prefuse: a toolkit for interactive information visualization

Jeffrey Heer
Group for User Interface Research
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720-1776, USA
jheer@cs.berkeley.edu

Stuart K. Card
User Interface Research Group
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94301, USA
card@parc.com

James A. Landay
DUB Group
Computer Science & Engineering
University of Washington
Seattle, WA 98195-2350, USA
landay@cs.washington.edu

ABSTRACT
Conventional software toolkits for building graphical user interfaces are typically not well-suited for the demands of information visualization, in which thousands of interactive objects may be visible at any time and data items may regularly come in and out of existence. In this paper we present a set of higher-level abstractions for authoring scalable, highly-interactive visualizations of large data sets. These abstractions are realized in prefuse, an extensible user interface toolkit for visualizing both structured and unstructured data. The toolkit supports node-link diagrams, containment diagrams, and visualizations of unstructured (edge-free) data such as scatter plots and timelines. prefuse applies scalable abstractions for mapping abstract data into visual forms and manipulating visual data in aggregate, supporting design in a modular and principled fashion. To evaluate the architecture, we built a series of applications demonstrating the toolkit's power and flexibility. We also conducted a qualitative user study with eight programmers and found that, with the toolkit, non-experts could quickly build useful interactive visualizations.

Keywords
Information visualization, user interfaces, toolkits, graphs, trees, interaction techniques, navigation, 2D graphics, Java

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
Information visualization seeks to augment human cognition by leveraging human visual capabilities to make sense of large collections of abstract information [10]. Since the introduction of data graphics in the late 1700's [45], visual representations of abstract information have been used to demystify data and reveal otherwise hidden patterns. The advent of graphical interfaces has further enabled direct interaction with visualized information, giving rise to over a decade of information visualization research and 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 [18], there are technical roadblocks as well. One is that information visualization applications are difficult to build, requiring both mathematical and programming skills to implement complex layout algorithms and dynamic graphics, resulting in applications of increased complexity with more stringent performance demands. Another reason is that infovis applications do not lend themselves to "one size fits all" solutions. While successful visualizations often reuse established techniques, many do so in a way uniquely tailored to the semantics of the application domain (*e.g.*, [29]), requiring additional programming. Finally, while many visualization and interaction techniques have been developed, we lack a unified software framework for applying them in novel ways. This suggests the need for a toolkit approach, supporting a diversity of customized applications by providing high-level support for common visualization solutions in a reusable fashion.

While available GUI toolkits have significantly accelerated the development of user interfaces [35], infovis brings two unique scalability requirements for which current UI toolkits are not well-suited. The first is in terms of *performance*. Handling thousands of dynamic data items while maintaining real-time animated interaction is a technical challenge not well addressed by standard toolkits in which individual widgets incur a high overhead. Simply waiting for faster processors is not sufficient, as increases in computing power and networking have brought similar increases in the size of relevant data sets. Furthermore, dealing with thousands of on-screen data elements and perhaps millions of off-screen elements raises issues of *programming scalability*, affecting the program complexity and time costs programmers of such applications must endure. Suitable abstractions are needed for presenting and manipulating large collections of visualized data.

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. The basic 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 interactive visualizations. This comprises node-link diagrams, containment diagrams, and visualizations of unstructured (edge-free) data such as scatter plots and timelines. prefuse introduces abstractions for *filtering* source data into visualized content and using composable *actions* to perform batch processing of this content, allowing developers to manipulate content in aggregate. prefuse also includes a rich library of layout algorithms, navigation and interaction techniques, integrated search, and more.

The primary goal of prefuse is to facilitate visualization design by reducing implementation costs, allowing existing techniques to be combined in new ways, and promoting application modularity and extensibility. Furthermore, prefuse potentially *lowers the threshold* [35] for application design by abstracting much of the mathematics and systems engineering common to interactive visualizations.

To provide a modular toolkit in a principled manner, prefuse implements existing theoretical frameworks for information visualization [9,10,13]. These models decompose 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 1). Prior work has validated the model's expressiveness, providing a comprehensive taxonomy of visualization techniques [13]. prefuse implements this model by providing suitable abstract data structures, transforming source data into visualizable content through *filtering*, manipulating visual data using lists of composable *actions*, and mapping visual data into interactive views through a configurable rendering system. prefuse also demonstrates that these generalized abstractions can be provided without unduly sacrificing performance.

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 (including database connectivity) 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 modules
- A library of provided layout and distortion techniques
- Animation and time-based processing
- Graphics transforms, including panning and zooming
- A full force simulator for physics-based interfaces
- Interactor components for common interactions
- Integrated color maps and search functionality
- Event logging to support visualization evaluation

prefuse is written in the Java programming language using the Java2D graphics library. We decided to support only 2D visualizations, as the benefits and tradeoffs incurred by 3D visualization are still an unresolved topic of debate [14,15]. Furthermore, 3D programming is often more complicated, raising the threshold for end-programmer extensibility.

In the next section we describe the design of the prefuse toolkit. We then discuss the toolkit's evaluation, presenting applications demonstrating the toolkit's flexibility and power, and a qualitative user study demonstrating the usability of the prefuse API. Finally, we survey related work and conclude.

DESIGN OF THE PREFUSE TOOLKIT

We now describe the toolkit design (illustrated in Figure 1), 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 [16] 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.

The filtering abstraction fits comfortably in current design frameworks. In the data state model of [13], 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 [27]. 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 data set by using separate filters, and different views of a specific visualization by reusing the same filtered items.

[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.

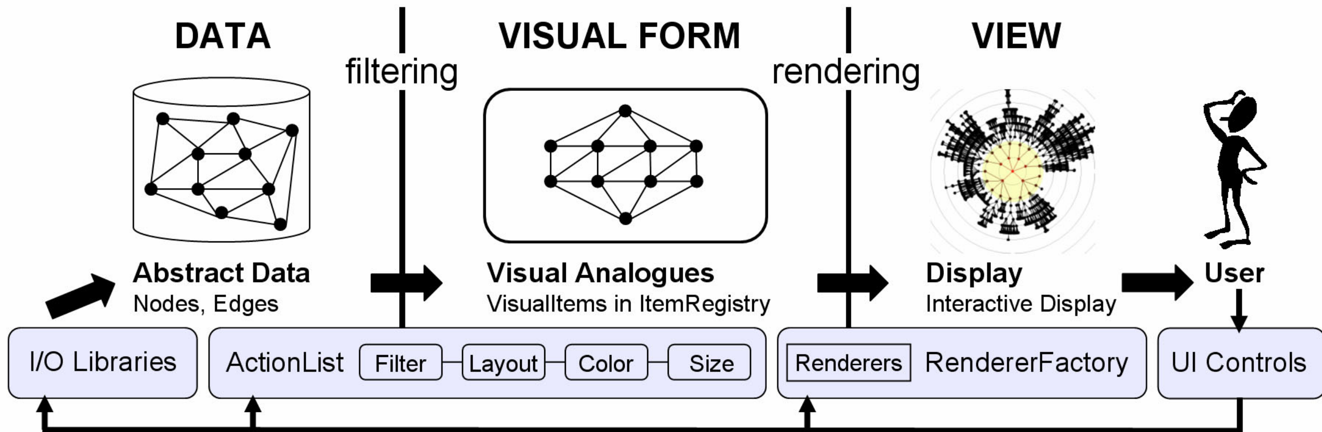

Figure 1. The prefuse visualization framework. Lists of composable actions filter abstract data into visualizable content and assign visual properties (position, color, size, font, *etc*). Renderer modules, provided on a per-item basis by a RendererFactory, draw the VisualItems to construct interactive 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 needed, 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 many focus+context schemes — by keeping only a working set of currently 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 [16,19]. 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 [22]. 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 [22] 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 the 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 [36] – 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 [10]).

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 allows 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, force-directed, top-down (Reingold-Tilford) [38], radial [48], indented outline, and tree map [8,42] 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 [41] and bifocal distortion [30].

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], building a quad-tree of elements annotated with center of mass values, 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 [47]) from toolkit users. Users can also write custom force functions and add them to the simulator.

Interactive Controls. Inspired by the Interactor paradigm [34], 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 [23].

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.

EVALUATION

In this section we present a set of applications built with the prefuse framework, verifying the toolkit's expressiveness and scalability, and the results of a study of the toolkit API.

Test Applications

Each iteration of toolkit design has included the design and implementation of applications to test toolkit expressiveness, application development rates, and scalability. Here we introduce some of these applications, which span existing systems and novel research. Where available, we include the time cost and lines of code needed to write the applications.

Animated Radial Layout

The first prefuse application was a re-implementation of Yee et al's system for animated exploration of graphs using radial layout [48], shown visualizing a social network of terrorists involved in the 9/11 attacks in Figure 2. Clicking a node in the visualization initiates an animated transition in which that node becomes the new center of the diagram. To avoid visual "clumping" during animation, the nodes follow arced trajectories. Our application also highlights a node and its neighbors upon mouse-over, facilitating exploration.

We built this application in prefuse using three ActionLists. The first filters the graph data and computes a radial layout. The second animates between configurations, updating item colors and interpolating node positions in polar coordinates. When clicked, a node is made the new focus and a focus listener schedules these lists to run. A third list updates color values in response to mouse-over events. This application consists of 190 lines of code.

A radial layout can suffer from occlusion when too many items are present. To improve legibility, we added jitter to the application by including 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 feature.

Force-Directed Layout

Using prefuse's provided force simulator, it is very easy to create physics-based interfaces. Force-based techniques are commonly used for graph layout, for example by creating a simulation in which nodes exert anti-gravity to push each other away, edges act as springs, and friction or drag forces are used to ensure that items settle. A well-known visualization utilizing these techniques is the Visual Thesaurus from **plumb**design [46]. We have built a similar application in prefuse, shown in Figure 3 visualizing an online social network. 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.

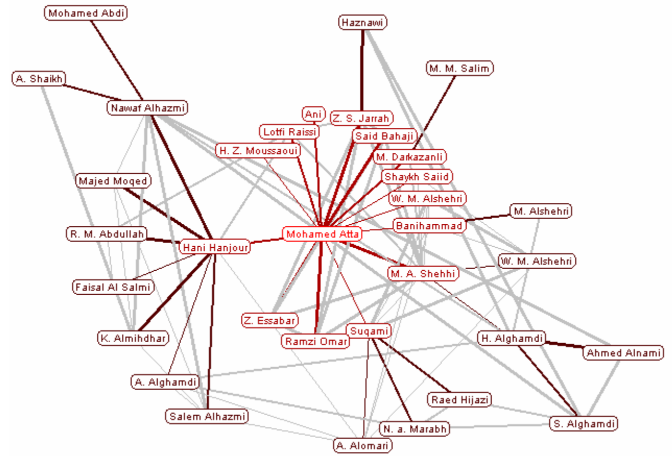

Figure 2. Animated radial layout of terrorist connections.

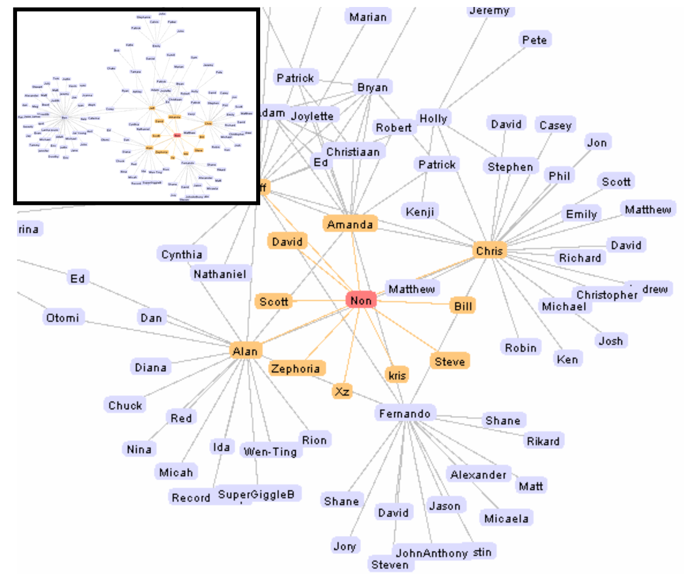

Figure 3. Zoomable force-directed layout of an online social network, including an overview display.

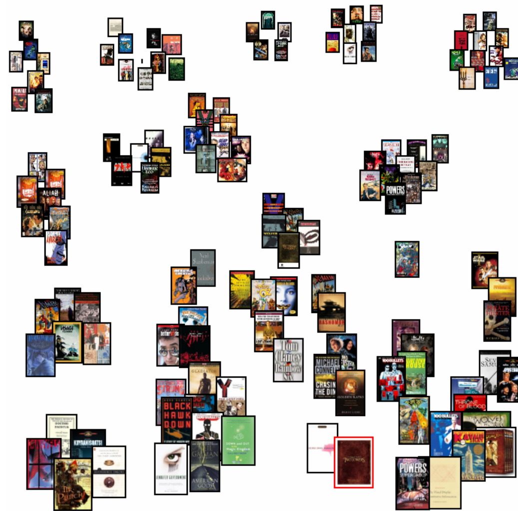

Figure 4. Data Mountain of a book and movie collection.

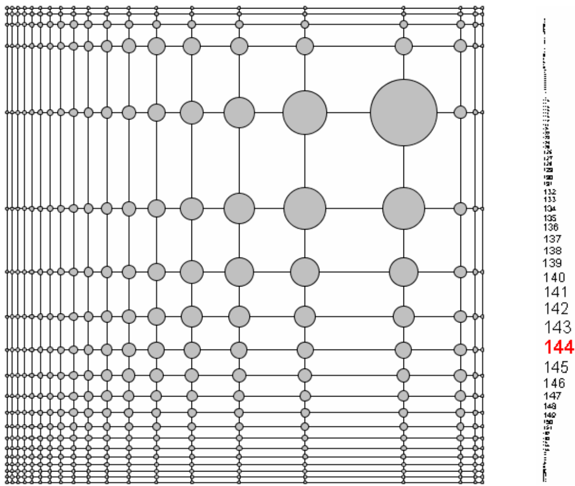

Figure 5a. Space Distortion demo. **5b.** A Fisheye Menu.

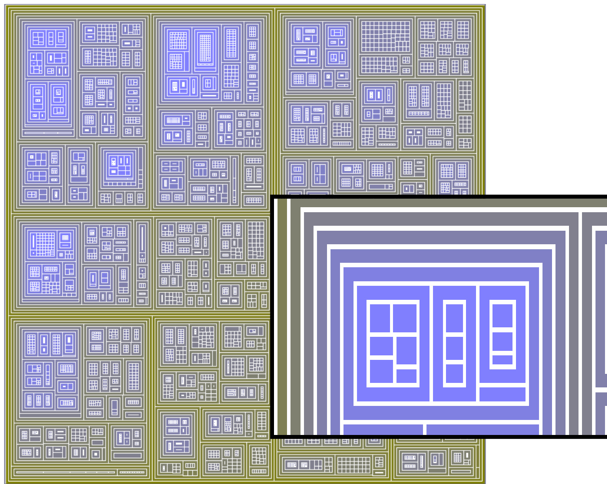

Figure 6. TreeMap of a nearly 8,000 node ontology. The callout shows a zoomed-in portion of the map.

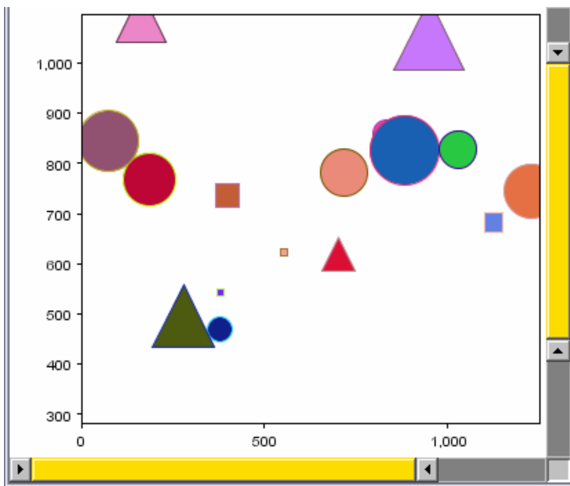

Figure 7. SpotPlot scatter plot. Range sliders control the scale and view of visualized data.

Data Mountain

Another example application using the prefuse force simulator is a re-implementation of the Data Mountain, which allows users to visually organize documents in a 2½-dimensional environment [39]. Figure 4 shows a Data Mountain built using prefuse, visualizing a colleague's collection of books and movies. Image URLs are included as data attributes, allowing images to be automatically retrieved from the web by prefuse's image renderer. Images are scaled according to an item's size value, which is set 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 is used to push nearby documents out of the way, while invisible springs are used to anchor items to their locations. As a result, perturbed documents return to their original positions after a dragged document has passed nearby. These spring anchor points are updated when the drag completes. The entire application was written in under 2 hours and consists of 211 lines of code.

Distortion-Based Navigation

Figure 5a depicts a graph visualization using space distortion to present a focus+context view. 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. 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 [5], shown in Figure 5b. 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.

TreeMaps

As an example of containment diagrams, we built a TreeMap browser using prefuse, shown visualizing an 8,000 node hierarchy in Figure 6. 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 [8], and a ColorFunction that uses a color map to assign node color according to depth in the tree. With two extra lines of code we added panning and zooming to the display, allowing users to explore the various "cities" in the figure more closely. 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.

Scatter Plot

SpotPlot is a scatter plot viewer built by a colleague with whom we shared our toolkit. As shown in Figure 7, 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. We were encouraged that other researchers could pick up our toolkit and quickly build useful components.

Hyperbolic Tree Browser

In a more strenuous test of the toolkit, we used prefuse to re-implement the well-known hyperbolic tree browser [28], shown in Figure 8. Implementing the hyperbolic tree required writing a number of new Action modules. The first was a hyperbolic layout routine that takes a filtered tree, computes the hyperbolic coordinates of each data item (including control points for curved edges), and projects the coordinates on to the complex plane, storing the coordinates as attributes of visualized items. Another Action was written to map these complex coordinates to the actual on-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 items on the periphery of the display, improving interactive frame rates for large trees.

Though implementing the hyperbolic tree proved to be complicated, the prefuse architecture allowed the design to be mapped into individual, reusable Action modules and sped up design by providing highly-customizable rendering and animation support. In all, we wrote 631 lines of code, 372 in new Action modules and 259 in application code. Actual development time was less than three days.

Degree-of-Interest Trees

Finally, the most intricate prefuse application built to date is an enhanced version of the Degree-of-Interest Tree (DOITree) browser [11,19]. DOITrees are tree visualizations featuring multiple focus+context techniques, including the use of degree-of-interest functions [16] 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 9 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.

DOITrees are implemented using four ActionLists, all of which are sequentially scheduled in response to mouse clicks. 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 "zoomed-out" views of the data, facilitating exploration of different foci that may be quite far apart in the tree [19].

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 [19], provide integrated handling of search results, and customize item appearances to specific application domains. This application also demonstrates the toolkit's scalability, maintaining real-time interaction with data sets containing nearly a million items.

Finally, the applications above showcase prefuse's support of module reuse and extensibility, using provided modules (*e.g.*, filters, layouts, renderers, interactors) across visualizations, while also making it easy for both ourselves and others to introduce customized components.

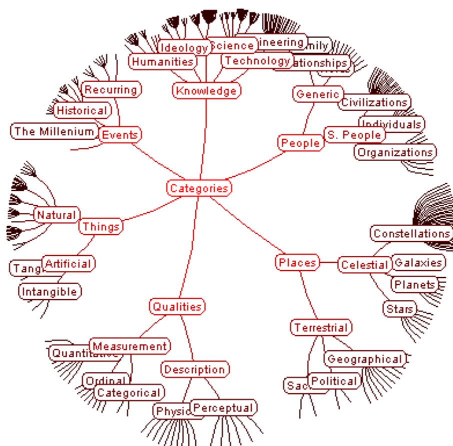

Figure 8. Hyperbolic Tree Browser.

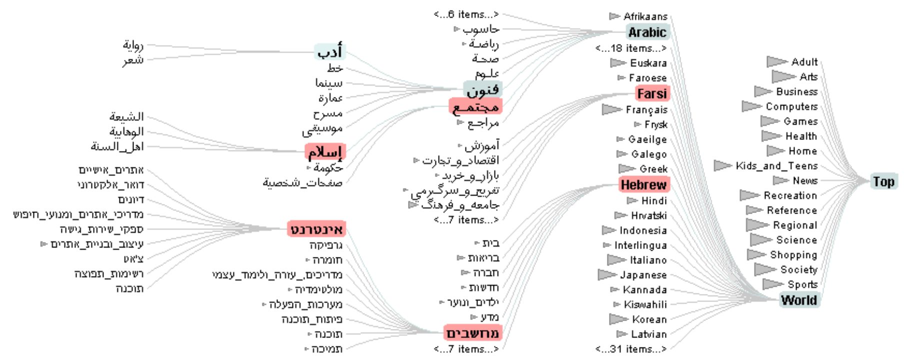

Figure 9. Degree-of-Interest Tree visualizing a 600,000 node web directory.

Qualitative User Study

We wanted to better understand the learnability and usability of the prefuse API for other programmers. In particular, toolkit abstractions such as filtering and runnable action lists might seem foreign to some programmers, constituting the *threshold* for toolkit use [35]. To investigate these concerns, we adopted the evaluation method of [26] and conducted a user study of the prefuse API, observing 8 programmers use 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 walk through 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. Subjects were asked to recount their approach to designing their application, and asked to describe, in their own words, different toolkit concepts such as filtering and action lists. 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. Four subjects wanted to apply a radial layout in their design, but ran into troubles when they filtered a general graph structure and the radial layout algorithm, expecting a tree structure, threw an exception. Confusion also surrounded the use of individual filtering modules. While the interviews revealed that 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 how the visualized items are linked together. 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 overlay a tree structure on filtered items even 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."

Evaluation Summary

Through the evaluation process, the toolkit has made great strides. Both the test applications 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 Actions and ActionLists, study subjects built useful visualizations in under an hour, and toolkit users have appreciated the accompanying extensibility. The creator of the SpotPlot application told us "I liked being able to define and think about the individual components separately. For example, I like the idea of being able to plug in a jitter component later."

We have found that iterative design, a proven method for developing user interfaces, has also proven a valuable design method for software toolkits, where the users are programmers and the interface is an API [26]. prefuse is now being used by other researchers and by students in an infovis course. We are following this work in a longitudinal study of toolkit use, and look forward to the insights (and dilemmas) this next round of evaluation will bring.

RELATED WORK

The design of prefuse extends a large body of related work, categorized here into three areas: information visualization, graph drawing, and user interface and graphics toolkits.

Information Visualization

prefuse has been inspired by the rich body of information visualization work that has preceded it. This includes a host of designs, including TreeMaps [8,42], Cone Trees [40], Perspective Walls [32], StarField displays [1], Hyperbolic trees [28], DOITrees [11,19], SpaceTrees [37], and more. In addition, prefuse leverages transformation and navigation techniques, including focus+context schemes [16], space distortion [30], point-of-interest navigation [31], and panning and zooming [23,36]. prefuse also inherits from the Information Visualizer (IV) [12], perhaps the first integrated framework for infovis. prefuse's activity scheduler works analogously to the IV's governor, overseeing activities and adjusting frame rates as necessary. prefuse furthers this work by introducing high-level support for application design, including theoretically-grounded abstractions for processing visualized data and a library of reusable components.

Equivalent to the information visualization reference models of [9] and [10], Chi proposed the data state model [13] as a framework for structuring infovis applications. prefuse contributes a generalized implementation of this theoretical model to support a wide range of visualization designs [13]. prefuse's data structures provide *Analytical Abstractions* of data. Filter Actions compute *visualization transformations*, creating a *Visualization Abstraction* of VisualItems in an ItemRegistry. Other Actions operate upon this abstraction to set visual properties. The RendererFactory and Renderers then provide *visual mapping transformations* to create *Views*, realized as Display instances.

Most information visualization research to date has consisted of exploring the space of successful designs and techniques. We believe the field is now moving into a second phase in which this knowledge is applied in a principled manner to achieve more powerful visualization environments, as currently exemplified by the Polaris database visualization system [44]. prefuse furthers this trend by situating infovis research and models in an extensible, modular framework.

Graph Drawing

For decades, the graph drawing community has devised algorithms for the aesthetic layout of graph structures. These are given thorough coverage in [4]. Perhaps the best known software for graph drawing is the excellent graphviz package from AT&T [17], which creates static images of graphs.

A graph drawing project with similar aims as prefuse is Marshall et al.'s Graph Visualization Framework (GVF) [33], which uses a related notion of filtering for determining graph visibility, and also includes a library of layout routines. The GVF, however, has only limited support for full interactivity and animation.

There are several other research and commercial graph drawing systems, including pajek [3] and products from Tom Sawyer and yWorks. prefuse is differentiated by its focus on interactivity and design flexibility. For example, one could not build a DOITree with these tools. In recent years the graph drawing community has moved towards more interactive solutions, and we hope prefuse contributes to this.

User Interface and Graphics Toolkits

prefuse also leverages previous work on user interface toolkits, such as pioneering work on input abstractions like the model-view-controller [27] and interactor [34] paradigms, and the rich history and lessons learned from toolkit development [35]. This includes early systems for graph layout and editing [20,24] and for including animation in user interface toolkits [22].

More recent toolkits include support for advanced graphics capabilities, such as SATIN [21], a toolkit for pen-based interaction, and the Jazz [6,7] and Piccolo [6] toolkits for zoomable user interfaces. These toolkits use a scenegraph abstraction to structure interface layout and support animation and zooming. prefuse shares with these toolkits the use of lightweight glyphs (*i.e.,* VisualItems) instead of full widgets and support for advanced graphics and animation. Similar to Jazz, prefuse utilizes aspects of "minilithic" design [6,7], for example by decoupling interactive objects from rendering to provide increased functionality. The use of prefuse's RendererFactory to achieve dynamic rendering also parallels SATIN's multi-views. In contrast, however, prefuse is specifically designed for infovis applications, and provides higher-level abstractions for presentation, navigation, and batch processing of interactive objects.

Finally, concepts from 3D graphics toolkits, such as OpenGL [43] and VTK [25], have influenced prefuse. Though different in application and implementation, the ActionList was inspired by 3D rendering pipelines. While 3D toolkits provide a wide array of functionality, they impose a steeper learning curve and operate at a lower level. prefuse simplifies design by providing targeted abstractions for information visualization and an integrated framework supporting common presentation and interaction techniques.

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.

Test applications built with the toolkit demonstrate the flexibility and performance of the prefuse architecture. A user study showed that programmers could 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 software and interactive demonstrations are available on the web at `http://prefuse.sourceforce.net`.

ACKNOWLEDGMENTS

We would like to thank our colleagues at Berkeley and PARC, particularly Alan Newberger, Jock Mackinlay, Ed Chi, Scott Klemmer, and Lance Good, for their insight and comments. We also profusely thank all the subjects in our user study. The first author was supported by an NDSEG fellowship.

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. Bederson, B.B. Fisheye Menus. *UIST'00*. pp. 217-225, 2000.
6. 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.
7. Bederson, B.B., J. Meyer, and L. Good. Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java. *UIST'00*. pp. 171-180 2000.
8. 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.
9. Card, S.K., Information Visualization, in *The Human-Computer Interaction Handbook*. Lawrence Erlbaum Associates, 2002.
10. Card, S.K., J.D. Mackinlay, and B. Shneiderman, *Readings in Information Visualization: Using Vision to Think*. San Francisco, California: Morgan-Kaufmann, 1999.
11. Card, S.K. and D. Nation. Degree-of-Interest Trees: A Component of an Attention-Reactive User Interface. *Advanced Visual Interfaces*. 2002.
12. Card, S.K., G.G. Robertson, and J.D. Mackinlay. The Information Visualizer, an Information Workspace. *CHI'91*. pp. 181-188 1991.
13. Chi, E.H. A Taxonomy of Visualization Techniques Using the Data State Reference Model. *InfoVis '00*. pp. 69-75 2000.
14. Cockburn, A. and B. McKenzie. Evaluating the Effectiveness of Spatial Memory in 2D and 3D Physical and Virtual Environments. *CHI'02,* Minneapolis, MN. pp. 203-210 2002.
15. Czerwinski, M., D.S. Tan, and G.G. Robertson. Women Take a Wider View. *CHI'02*. pp. 195-202, 2002.
16. 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.
17. Graphviz. http://www.research.att.com/sw/tools/graphviz/
18. Grokking the Infoviz, *Economist Technology Quarterly*, June 2003.
19. Heer, J. and S.K. Card. DOITrees Revisited: Scalable, Space-Constrained Visualization of Hierarchical Data. *Advanced Visual Interfaces*. Gallipoli, Italy, May 2004.
20. Henry, T.R. and S.E. Hudson. Interactive Graph Layout. *UIST'91*. pp. 55-64, November 1991.
21. Hong, J.I. and J.A. Landay. SATIN: A Toolkit for Informal Ink-Based Applications. *UIST'00*. pp. 63-72 2000.
22. Hudson, S. and J.T. Stasko. Animation Support in a User Interface Toolkit: Flexible, Robust, and Reusable Abstractions. *UIST'93*. pp. 57-67, 1993.
23. Igarashi, T. and K. Hinckley. Speed-Dependent Automatic Zooming for Browsing Large Documents. *UIST'00*. pp. 139-148, 2000.
24. Karrer, A. and W. Scacchi. Requirements for an Extensible Object-Oriented Tree/Graph Editor. *UIST'90*. pp. 84-91, October 1990.
25. *The Visualization Toolkit User's Guide*: Kitware, Inc., 2003.
26. Klemmer, S.R., J. Li, J. Lin, and J.A. Landay. Papier-Mâché: Toolkit Support for Tangible Input. *CHI'04,* Vienna, Austria 2004.
27. 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.
28. 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.
29. Lee, B., C.S. Parr, D. Campbell, and B. Bederson. How Users Interact with Biodiversity Information Using Taxontree. *Advanced Visual Interfaces*. Gallipoli, Italy 2004.
30. 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.
31. 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.
32. Mackinlay, J.D., G. Robertson, and S.K. Card. The Perspective Wall: Detail and Context Smoothly Integrated. *CHI91*. pp. 173-179 1991.
33. 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.
34. Myers, B.A., A New Model for Handling Input. *ACM Transactions on Information Systems*, 1990. **8**(3): p. 289-320.
35. 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.
36. Perlin, K. and D. Fox. Pad: An Alternative Approach to the Computer Interface. *SIGGRAPH'93*. pp. 57-64, 1993.
37. 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.
38. Reingold, E.M. and J.S. Tilford, Tidier Drawings of Trees. *IEEE Transactions of Software Engineering*, 1981. **SE-7**: p. 21-28.
39. 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.
40. Robertson, G.G., J.D. Mackinlay, and S.K. Card. Cone Trees: Animated 3D Visualizations of Hierarchical Information. *CHI'91*. pp. 189-194, 1991.
41. Sarkar, M. and M.H. Brown. Graphical Fisheye Views of Graphs. *CHI'92*. pp. 83-91, May 1992.
42. Treemaps for Space-Constrained Visualization of Hierarchies. 1998. http://www.cs.umd.edu/hcil/treemap-history/
43. Shreiner, D., M. Woo, J. Neider, and T. Davis, *OpenGL Programming Guide*. Addison-Wesley, 2003.
44. 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).
45. Tufte, *The Visual Display of Quantitative Information*. Graphics Press, 1983.
46. Visual Thesaurus. http://www.visualthesaurus.com
47. Runge-Kutta Method, From MathWorld. http://mathworld.wolfram.com/Runge-KuttaMethod.html
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.