The InfoVis Toolkit

Jean-Daniel Fekete[1]
INRIA Futurs/LRI
Bat. 490
Université Paris-Sud
F91405 Orsay Cedex, France
Tel: 1-33-1-69-15-34-60

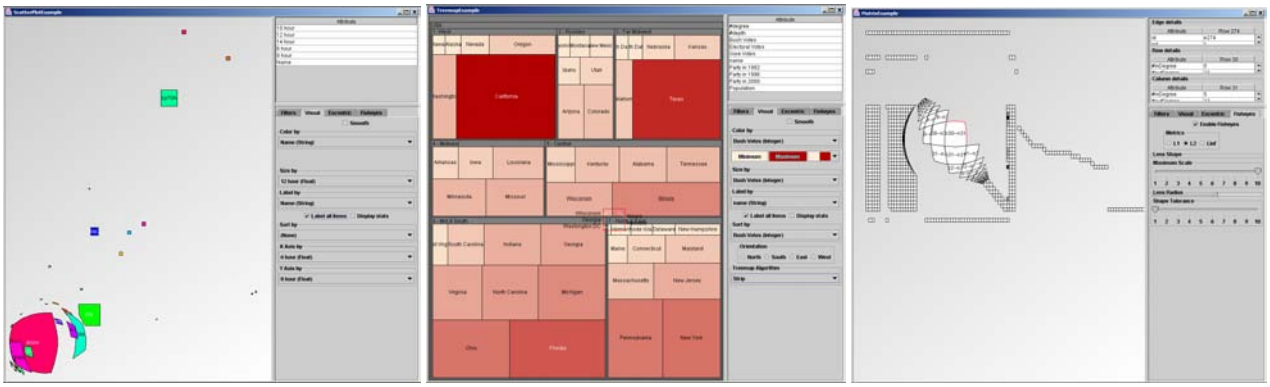

Figure 1: Examples of Scatter Plot, Treemap and Graph Visualizations Built with the InfoVis Toolkit

ABSTRACT

This article presents the *InfoVis Toolkit*, designed to support the creation, extension and integration of advanced 2D Information Visualization components into interactive Java Swing applications. The InfoVis Toolkit provides specific data structures to achieve a fast action/feedback loop required by dynamic queries. It comes with a large set of components such as range sliders and tailored control panels required to control and configure the visualizations. These components are integrated into a coherent framework that simplifies the management of rich data structures and the design and extension of visualizations. Supported data structures currently include tables, trees and graphs. Supported visualizations include scatter plots, time series, parallel coordinates, treemaps, icicle trees, node-link diagrams for trees and graphs and adjacency matrices for graphs. All visualizations can use fisheye lenses and dynamic labeling. The InfoVis Toolkit supports hardware acceleration when available through Agile2D, an implementation of the Java Graphics API based on OpenGL, achieving speedups of 10 to 200 times.

The article also shows how new visualizations can be added and extended to become components, enriching visualizations as well as general applications.

CR Categories: I.3.6 [Methodology and Techniques] Graphics data structures and data types; H.5 [User Interfaces] Graphical User Interface, Benchmarking; C.4 [Performance of Systems]: Design Studies.
Keywords: Information Visualization, Toolkit, Graphics, Integration.

[1] Email: Jean-Daniel.Fekete@inria.fr

IEEE Symposium on Information Visualization 2004
October 10-12, Austin, Texas, USA
0-7803-8779-1/04/$20.00 ©2004 IEEE

In recent years, Information Visualization has become popular outside its research community, both in industry and research. It is recognized as an important medium for communication, exploration and analysis in Data Mining, biology, sociology or cartography, to name a few.

Despite its well understood potential, information visualization applications are difficult to implement. They require a set of components – such as range sliders or fisheye lenses – and mechanisms – such as dynamic queries – that are not available or not well supported by traditional GUI toolkits. The literature on Information Visualization is large and resources describing the concrete implementation of key components are sometimes hard to find.

This article describes the *InfoVis Toolkit*: a coherent software architecture and a set of Java-based components designed to support the creation of information visualization applications and components for a large set of data structures. Its key features are:

- Generic data structures suited to visualization
- Specific algorithms to visualize these data structures
- Mechanisms and components to perform direct manipulation on the visualizations
- Mechanisms and components to select, filter and perform well-known generic information visualization tasks
- Components to perform labeling and spatial deformation.

To paraphrase Alan Kay, a good toolkit should be designed so that simple things become simple to do and complex things become possible. The article describes some simple applications which have been simple to do. The toolkit is also used for complex applications such as [16]. Another important point raised by Brian Gaines [15] is the ability to replicate novel ideas as a mandatory step towards the evolution of a science. Every year, new information visualization ideas are presented but their replication and integration into existing applications is very

difficult and sometimes never happens. Currently, there is no integrated toolkit facilitating the quick replication and integration of novel information visualization techniques, slowing down the development of the domain.

This article describes the general framework of the toolkit and some of its specific parts followed by a survey of existing tools compared to the InfoVis Toolkit. The last section demonstrates how new visualizations can be created and recent visualization techniques integrated into InfoVis. The conclusion then outlines areas of future work.

2. THE INFOVIS TOOLKIT

The InfoVis Toolkit is a Java library and software architecture relying on the Swing GUI and organized around five main parts: tables, columns, visualizations, components and input/ output. Figure 2 details the parts described below.

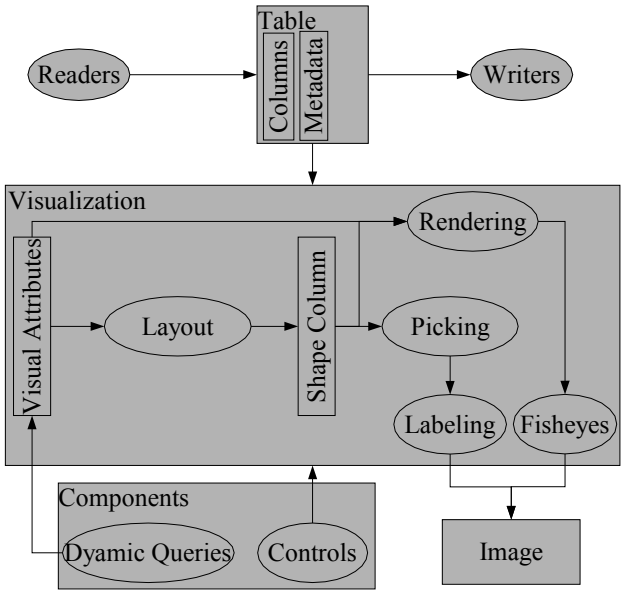

Figure 2: Internal structure of the InfoVis Toolkit. Squares represent data structures whereas ellipses represent functions.

The InfoVis toolkit provides a *unified underlying data structure* based on tables. Representing data structures with tables improves the memory footprint and performance, compared to ad-hoc data structures used by other specialized InfoVis applications. Any data structure can easily be implemented on top of tables and still expose a high-level Object Oriented interface for ease of programming.

Layout algorithms are encapsulated into *Visualization* components that map data structures into visual shapes. Visualizations natively support dynamic labeling and fisheye views.

Using this unified framework, a large number of interactive components required by information visualization are made generic and reusable across all the concrete data types and visualizations. These components are: dynamic queries and filters, selection, sorting and visual attributes manipulation.

The InfoVis Toolkit currently supports three concrete data structures: tables, trees and graphs. For each data structure, it supports several visualizations: time series, parallel coordinates and scatter plots for tables, node-link diagrams and treemaps for trees, node-link diagrams and adjacency matrices for graphs.

We have also added experimental support for accelerated graphics based on the OpenGL API to allow for richer graphics attributes and faster rendering.

2.1 Tables and Columns

A table is a list of named columns plus metadata and user data. A column manages rows of elements of homogeneous type, i.e. integers, floating points or strings. The elements are indexed so columns are usually implemented with primitive arrays. Some rows can be undefined. This mechanism is important because in real data sets, values may be missing. Allowing undefined elements is also very useful for representing general data structures.

For columns containing Java Objects and derived types, undefined rows contain the null value. For scalar types, an associative structure keeps track of undefined rows. This implementation is fast and efficient for dense columns, where most rows are defined. For sparse attributes – such as an XML structure encoded as a tree where each element may have a set of attributes – we provide sparse column implementations also based on associative structures. Even for these associative structures, using integers as keys is much faster than using objects.

Columns also support the following features:

- they contain metadata, e.g. to express that an integer column contains categorical or numeral values;
- they can trigger notifications when their content is modified. Since columns are often modified in large chunks, notification can be deferred;
- they support formatting for input and output so, for example, dates can be stored in columns of "long integers" data types and still appear as dates when read or displayed. This is important because scalar attributes are more space and time efficient than the equivalent complex objects.

A data set is stored as a table where each row represents a record and each column an attribute. This is natural for tabular data sets, but we also represent trees and graphs with this data structure. Trees and graphs are implemented as wrappers on top of tables with topological information represented by internal columns. By convention, an internal column is a normal column with a special prefix in front of its name. It can contain internal information, e.g. topological or data synthesized from other columns. These columns are not saved to files by data writers and not available for dynamic queries, with some exceptions.

Representing the topology of a tree consists in adding a "parent", "first child" and "next sibling" column [18]. More columns are created on demand if performance requires so, for instance for the degree of nodes (number of children), the sorted children list or the children depth. Since these attributes are synthesized from the basic topological structure and should be recomputed when the topology changes, they are not created by default. The toolkit supports a synchronization mechanism to trigger the re-computation of synthesized columns when their dependencies are changed. This re-computation occurs rarely in real situations since the data sets are usually not modified once they have been loaded. With this synchronization mechanism, any synthesized value can be turned into a column and handled like any other attribute value for dynamic queries or more general filtering.

Internal columns are also used for selection and dynamic filtering. Selection is managed through a column of boolean values (a row is selected when its column value is true) whereas dynamic filtering uses a column of bit sets (see section 2.3.) Boolean columns implement the Java ListSelectionModel interface and Tables implement the TableModel interface, enabling their integration into standard Java components.

As shown in Figure 3, creating and manipulating a tree based on a table is still done using an object-oriented programming style.

```
Tree tree = new DefaultTree();
IntColumn date = new IntColumn("date");
date.setFormat(new UTCDateFormat());
StringColumn name = new StringColumn("name");
tree.addColumn(date);
tree.addColumn(name);
int n1 = tree.addNode(Tree.ROOT);
name.setValue(n1, "Root");
date.setValue(n1, "13/Mar/2004 11:23:30");
int n2 = tree.addNode(n1);
…
```

Figure 3: Example of tree creation and initialization using the Infovis Toolkit

2.2 Visualizations

Visualizations transform a set of semantic attributes stored in table columns into visual representations. They also perform filtering, zooming, navigation and picking. Each Visualization exposes a list of visual attributes that can be associated with columns. It then maintains an internal column of graphic shapes that are filtered before being rendered. Visualizations are redisplayed when at least one of the columns it refers to is modified. Furthermore, when a visual attribute used to compute the shapes is modified, the shapes are invalidated and recomputed for the next rendering. This mechanism unifies all the column changes, either due to a change in attribute values, in selection or during dynamic queries. No attempt is made to optimize dynamic queries at this level.

Standard visual attributes include color, size, label, transparency and sorting order. Selecting, filtering and sorting are also associated with columns in a similar way. Creating coordinated visualizations only requires several visualizations to refer to the same table and share their selection and filtering visual columns. For creating un-coordinated visualizations, different selection and filter columns should be associated with the visualizations.

Specific visualizations can add more visual attributes or add constraints to them. Scatter plots add x-axis and y-axis visual attributes.

Visualizations can be stacked. For example, node-link visualizations are composed of two layers: one for the nodes and one for the link underneath. Different visual attributes can then be assigned to links and to nodes. This mechanism is also used for Excentric Labels, implemented as a visualization layer on top of all the visualizations.

Visualizations use several sub-components to manage colors, permutations, redisplay, labeling and spatial deformation.

COLORS Mapping from abstract attributes to color is done through a *color visualization*: an interface that returns a color from a table row. Currently, we support four types of mapping for columns categorized as *sequential*, *categorical*, *differential* and *explicit*. The first three are described by Brewer [8] whereas the fourth simply means that the column directly contains a color specification. These categories can be explicitly stated in the column's "valueCategory" metadata or guessed from the column type and range. When a column is specified for the color visual attribute, its color visualization is returned by a *Color Visualization Factory*. Factories are used in several places in the InfoVis Toolkit. They are meant to be extended and modified by programmers; they allow a loose coupling between related components – like columns and their color visualizations. Globally changing color management in all the visualizations of the toolkit only requires the corresponding factory object to be modified.

PERMUTATIONS. Permutations are used both for sorting and deep filtering. They specify an order for table rows with the capability of filtering out a row by not specifying it in the order, hiding it from the visualization. Permutations also maintain the reverse mapping, from a row number to its index, and the count of visible rows. For tables, nothing more is required to manage permutations of rows. For trees, an updated view of the tree topology has to be maintained by the tree visualization with the children list sorted and filtered according to the permutation. For graph visualizations, we maintain permutations for the vertices and for the edges. Node-link diagram layouts are usually sensitive to the vertices and edges order. Matrix visualization requires two vertices permutations for the row and column order. The graph visualizations also need to maintain a modified graph topology with the vertices and edges sorted and filtered according to the vertices and edges permutations.

REDISPLAY. Redisplay is split between layout and rendering. Most of the time, a layout can be reused several times. Consider a user exploring a visualization: the first redisplay computes the layout. Then, the user explores the display, looking at labels through Tooltips or Excentric Labels. These dynamic labels require some picking to be computed, the picking reuses the computed layout. Selection only causes a redisplay without re-layout. In general, filtering only changes the set of redisplayed items, not their layout. This may seem odd for treemaps or graphs. We could recompute the layout each time an item is filtered, but that would usually change the display dramatically, making it hard or impossible to follow the changes from one frame to the next. Instead, just like in the Treemap4 program, we "grey out" filtered items interactively and offer a "remove filtered" option to erase them afterwards through the permutation. Only this last command requires a re-computation of the layout. Some dynamic filters do trigger a re-layout, in particular the filtering of the X or Y axis column of scatter plots.

The complexity of layout algorithms is linear for all table and tree visualizations (we are not aware of the need for more complex algorithms.) For graphs, only the matrix visualization is linear with the number of edges. All other graph layout algorithms are more complex and cannot be computed in interactive time for more than a few hundred items. This is also why we do not perform a layout when filtering a tree or a graph. When the user triggers a "hide filtered" or "hide selected" button, the items are hidden (removed from the permutation) and the layout is performed without them, which may take a couple of seconds for complex graphs.

RENDERING Visualizations maintain a column of shapes and repaint them when required. The rendering of items relies on shapes but also on color computation and optionally fisheye lenses. By default, the rendering iterates over each non-filtered rows in permutation order, computing the color with the color

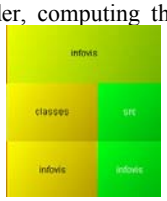

Figure 4: Smooth-shaded rendering or items.

The basic visualization allows for smooth shaded rendering where, instead of outlining items and drawing them with a flat color, items are shaded slightly so that they are distinguishable, even if they overlap (Figure 4.) However, using smooth shading (using GradientPaint objects) is very expensive in Java.

We experimented with native OpenGL graphics from Java, but gave up because it forced us to maintain two different implementations of each visualization to remain compatible with Java components. Instead, we have used Agile2D, an encapsulation of Graphics2D based on OpenGL, to get better rendering performance. Despite its merits, Agile2D support is still experimental because the current implementation of Java and Swing is not designed to support alternate Graphics2D implementations. This leads to performance issue due to the lack of software double-buffer support, forcing to redraw everything even for a slight change. However, the potentials are very promising, especially for visual attributes such as transparency or gradient that are very expensive in native Java, as described in [11].

LABELING Visualizations optionally support tool tips or dynamic labeling [13]. They use the visualization's picking mechanism to compute the labels under the pointer. Two methods are provided for picking: one returns the topmost item under a position and the second returns a list of items intersecting a rectangle.

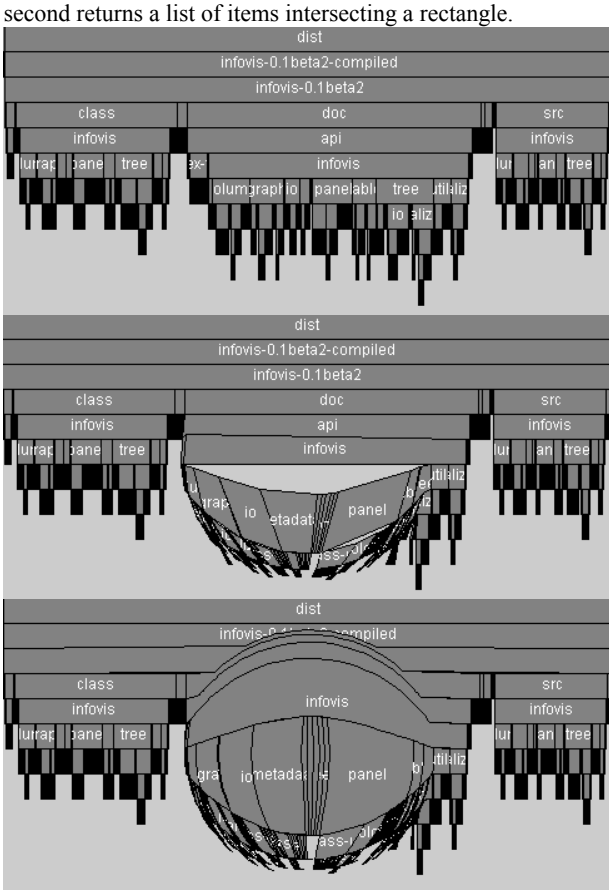

Figure 5: Subdivision of Shapes through a fisheye;
a) is the non-deformed visualization,
b) is the non-subdivided deformation,
c) is the correctly subdivided deformation.

SPATIAL DEFORMATIONS Spatial deformations can be applied by the rendering after the shapes have been computed by the layout. The toolkit currently supports a subset of Carpendale's [9] deformations within an extensible framework. We use a *Fisheye* object that transforms a Java shape into its deformation through the lens. Our implementation checks whether a specified shape intersects the lens and, if not, returns it without further processing. If it does, we iterate over its outline, applying the lens deformation to each of the control vertices. This method alone produces bad results even for simple shapes (Figure 5b.) Instead, we further subdivide the shape's outline into small segments (Figure 5c). First, we subdivide curved segments into small line segments using a flatness tolerance of 1/maxScale where maxScale is the maximum scale of the fisheye lens. This alone is not enough since long straight line – having a null flatness – need to be subdivided too. This subdivision is adaptive: on portions outside the lens and inside the focus, only the endpoints are transformed. In the compression area the lines are subdivided into segments of at most 1/3 the size of the compression region (Figure 5c.)

We also tried a regular grid-sampling on a view-aligned grid, with a default grid value of 4 pixels. This subdivision is not adaptive but behaves in a predictable manner, with worse performance than the adaptive algorithm, even with a small tolerance. We also provide interactive controls for users to choose the tolerance if they wish to trade speed for quality.

2.3 Dynamic Queries

Dynamic Queries are split into two parts: managing the filter column related to one or several visualizations and managing the Java/Swing component for the actual interaction. Filtering performance should allow for smooth interaction so performance is important. Dynamic queries are generally composed of primitive filtering expressions combined by an "and" conjunction [2]. To perform this operation as quickly as possible, dynamic queries rely on a column of bit sets. Each expression is allocated one bit. For each row, this bit is set when the expression returns true (the row is filtered.) When all the filters have been applied, only the rows with no bit set are displayed. When a dynamic filter is applied, only its bit is recomputed for all the rows so updating is always in time linear with the number of rows (if the filter time is constant, which is true for all our filters).

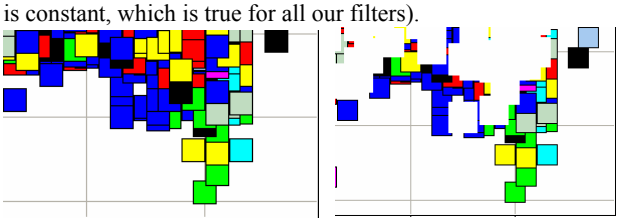

Figure 6: Optimizations of dynamic queries lead to incorrect display, all overlapping items being erased when some items are filtered out

Tanin et al. [25] describe two optimizations to dynamic queries that are implemented by several visualization systems (including Spotfire and Treemap4). First, they note that sliders are displayed using a specified number of pixels and no more slider positions can be perceived so they pre-compute, for each pixel position of the sliders, the set of items that are affected by the slider going though this pixel (it can increase or decrease depending on whether the slider goes one way or the other.) Using this technique, a second optimization is then applied: not all items are redisplayed when the slider moves. When new items are added, they are simply displayed on top of the others. When they are removed, the items are drawn using the background color. Nothing is done to show the items underneath in scatter plots, as seen in Figure 6. This is considered as acceptable since this rendering is only transient, during the dynamic query.

We do not perform these optimizations for three reasons: they require a complicated implementation with intricate inter-

dependencies between all the sliders to correctly compute the delta items; we do not want to deal with transient states during redisplay; finally, we want to provide sub-pixel precision when interacting with range-sliders. This is a very important issue when visualizing large data sets: if the precision of the sliders were related to their sizes, sliders would compete for screen real-estate with the visualization itself. Therefore, our range-sliders offer sub-pixel resolution: by moving the pointer away from the slider on the orthogonal direction, we increase the resolution and therefore the virtual length of the slider. There are many other ways described in the literature to avoid this pixel resolution problem [19] [1] and we felt using the optimization of Tanin et al. would limit the toolkit. The filtering speed is approximately 3,000,000 items per second. The limiting factor to achieve a 100ms interaction loop is therefore the rendering, which limits to 10,000 the maximal number of visible items for smooth interaction.

2.4 COMPONENTS

The information visualization literature describes a very large and rich set of interaction components, such as range sliders (or double edge sliders), alpha sliders and others visualization sliders. Moreover, visualizations can themselves be tailored into components for specific interaction tasks, blurring the limit between information visualization components and traditional interactive components or widgets. For example, a tree selection component in a toolkit is an interactive visualization using a specific representation and interaction. There is no reason why only one type of visualization should be provided. Similarly, a data slider is simply a slider with a visualization overlaid on top of it. Based on these observations, we designed the InfoVis toolkit visualizations so they can be used as components or within components.

In addition to the visualization components, the InfoVis toolkit provides several components to support interactive manipulations. By default, each visualization comes with a control panel organized in a tab group to interactively manipulate or configure the visualization (Figure 7.) The coupling between the visualizations and their panels is done through factories to allow programmers to substitute their own panels, components and interaction modes. Predefined components include range sliders and color visualization selectors. More components can be added as Swing components or InfoVis embedded components. Sliders or range sliders can then visualize interesting features such as text paragraph marks, code indentation depth and distribution of data. Tree selection components can be implemented using any tree visualization.

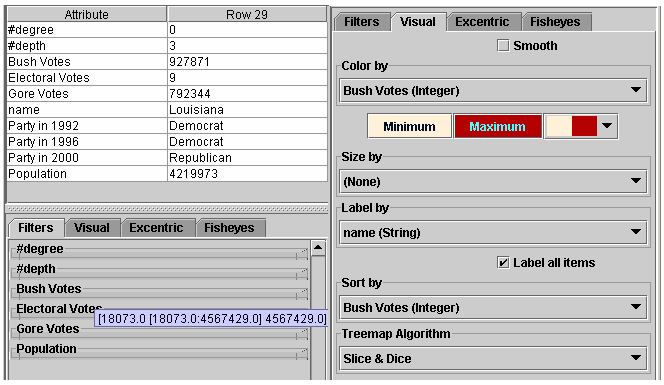

Figure 7: Control panels for treemap visualization

3. RELATED WORK

Implementing information visualization might look simple using a GUI toolkit: create a data structure for holding the data and use a component to render it on screen; then, add selection and dynamic queries. Toolkits such as Java Swing already have data structures for tables and trees as well as components to display and interact with them. However, these toolkits offer no support for dynamic queries, mapping of data attributes to visual attributes, dynamic labeling, spatial deformation, loading and saving from various formats, etc. Creating all these components from scratch is very long, tedious and frequently difficult.

Still, most InfoVis projects and products are created from scratch and several research centers have developed different applications for specific data structures and visualizations, all supporting a different subset of the useful components. Supporting all the components is long, difficult, and requires a global consistency hard to achieve when crafting a proof of concept.

Solutions exist to avoid starting from scratch. Among the toolkits related to Information Visualization, the most popular are PAD++ and Jazz [5] [6], GGobi [23], *XML Toolkit[7]*, Polaris [22] and GeoVista studio [24]. PAD++ and more recently Jazz are scene graph management toolkits designed to build zoomable user interfaces (ZUIs). They have been successfully used for creating InfoVis applications such as PhotoMesa [4] and SpaceTree [17]. These applications demonstrate new presentations and navigations, however, they do not offer the filtering and visual attribute management required to fully support information visualization techniques such as dynamic queries, dynamic labeling or spatial deformations.

GGobi and Polaris are specialized for visualization of tabular data structures. Polaris seems the closest system to InfoVis but is written in C++/OpenGL and organizes its in-memory database as tuples instead of columns. One important feature of Polaris is its ability to balance the rendering load among several visualizations to allow for real-time monitoring. This capability could be implemented in InfoVis but is not currently supported. Since Polaris is not available in source form, it is difficult to compare it in more details with InfoVis.

The XML Toolkit is a collection of information visualization algorithms rather than a full toolkit. It relies on the standard Java data structures interfaces such as TreeModel or TableModel which are not optimized in space or time but are well documented.

GeoVista studio is a large library of component based on the Java Beans protocol [10] to connect and configure the components using a visual programming interface. It can be considered as a high-level mechanism to choose and configure visualization components such as those provided by the InfoVis Toolkit. Indeed, it currently uses some of its interactive components such as the Excentric Labels.

Scientific visualization toolkits, such as the Visualization Toolkit [20] or IBM OpenDX [26], have a similar goal as the InfoVis Toolkit but for a different domain. They do not provide extended support for 2D visualizations, dynamic queries, generic data structures, labeling, space deformation etc.

Commercial information visualization applications, such as SpotFire [3] usually come with a development toolkit to customize them. However, the level of customization they provide is limited. For example, it does not allow replacing all the range sliders by another kind of component or adding Excentric Labeling [13]. Doing so is very important when designing novel information visualization components and requires deep access into the toolkit/application.

The InfoVis Toolkit has been inspired by several systems,

mainly Treemap4 (www.cs.umd.edu/hcil/treemap), SpaceTree [17] and MillionVis[14].

Assessing the quality of a toolkit is a difficult task. Shneiderman and Fekete [21] describe six criteria to qualify software tools for HCI. We list them here and apply them to the InfoVis Toolkit:

1. *Part of the application built using the tool:* data structures, presentation part and interaction part.
2. *Learning time:* long (weeks)
3. *Building time:* short (hours)
4. *Methodology imposed or advised:* create specific data structures first, then apply or create visualizations, then new interactions if required and finally specific control panels if needed.
5. *Communication with other subsystems:* integration of a rich and extendable set of input/output formats. Use of standard Java/Swing mechanisms for notifications (Listeners, Models and Events).
6. *Extensibility and Modularity:* very extensible but with design limits such as no 3D support for example.

The next section provides more concrete examples of extensions and applications of the InfoVis Toolkit.

4. EXAMPLES OF EXTENSIONS

The InfoVis Toolkit user is the application programmer. We describe five examples to let her/him assess the potentials of the toolkit:

1. implementing parallel coordinates
2. turning the standard tree layout into a radial tree layout
3. visualization of graphs as treemaps with links
4. implementation of the EdgeLens technique
5. visualizing an image repository as a treemap with thumbnails

All of the examples are in the InfoVis Toolkit distribution.

We asked undergraduate students to implement the Parallel Coordinates visualization using either the InfoVis Toolkit or a toolkit they freely chose. The InfoVis Toolkit implementation required 96 lines of code (Figure 8). Most student groups using InfoVis added interaction techniques to manipulate the axes because they felt doing only the visualization was not enough. Other groups have chosen different languages such as Tcl/Tk or Java without the InfoVis Toolkit. It took 600 to 6000 lines of code then to implement the visualization with fair results but much less functionalities in term of dynamic queries, input/output etc. It took one day of work to implement this visualization for an undergraduate student staring with the InfoVis Toolkit. His code is now distributed with the toolkit.

4.1 Radial Trees

Implementing radial trees (Figure 9) from standard (Cartesian) trees requires 37 lines of Java and took one day to an undergraduate student, most of this time being spent on recalling his trigonometric skills.

4.2 Visualization of Graphs as Treemaps with Links

In [12], we describe a technique for visualizing a graph as a treemap with overlaid links. To implement this technique using the InfoVis Toolkit, we had to overlay a set of links to a treemap. Since visualizations can be stacked, this is supported natively by the toolkit. A second aspect of our technique consists in avoiding arrows by using the bias of curvature for expressing the orientation of a link. We use a quadratic Bézier curve biased towards the starting point (Figure 10). The *LinkVisualization* class computes the link shapes using a *LinkShaper* object.

```
public class ParallelCoordinatesVisualization
  extends TimeSeriesVisualization {
public ParallelCoordinatesVisualization(Table table) {
  super(table);
}
public void paintBackground(Graphics2D graphics,
                            Rectangle2D bounds) {
  super.paintBackground(graphics, bounds);
  double sx = bounds.getWidth()/(columns.size()-1);
  graphics.setColor(Color.BLACK);
  for (int i = 0; i < columns.size(); i++) {
    int x = (int)(sx * i + bounds.getX());
    graphics.drawLine(x, (int) bounds.getY(),
                          x, (int) bounds.getHeight());
  }
}
public void computeShapes(Rectangle2D bounds) {
  double sx = bounds.getWidth()/(columns.size()-1);
  for (RowIterator iter = iterator(); iter.hasNext();) {
    int i = iter.nextRow();
    GeneralPath p = new GeneralPath();
    for (int col = 0; col < columns.size(); col++) {
        NumberColumn n = getNumberColumnAt(col);
        double min = n.getDoubleMin();
        double max = n.getDoubleMax();
        double diff = (max - min);
        double sy = bounds.getHeight() / diff;
        float x = (float) (sx * col + bounds.getX());
        float h = (float) (sy * (n.getDoubleAt(i) + min));
        float y =
            (float)(bounds.getY()+bounds.getHeight()-h);
        if (col == 0) {
          p.moveTo(x, y);
        } else {
          p.lineTo(x, y);
        }
    }
    setShapeAt(i, p);
  }
}
}
```

Figure 8: Implementation of Parallel Coordinates

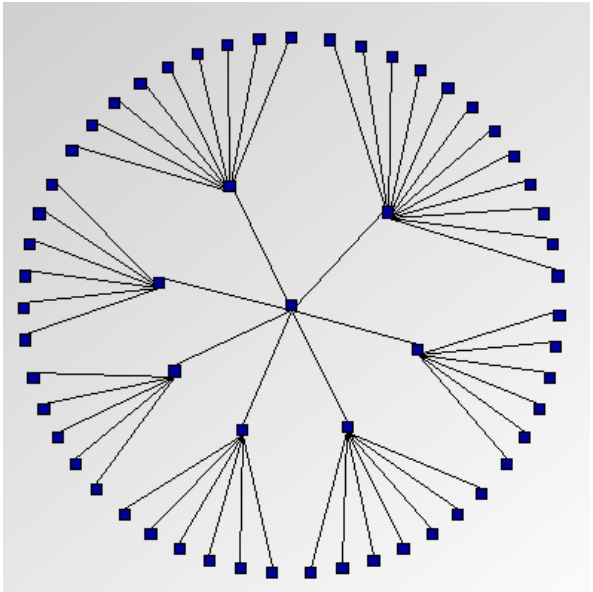

Figure 9: Radial tree visualization

Implementing the new subclass of *LinkShaper* takes about 50 lines. The reading of a Web site, extracting its tree structure and links rely on components already provided by the Toolkit such as the *HTMLGraphReader* so this part of code also requires around 50 lines. Finally, there are many possible choices in term of interaction. We have implemented three of them: static display of all of the links, dynamic display of the links starting or ending at

selected items and dynamic display of links starting or ending at the item under the pointer. The implementation of each of these interactions roughly requires one to ten lines of code.

Finally, all of these interactions modes can be useful so a control panel is added to let users configure the interaction style.

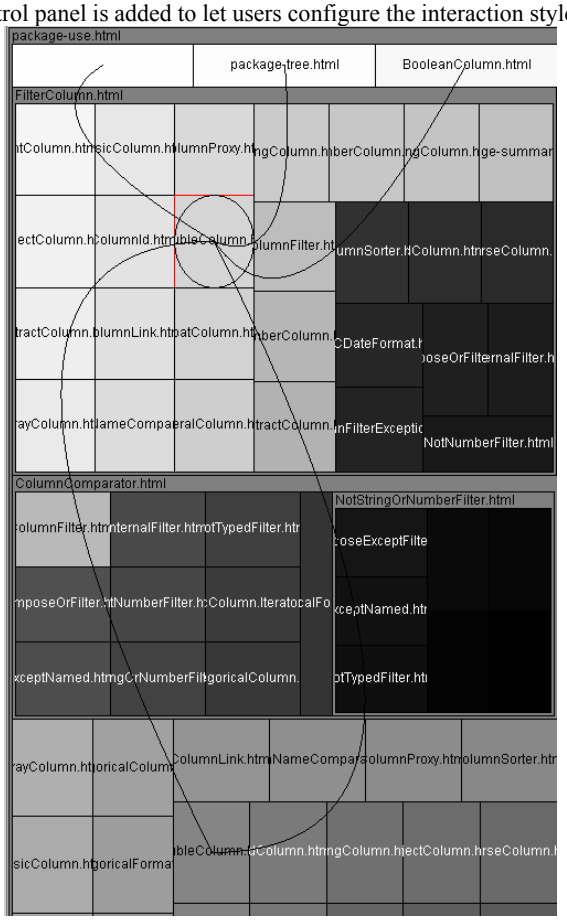

Figure 10: Visualization of the InfoVis Toolkit HTML manual as a treemap with overlaid links.

4.3 EdgeLens

In [27], Wong and Carpendale present a dynamic visualization technique to improve the readability of node-link representations by pushing edges away from the pointer. They use a modified version of Fisheyes transformations to deform links dynamically as the user moves his pointer around a graph.

We have implemented this technique in the InfoVis Toolkit by restricting the Fisheyes lens to only deform the link layer and not the other layers, as shown in Figure 11, requiring 1 line of code.

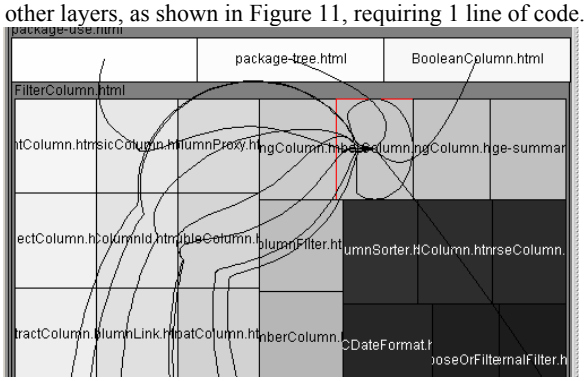

Figure 11: "EdgeLens" in InfoVis

The original article proposes two methods for bending the links: locally and globally. We only implemented the local bending because the global bending mechanism requires a modification of the standard Fisheyes technique that we haven't implemented but would be much simpler than the current implementation.

4.4 Image Thumbnails in Treemaps

We have created of subclass of the treemap visualization to show image thumbnails when representing a file-system hierarchy containing images (Figure 12). The source code is 200 lines long, mostly due to the computation and cacheing of image thumbnails (40 lines are for the visualization, 160 for the management of images).

This representation is a simplification of PhotoMesa [4] which adds more treemaps techniques such as "bubble maps" and "quantum visualization". They can be implemented as Treemap algorithms and added to the application when everything else works. This will probably be one of the exercises for next year's class on Information Visualization.

Figure 12: Visualization of a file-system hierarchy containing images in a Treemap

5. Conclusion and Future Work

This article described the InfoVis Toolkit, a toolkit that supports the development and extension of 2D Information Visualization components and applications using Java and Swing. Its key features are:

- Generic data structures suited to visualization
- Specific algorithms to visualize these data structures
- Mechanisms and components to perform direct manipulation on the visualizations
- Mechanisms and components to select, filter and perform well-known generic tasks of information visualization
- Components to perform labeling and spatial deformation.

InfoVis brings together several ideas from different domains and assembles them in a consistent framework, supporting the creation of new visualization techniques, thanks to optimized data structures and components to fit them together. It also supports the creation of new interaction components – such as new space deformation techniques or new sliders – that can easily replace existing ones for interacting on visualizations. It finally allows information visualization techniques to be easily integrated into any interactive application, bridging the gap between the information visualization community and the communities that need it.

The InfoVis Toolkit consists of approximately 30,000 lines of Java and a 300K Jar file. It is currently licensed under the QPL and available at: http://www.lri.fr/~fekete/InfovisToolkit. It is used by several research projects in domains including biology, cartography and trace analysis.

A major concern with the InfoVis toolkit is offering performance

without losing flexibility and modularity. We will improve the Agile2D system to offer new abstractions while keeping with the Java2D compatibility as much as possible. We also hope Sun will allow better integration for non-native implementations of Graphics2D. By relying more on OpenGL, we expect to offer richer visual attributes to visualizations, including management of the third dimension with its related capabilities such as lighting, fog, depth clipping and stereovision to name a few. These capabilities do not require any 3D navigation to be usable.

In the near future, we also plan to implement mechanisms to support animation and continuous monitoring for time-oriented visualizations.

We look forward to continuing the development of the InfoVis Toolkit and expect the Information visualization community will provide visualization components and useful feedback.

REFERENCES

[1] C. Ahlberg and B. Shneiderman, The Alphaslider: A Compact and Rapid Selector. in *Proceedings of CHI '94*, (Boston, MA, 1994), ACM Press, 365-371.

[2] C. Ahlberg and B. Shneiderman, Visual Information Seeking: Tight Coupling of Dynamic Query Filters with Starfield Displays. in *Proceedings of CHI '94*, (Boston, MA, 1994), ACM Press, 313-317.

[3] C. Ahlberg and E. Wistrand, IVEE: An Information Visualization & Exploration Environment. in *Proceedings of the IEEE Symposium on Information Visualization '95*, (1995), IEEE Press, 66-73.

[4] B.B. Bederson, PhotoMesa: A Zoomable Image Browser Using Quantum Treemaps and Bubblemaps. in *Proceedings of the 14th annual ACM symposium on User interface software and technology*, (Orlando, Florida, 2001), ACM Press, 71 - 80.

[5] B.B. Bederson, J. Hollan, K. Perlin, J. Meyer, D. Bacon and G. Furnas Pad++: A Zoomable Graphical Sketchpad for Exploring Alternate Interface Physics. *Journal of Visual Languages and Computing*, *7*. 3-31.

[6] B.B. Bederson, J. Meyer and L. Good, Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java. in *Proceedings of User Interface and Software Technology (UIST 2000)*, (San Diego, CA, 2000), ACM Press, 171-180.

[7] K. Börner and Y. Zhou, A Software Repository for Education and Research in Information Visualization. in *Information Visualisation Conference*, (London, England, 2001), 257-262.

[8] C.A. Brewer, Guidelines for Use of the Perceptual Dimensions of Color for Mapping and Visualization. in *Proceedings of the International Society for Optical Engineering (SPIE)*, (San José, CA, 1994), 54-63.

[9] M.S.T. Carpendale and C. Montagnese, A framework for unifying presentation space. in *Proceedings of the 14th annual ACM symposium on User interface software and technology*, (Orlando, Florida, 2001), ACM Press, 61-70.

[10] R. Englander *Developing Java Beans*. O'Reilly & Associates, 1997.

[11] J.-D. Fekete. The Infovis Toolkit *INRIA Futurs Research Report*, INRIA Futurs, Orsay, 2003, 15.

[12] J.-D. Fekete, N. Dang, C. Plaisant and D. Wang, Interactive Poster: Overlaying Graph Links on Treemaps. in *IEEE Symposium on Information Visualization*, (Seattle, WA, 2003).

[13] J.-D. Fekete and C. Plaisant, Excentric Labeling: Dynamic Neighborhood Labeling for Data Visualization. in *Proceedings of ACM CHI 99 Conference on Human Factors in Computing Systems*, (1999), 512-519.

[14] J.-D. Fekete and C. Plaisant, Interactive Information Visualization of a Million Items. in *IEEE Symposium on Information Visualization (InfoVis'02)*, (Boston, MA, 2002), IEEE Press, 117-124.

[15] B.R. Gaines Modeling and forecasting the information sciences. *Information Sciences: an International Journal, Special issue on information sciences—past, present, and future*, *57-58*. 3 - 22.

[16] M. Ghoniem, N. Jussien and J.-D. Fekete, VISEXP: visualizing constraint solver dynamics using explanations. in *FLAIRS'04: Seventeenth international Florida Artificial Intelligence Research Society conference*, (Miami Beach, FL, 2004), AAAI press.

[17] J. Grosjean, C. Plaisant and B.B. Bederson, SpaceTree: Supporting Exploration in Large Node Link Tree, Design Evolution and Empirical Evaluation. in *IEEE Symposium on Information Visualization (InfoVis'02)*, (Boston, MA, 2002), IEEE Press, 57 -64.

[18] D.E. Knuth Fundamental Algorithms. *1*.

[19] S. Pook, E. Lecolinet, G. Vayssex and E. Barillot, Control Menu: Execution and Control in a Single Interactor. in *In Extended Abstracts of CHI2000*, (Den Hague, The Netherlands, 2000), ACM Press, 263- 264.

[20] W. Schroeder, K. Martin and B. Lorensen *The Visualization Toolkit: An Object Oriented Approach to 3D Graphics*. Kitware, Inc., 2003.

[21] B. Shneiderman and C. Plaisant *Designing the User Interface*. Addison-Wesley Publisher, 2004.

[22] C. Stolte, 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*, *8* (1). 52-65.

[23] J. Symanzik, D.F. Swayne, D.T. Lang and D. Cook, Software Integration for Multivariate Exploratory Spatial Data Analysis. in *Proceedings of the SCISS Specialist Meeting "New Tools for Spatial Data Analysis"*, (Santa Barbara, CA, 2002).

[24] M. Takatsuka and M. Gahegan GeoVISTA Studio: A Codeless Visual Programming Environment For Geoscientific Data Analysis and Visualization. *The Journal of Computers & Geosciences*, *28* (10). 1131-1144.

[25] E. Tanin, R. Beigel and B. Shneiderman Incremental data Structures and Algorithms for Dynamic Query Interfaces. *SIGMOD Record*, *25* (4). 21-24.

[26] D. Thompson, J. Braun and R. Ford *OpenDX: Paths to Visualization*. VIS, Inc., 2000.

[27] N. Wong, S. Carpendale and S. Greenberg, EdgeLens: An Interactive Method for Managing Edge Congestion in Graphs. in *2003 IEEE Symposium on Information Visualization*, (Seattle, WA, 2003), IEEE Press, 52-58.