person.

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 into the search box immediately causes both matching people in the graph display and matching text in the profile display to highlight.

In addition to the browsing mode described above, Vizster supports a comparison mode (see Figure 12), 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,



Amanda

User ID	189812
Friends ○	73
Age ○	??
Gender [✓]	Female
Status ○	In a Relationship
Location ○	Palo Alto, CA
Hometown ○	St. Louis
Occupation	corporate drone
Interests	human computer interaction, drawing and other artsy fartsy things, reading, music (making and listening), philosophy (mostly personal identity and consciousness), comics, snowboarding, finding my next marathon, if it might break my neck I'll try it.
Music	Johnny Cash, Operation Ivy, The Clash, Liz Phair, Tribe Called Quest, Franz Liszt, White Stripes. ooh Nina Simone... she's sexy. Ani di Franco was sexy too when she and I were both more pissed off.
Books	A Confederacy of Dunces (John Kennedy Toole), Godel Escher Bach (Hofstadter), Lolita (Nabokov), Letters to a Young Poet (Rilke -- does that make me pretentious?), Watchmen (Alan Moore). Guns Germs and Steel, SeaBiscuit... Also like cummings, T.S. Eliot.
TV Shows	Buffy, Scrubs. The Simpsons, but isn't that just so cliched?

Figure 11. Vizster in comparison mode, using color to display the genders of visualized friends.

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, interactors) 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 user 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, 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, 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. All subjects were able to load data from disk, construct working action lists, and subclass existing modules to customize processing. Subjects did not necessarily complete tasks in the order presented (they were told this was fine) and half encountered trouble at some point during development.

The most common difficulty for subjects was structuring data appropriately. For example, four subjects wanted to apply a radial layout in their design, but ran into trouble when they used a general graph filter and the radial layout algorithm, expecting the graph to have a tree imposed on it, threw an exception. Confusion also surrounded the use of individual filtering modules. While the interviews revealed all subjects grasped the general concept of filtering, one subject didn't realize that, as implemented, they were responsible not only for controlling what is visualized, but also constructing a separate topology of the visual items. This was confounded by an earlier toolkit design that was overly confusing, in which individual filters were used to process nodes and edges separately. This roadblock prevented the subject from finishing all the tasks.

In response to these issues, we subsequently redesigned the filters provided by the toolkit. Instead of separate modules for different data aspects (*e.g.*, node filters, edge filters), we now provide unified filters for filtering visual structures. Furthermore, we made the filters more robust to input data. For example, a TreeFilter will now automatically overlay a tree structure on filtered items even when the source data is a general graph, further taking advantage of the representational flexibility provided by the filtering abstraction.

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. (In fact, the study tasks were purposefully designed such that cut and paste strategies still required integrating across the various available code samples.) 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, suggesting that a toolkit's "user interface" is not just an API, but also associated materials (code samples, documentation), 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 1000 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 helped unearth additional user requirements, from bug fixes to the need for improved documentation. Overall, reaction to prefuse has been overwhelmingly positive, enabling users to create new visualizations of their own, many of whom report having only limited programming experience.

CONCLUSION

In this report 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 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 problems. In future work, we plan to introduce more powerful operations for manipulating source data, provide additional processing, rendering, and interaction components, and potentially develop a visual environment for application authoring. First and foremost, however, 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, interactive demonstrations, and video demonstrations are available at `http://prefuse.sourceforge.net`.

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. 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, and J. Meyer, *Toolkit Design for Interactive Structured Graphics*. Technical Report HCIL-2003-01, CS-TR-4432 , UMIACS-TR-2003-03, University of Maryland 2003.

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. *10th IEEE Symposium on Information Visualization (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. Runge-Kutta Method, From MathWorld. http://mathworld.wolfram.com/Runge-KuttaMethod.html

49. 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.