Prefuse Tutorial

Prefuse is a framework for building interactive information visualization applications using Java. It provides flexible data structures for:

- importing and storing the data
- mapping the data to visual elements
- adding direct manipulation interaction

The entire Prefuse toolkit is written in Java 1.4, using the Java 2D graphics libraries. It is licensed under a BSD license, and can be freely used for both commercial and non-commercial purposes.

Prefuse was named after the band *Prefuse 73*. It is pronounced to rhyme with **refuse**.

1. Download and Install Prefuse

Prefuse can be downloaded from http://prefuse.org/. The structure of the toolkit is as follows:

- build: compiled classes and jars (once generated)
- data: demo data
- demos: demo applications
- doc: JavaDoc files (once generated)
- lib: third-party libraries
- src: source code for Prefuse
- test: JUnit tests

Importing Prefuse into a development environment such as **Eclipse** or **NetBeans** will greatly simplify your ability to generate code using the toolkit. In this tutorial, we will assume you are using NetBeans version 6.0. If you prefer Eclipse, follow the instructions provided in the readme.txt file that comes with Prefuse.

Since the project was developed in Eclipse, you'll need to download the NetBeans plugin that can be used to import Eclipse projects. This can be done under Tools -> Plugins, and then navigating the available plugins for the `Eclipse Project Importer`.

After downloading the Prefuse project, you should place the directory in your Java source directory. We'll assume this directory is ~/Java/. The next step is to do the import into NetBeans. Select File -> Import Project -> Eclipse Project (note that this option is not available until you download the Eclipse Project Importer). Since the project isn't in an existing Eclipse workspace, select the second option (Import Project Ignoring Project Dependancies). Navigate to ~/Java/prefuse for both the project to import and the destination folder (the importer doesn't move the source files, so we had to put them in the location we wanted to begin with).

There are two problems with this import method: the Netbeans project information and the source files are in different directories, and there is a missing Library file (Junit). To solve the first problem, close the project, navigate to ~/Java/prefuse (or whichever directory you chose to use), and move the contents of ~/Java/prefuse/prefuse back one directory level (this will overwrite the build.xml file, and place a nbproject directory in ~/Java/prefuse). To solve the second project, either download and import junit.jar into the project, or delete the code in the Test package.

In order to use the Prefuse toolkit in other projects, you will need to build the `prefuse.jar` file. To do so, right-click on the Prefuse project, and select Clean and Build. If you want to use the Prefuse library in other projects, you can either add a link to the project (right-click on Libraries and select Add Project), or add a link to the jar file you just created (right-click on Libraries and select Add JAR/Folder).

From this point on, we'll assume that you have created a separate project for this course, and added the appropriate link to the Prefuse project or jar file.

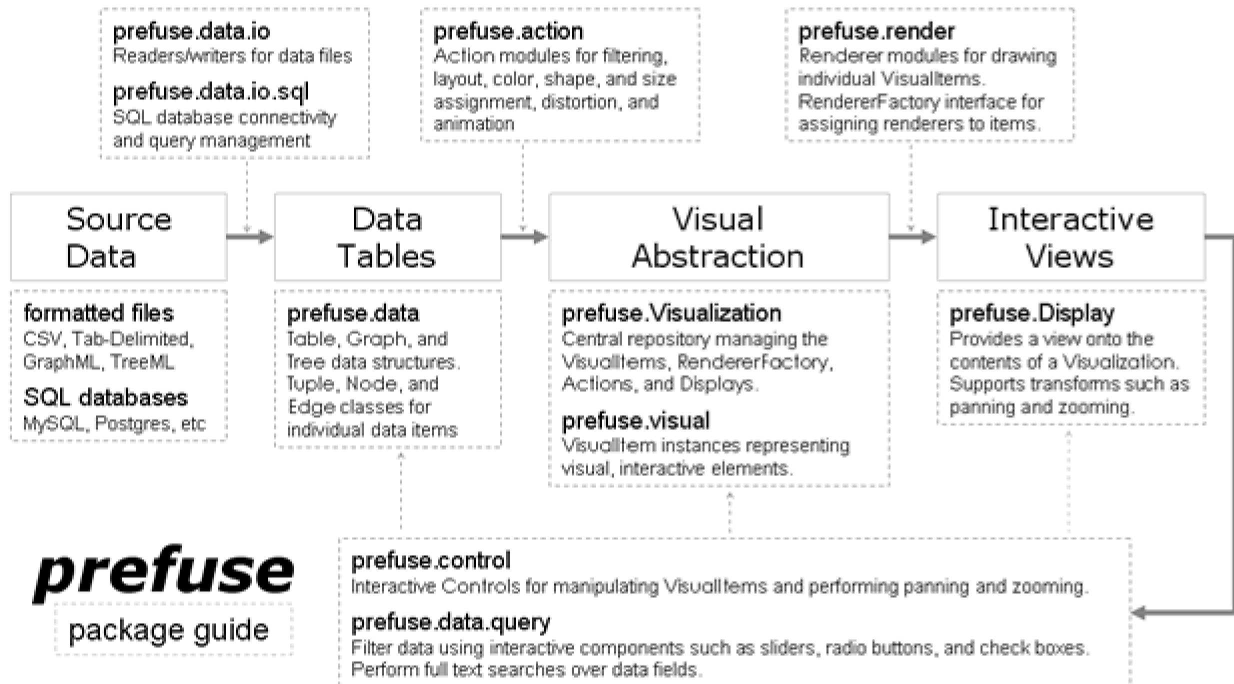
2. Prefuse Basics

In general, when using Prefuse, there are four things that need to be handled in your Java program:

1. importing your source data into the **internal Prefuse data structures**
   - there are a number of methods that greatly simplify this step
   - data can be imported from simple data files, XML files, over the Web, or directly from databases
2. map the data to a **visual abstraction**
   - includes features such as the spatial layout, color, size, and shape for the data elements
   - each of these can be set programmatically based on the data values
3. generate a **view** of the visual abstraction
   - the view may be zoomed to show only a subset of the data in the visual abstraction
   - multiple views may be generated to show the same visual abstraction from different perspectives
4. add **user interaction** features that can affect the elements from the previous steps
   - zooming in affects the view
   - selecting an object may alter the visual abstraction
   - filtering features may affect the internal Prefuse data structures

The process here is similar to the **model-view-controller** design pattern. The model is represented by the *internal Prefuse data structures*. The view is supported by the *visual abstraction* and the corresponding *views*. The controller is supported by the *user interaction features*.

The Prefuse documentation includes an excellent diagram illustrating various Prefuse packages that support generating an information visualization application:



3. Loading Data

Prefuse includes a number of support classes to make the loading of data into the internal representations easier. `prefuse.data` contains the classes to efficiently store the data as a `Table`, `Graph`, or `Tree` data structure. Table rows are stored in the `Tupple` class. Graphs and trees make use of the `Node` and `Edge` classes.

There is a built-in expression language and interpreter for querying the data, or creating derived data as functions of existing data. The details of this expression language are contained in the `prefuse.data.expression` package. This expression language can be useful for manipulating the data in order to support the creation of an effective visual representation.

The `prefuse.data.io` package contains classes for reading and writing table, graph, and tree data. For tabular data, CSV (comma-separated values) and other delimited text files are supported. For graph and tree data, the XML-based GraphML and TreeML formats are supported.

Data stored in a database such as **MySQL** can be queried and loaded into the internal `Table` data structures using the classes contained in `prefuse.data.io.sql`. The structured tables returned from a database query can also be used as the node and edge tables for a `Graph` or `Tree` data structure.

4. Visual Abstraction

The key class in Prefuse for generating a visual representation is the `Visualization` class in the `prefuse` package. A visual abstraction of the data loaded into the internal data structures in the previous step (`Table`, `Graph`, or `Tree`) is created by adding the data to the `Visualization` object. For example:

```
Table   t = new CSVTableReader().readTable("data/population.csv");
Visualization vis = new Visualization();
VisualTable vt = vis.addTable("myTable", t);
```

In this sample code, `t` contains the original data loaded into the internal Prefuse data structures, `vis` is the object that controls the visualization (i.e, the controller in the MVC framework), and `vt` is a reference to the visual abstraction of the table `t`. In this visual abstract of the original data, the original data is present, along with visualization-specific data such as x,y cooridinates, colour, size, and font values.

New derived data values can be generated and added to the visual abstractions using, for example, the expression language previously mentioned:

```
vt.addColumn("label", "CONCAT([Geographic name], ' (Population: ',
FORMAT([2006 Population],0), ')')");
```

The specification of the visual mappings of data in the visual abstraction are provided by the **Action** modules. There are a collection of classes in the `prefuse.action` package and its sub-packages to handle layouts, visual encodings, distortions, and animations. It is possible for a programmer to create new classes that extend the capabilities of these classes, resulting in new methods for visualizing the data.

In the `prefuse.action.assigment` package are the classes that deal with assigning actions to the `Visualization` object which handle the visual encoding of the data. These classes can be divided into two categories:

- those that specify how to render the item based on the data
  - `DataColorAction`
  - `DataShapeAction`
  - `DataSizeAction`
- and those that specify how to render the item in a static manner
  - `ColorAction`
  - `ShapeAction`
  - `SizeAction`
  - `FontAction`
  - `StrokeAction`

For example, the shape of an object might be the same, regardless of the type of object. However, the size of the object might be based on some attribute of the data:

```
ShapeAction shape = new ShapeAction("canUrban", Constants.SHAPE_RECTANGLE);
```

```
DataSizeAction size = new DataSizeAction("canUrban", "population ordinal");
```

Collections of actions are usually grouped together into an `ActionList` object, which is then added to the `Visualization` object via the `putAction` method. This method takes both the `ActionList` and a name assigned to the collection of actions. This allows the actions to programatically be started and stopped, as needed. To enable the actions, the `run` method of the `Visualization` object is executed, using the name given to the collection of actions.

The final say of how an item is rendered in Prefuse depends upon the `Renderer` for the particular item. A `RendererFactory` allows different visual elements to rendered by different `Renderer` classes, based on properties of the item. For example, the axes of a scatterplot would need to be rendered differently than the data items. The classes that support the rendering operations are in the `prefuse.render` package. An example of how to create a `RenderFactory` class and assign it to a `Visualization` object is provided below.

```
vis.setRendererFactory(new RendererFactory() {

    AbstractShapeRenderer sr = new ShapeRenderer();
    Renderer arY = new AxisRenderer(Constants.RIGHT, Constants.TOP);
    Renderer arX = new AxisRenderer(Constants.CENTER, Constants.FAR_BOTTOM);

    public Renderer getRenderer(VisualItem item) {
        return item.isInGroup("ylab") ? arY : item.isInGroup("xlab") ? arX :
sr;
    }
});
```

5. Views and User Interaction

Once a visual abstraction of the data is loaded into the `Visualization`, the actions for handling the layout and encoding of the data are specified, and the program is instructed how to render the items, the last step is to specify a **view** of the `Visualization`. Interactive views are controlled by the `Display` object (part of the `prefuse` package). You can think of the `Display` object as providing a "window" into the contents of the `Visualization` object. This object draws all the items within its current view, and can support panning, zooming and rotation.

A single `Visualization` object can support multiple `Display` objects simultaneously, allowing multiple views into the same data set. For example, a program can be written with one view showing an overview of the data, and a second with a detailed view into specific elements. Modifications to the `Display` objects do not affect one another; however, modifications of the `Visualization` object, or the visual abstractions would be visible in both displays. As such, if the user selects a particular object in one display, the other display will show that object as being selected as well.

A new `Display` for an existing `Visualization` can be generated easily. The parameters for the display can then be set.

```
Display myDisplay = new Display(vis);
myDisplay.setSize(700, 450);
myDisplay.setHighQuality(true);
```

Interactive controls of the display are specified using the `Control` objects found in `prefuse.controls`. These controls might be to handle tool tips (`ToolTipControl`), hovering the mouse (`HoverActionControl`), or the focusing of items the user clicks on (`FocusControl`). Other controls that don't affect the data include the `DragControl`, `PanControl`, and the `ZoomControl`.

Custom controls that affect other objects in the program can be created by creating a new `ControlAdapter` and overriding the `itemEntered` and `itemExited` methods. This overriding of the methods can occur in-line, or you can create a new class that inherits from `ControlAdapter`, and then overrides the methods (this is the preferred method).

Once a control is created, it must be added to the `Display` using the `addControlListener` method:

```
ToolTipControl ttc = new ToolTipControl("label");
myDisplay.addControlListener(ttc);
```

Dynamic querying and filtering is supported through the classes in `prefuse.data.query`. For example, text searching is supported by the `SearchQueryBinding` class. Predicates such as the `AndPredicate` in `prefuse.data.expression.AndPredicate` do the work of executing a filter of the data based on the query objects. Of course, these query objects also need to be linked to other interface elements to get their input from the users.

```
SearchQueryBinding searchQ = new SearchQueryBinding(vt, "Geographic name");
AndPredicate filter = new AndPredicate(searchQ.getPredicate());
JSearchPanel searcher = searchQ.createSearchPanel();
searcher.setLabelText("Urban Centre: ");
```

Note that while it doesn't look like there is much going on in these four lines of code, there really is. When a user enters something in to `searcher`, a change is fired in `searchQ`, which causes a change in `filter`. Since `searchQ` is also linked to `vt` (the visual abstraction of the table data), the filtering actually occurs on this object. As a result, the table of data is filtered based on text matching of what the user entered and the contents of the "Geographic name" field of the table.

6. Example 1: Tabular Layout

The following Java program loads in a data set from the `data/population.csv` file, and provides a scatterplot visual representation of the data. Note that this is the same data set as provided with Assignment 1.

[population.csv]

[TabularDataVis.java]

```
package prefuse_tutorial;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Insets;
import java.awt.event.ComponentAdapter;
import java.awt.event.ComponentEvent;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.geom.Rectangle2D;
import java.text.NumberFormat;
import java.util.Iterator;

import javax.swing.BorderFactory;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SwingConstants;

import prefuse.Constants;
import prefuse.Display;
import prefuse.Visualization;
import prefuse.action.ActionList;
import prefuse.action.GroupAction;
import prefuse.action.RepaintAction;
import prefuse.action.assignment.ColorAction;
import prefuse.action.assignment.DataColorAction;
import prefuse.action.assignment.DataSizeAction;
import prefuse.action.assignment.ShapeAction;
import prefuse.action.filter.VisibilityFilter;
import prefuse.action.layout.AxisLabelLayout;
import prefuse.action.layout.AxisLayout;
import prefuse.controls.Control;
import prefuse.controls.ControlAdapter;
import prefuse.controls.ToolTipControl;
import prefuse.data.Table;
import prefuse.data.expression.AndPredicate;
import prefuse.data.io.CSVTableReader;
import prefuse.data.query.RangeQueryBinding;
import prefuse.data.query.SearchQueryBinding;
import prefuse.render.AbstractShapeRenderer;
import prefuse.render.AxisRenderer;
import prefuse.render.Renderer;
import prefuse.render.RendererFactory;
import prefuse.render.ShapeRenderer;
import prefuse.util.ColorLib;
import prefuse.util.FontLib;
import prefuse.util.UpdateListener;
import prefuse.util.ui.JFastLabel;
import prefuse.util.ui.JRangeSlider;
import prefuse.util.ui.JSearchPanel;
import prefuse.util.ui.UILib;
import prefuse.visual.VisualItem;
import prefuse.visual.VisualTable;
```

```
import prefuse.visual.expression.VisiblePredicate;

public class TabularDataVis extends JPanel {

    /*
     * main execution class (for running as an applet)
     */
    public static void main(String[] args) {

        JFrame f = buildFrame();
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setVisible(true);
    }

    /*
     * load the data and generate the frame that contains the visualization
     */
    public static JFrame buildFrame() {
        // load the data
        Table t = null;
        try {
            // data in CSV format, so use CSVTableReader
            t = new CSVTableReader().readTable("data/population.csv");
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }

        // set the title on the frame
        JFrame frame = new JFrame("Canadian Urban Population");

        // add in the visualization contents (calls the constructor for this
class)
        frame.setContentPane(new TabularDataVis(t));

        // pack the elements in the frame and return
        frame.pack();
        return frame;
    }

    /*
     * Global variables and configuration parameters
     */

    // summary information needs to be set and reloaded in support classes
    private String g_totalStr;
    private JFastLabel g_total = new JFastLabel("");
    // access to the visualization elements are needed within the support
classes
    private Visualization g_vis;
    private Display g_display;
    // containers for the data, x-axis labels, and y-axis labels
    private Rectangle2D g_dataB = new Rectangle2D.Double();
    private Rectangle2D g_xlabB = new Rectangle2D.Double();
    private Rectangle2D g_ylabB = new Rectangle2D.Double();
```

```
    /*
     * Constructor for the class
     * This is where all the important stuff happens
     */
    public TabularDataVis(Table t) {
        super(new BorderLayout());

        /*
         * Step 1: Setup the Visualization
         */

        // create a new visualization object, and assign it to the global
variable
        final Visualization vis = new Visualization();
        g_vis = vis;

        // create a visual abstraction of the table data (loaded in the
buildFrame method)
        // call our data "canUrban"
        VisualTable vt = vis.addTable("canUrban", t);

        // add a new column containing a label string showing
        // the Geographic name and population
        // note: uses the prefuse expression language
        vt.addColumn("label", "CONCAT([Geographic name], ' (Population: ',
FORMAT([2006 Population],0), ')')");

        // add a new column that divides the provinces by their geographic
location (derived values)
        // note: uses the prefuse expression language
        vt.addColumn("geographic location", "IF ([Province]='BC') THEN 1 ELSE
" +
                "(IF ([Province] = 'AB' OR [Province] = 'SK' OR [Province] =
'MB') THEN 2 ELSE " +
                "(IF ([Province] = 'ON' OR [Province] = 'QC') THEN 3 ELSE" +
                "(IF ([Province] = 'NS' OR [Province] = 'NB' OR [Province] =
'PE' OR [Province] = 'NL') THEN 4 ELSE 5)))");

        // add a new column that converts the population data to ordinal data
(derived values)
        // note: uses the prefuse expression language
        vt.addColumn("population ordinal", "IF ([2006 Population] > 5000000)
THEN 7 ELSE " +
                "(IF ([2006 Population] > 1000000) THEN 6 ELSE " +
                "(IF ([2006 Population] > 250000) THEN 5 ELSE " +
                "(IF ([2006 Population] > 100000) THEN 4 ELSE " +
                "(IF ([2006 Population] > 50000) THEN 3 ELSE " +
                "(IF ([2006 Population] > 20000) THEN 2 ELSE 1)))))");

        // create a new renderer factory for drawing the visual items
        vis.setRendererFactory(new RendererFactory() {

            // specify the default shape renderer (the actions will decide
how to actually render the visual elements)
            AbstractShapeRenderer sr = new ShapeRenderer();
            // renderers for the axes
```

```
            Renderer arY = new AxisRenderer(Constants.RIGHT, Constants.TOP);
            Renderer arX = new AxisRenderer(Constants.CENTER,
Constants.FAR_BOTTOM);

            // return the appropriate renderer for a given visual item
            public Renderer getRenderer(VisualItem item) {
                return item.isInGroup("ylab") ? arY : item.isInGroup("xlab")
? arX : sr;
            }
        });

        /*
         * Step 2: Add X-Axis
         */

        // add the x-axis
        AxisLayout xaxis = new AxisLayout("canUrban", "Province",
Constants.X_AXIS, VisiblePredicate.TRUE);

        // ensure the axis spans the width of the data container
        xaxis.setLayoutBounds(g_dataB);

        // add the labels to the x-axis
        AxisLabelLayout xlabels = new AxisLabelLayout("xlab", xaxis, g_xlabB,
15);

        /*
         * Step 3: Add the Y-Axis and its dynamic query feature
         */

        // dynamic query based on population data
        RangeQueryBinding populationQ = new RangeQueryBinding(vt, "2006
Population");
        AndPredicate filter = new AndPredicate(populationQ.getPredicate());

        // add the y-axis
        AxisLayout yaxis = new AxisLayout("canUrban", "2006 Population",
Constants.Y_AXIS, VisiblePredicate.TRUE);

        // set the range controls on the y-axis
        yaxis.setRangeModel(populationQ.getModel());
        populationQ.getNumberModel().setValueRange(0, 6000000, 0, 6000000);

        // ensure the y-axis spans the height of the data container
        yaxis.setLayoutBounds(g_dataB);

        // add the labels to the y-axis
        AxisLabelLayout ylabels = new AxisLabelLayout("ylab", yaxis,
g_ylabB);
        NumberFormat nf = NumberFormat.getIntegerInstance();
        nf.setMaximumFractionDigits(0);
        ylabels.setNumberFormat(nf);

        /*
         * Step 4: Add the search box
         */
```

```
        // dynamic query based on Geographic name data
        SearchQueryBinding searchQ = new SearchQueryBinding(vt, "Geographic
name");
        filter.add(searchQ.getPredicate());          // reuse the same filter
as the population query

        /*
         * Step 5: Colours and Shapes
         */

        // assign a set of five perceptually distinct colours to assign to
the provinces
        // chosen from ColorBrewer (5-class qualitative Set1)
        int[] palette = new int[]{
            ColorLib.rgb(77, 175, 74),
            ColorLib.rgb(55, 126, 184),
            ColorLib.rgb(228, 26, 28),
            ColorLib.rgb(152, 78, 163),
            ColorLib.rgb(255, 127, 0)
        };

        // specify the stroke (exterior line) based on the ordinal data
        DataColorAction color = new DataColorAction("canUrban", "geographic
location",
                Constants.ORDINAL, VisualItem.STROKECOLOR, palette);

        // specify the fill (interior) as a static colour (white)
        ColorAction fill = new ColorAction("canUrban", VisualItem.FILLCOLOR,
0);

        // represent all the data points with rectangles
        ShapeAction shape = new ShapeAction("canUrban",
Constants.SHAPE_RECTANGLE);

        // assign the size of the visual element based on the population data
(we
        // converted the 2006 Population data to ordinal values)
        DataSizeAction size = new DataSizeAction("canUrban", "population
ordinal");

        // setup a counter to keep track of which data points are currently
being viewed
        Counter cntr = new Counter("canUrban");

        /*
         * Step 6: Create the action list for drawing the visual elements
         */

        ActionList draw = new ActionList();
        draw.add(cntr);
        draw.add(color);
        draw.add(fill);
        draw.add(shape);
        draw.add(size);
        draw.add(xaxis);
```

```
        draw.add(yaxis);
        draw.add(ylabels);
        draw.add(new RepaintAction());
        vis.putAction("draw", draw);
        vis.putAction("xlabels", xlabels);

        /*
         * create the action list for updating the visual elements
         * (during interactive operations and re-sizing of the window)
         */
        ActionList update = new ActionList();
        update.add(new VisibilityFilter("canUrban", filter));// filter
performs the size/name filtering
        update.add(cntr);
        update.add(xaxis);
        update.add(yaxis);
        update.add(ylabels);
        update.add(new RepaintAction());
        vis.putAction("update", update);

        // create an update listener that will update the visualization when
fired
        UpdateListener lstnr = new UpdateListener() {

            public void update(Object src) {
                vis.run("update");
            }
        };

        // add this update listener to the filter, so that when the filter
changes (i.e.,
        // the user adjusts the axis parameters, or enters a name for
filtering), the
        // visualization is updated
        filter.addExpressionListener(lstnr);

        /*
         * Step 7: Setup the Display and the other Interface components
         * (scroll bar, query box, tool tips)
         */

        // create the display
        g_display = new Display(vis);

        // set the display properties
        g_display.setBorder(BorderFactory.createEmptyBorder(10, 20, 10, 20));
        g_display.setSize(700, 450);
        g_display.setHighQuality(true);

        // call the function that sets the sizes of the containers that
contain
        // the data and the axes
        displayLayout();

        // whenever the window is re-sized, update the layout of the axes
        g_display.addComponentListener(new ComponentAdapter() {
```

```
    public void componentResized(ComponentEvent e) {
        displayLayout();
    }
});

// title label (top left)
JFastLabel g_details = new JFastLabel("Canadian Urban Population");
g_details.setPreferredSize(new Dimension(350, 20));
g_details.setVerticalAlignment(SwingConstants.BOTTOM);

// total label (top right)
g_total.setPreferredSize(new Dimension(350, 20));
g_total.setHorizontalAlignment(SwingConstants.RIGHT);
g_total.setVerticalAlignment(SwingConstants.BOTTOM);

// tool tips
ToolTipControl ttc = new ToolTipControl("label");
Control hoverc = new ControlAdapter() {

    public void itemEntered(VisualItem item, MouseEvent evt) {
        if (item.isInGroup("canUrban")) {
            g_total.setText(item.getString("label"));
            item.setFillColor(item.getStrokeColor());
            item.setStrokeColor(ColorLib.rgb(0, 0, 0));
            item.getVisualization().repaint();
        }
    }

    public void itemExited(VisualItem item, MouseEvent evt) {
        if (item.isInGroup("canUrban")) {
            g_total.setText(g_totalStr);
            item.setFillColor(item.getEndFillColor());
            item.setStrokeColor(item.getEndStrokeColor());
            item.getVisualization().repaint();
        }
    }
};
g_display.addControlListener(ttc);
g_display.addControlListener(hoverc);

// vertical slider for adjusting the population filter
JRangeSlider slider = populationQ.createVerticalRangeSlider();
slider.setThumbColor(null);
slider.setToolTipText("drag the arrows to filter the data");
// smallest window: 200,000
slider.setMinExtent(200000);
slider.addMouseListener(new MouseAdapter() {

    public void mousePressed(MouseEvent e) {
        g_display.setHighQuality(false);
    }

    public void mouseReleased(MouseEvent e) {
        g_display.setHighQuality(true);
        g_display.repaint();
```

```
        }
    });

    // search box
    JSearchPanel searcher = searchQ.createSearchPanel();
    searcher.setLabelText("Urban Centre: ");
    searcher.setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 0));

    /*
     * Step 8: Create Containers for the Interface Elements
     */

    // add the listener to this component
    this.addComponentListener(lstnr);

    // container for elements at the top of the screen
    Box topContainer = new Box(BoxLayout.X_AXIS);
    topContainer.add(Box.createHorizontalStrut(5));
    topContainer.add(g_details);
    topContainer.add(Box.createHorizontalGlue());
    topContainer.add(Box.createHorizontalStrut(5));
    topContainer.add(g_total);
    topContainer.add(Box.createHorizontalStrut(5));

    // container for elements at the bottom of the screen
    Box bottomContainer = new Box(BoxLayout.X_AXIS);
    bottomContainer.add(Box.createHorizontalStrut(5));
    bottomContainer.add(searcher);
    bottomContainer.add(Box.createHorizontalGlue());
    bottomContainer.add(Box.createHorizontalStrut(5));
    bottomContainer.add(Box.createHorizontalStrut(16));

    // fonts, colours, etc.
    UILib.setColor(this, ColorLib.getColor(255, 255, 255), Color.GRAY);
    slider.setForeground(Color.LIGHT_GRAY);
    UILib.setFont(bottomContainer, FontLib.getFont("Tahoma", 15));
    g_details.setFont(FontLib.getFont("Tahoma", 18));
    g_total.setFont(FontLib.getFont("Tahoma", 16));

    // add the containers to the JPanel
    add(topContainer, BorderLayout.NORTH);
    add(g_display, BorderLayout.CENTER);
    add(slider, BorderLayout.EAST);
    add(bottomContainer, BorderLayout.SOUTH);

    /*
     * Step 9: Start the Visualization
     */

    vis.run("draw");
    vis.run("xlabels");

}

/*
 * calculate the sizes of the data and axes containers based on the
```

```
     * display size, and then tell the visualization to update itself and
     * re-draw the x-axis labels
     */
    public void displayLayout() {
        Insets i = g_display.getInsets();
        int w = g_display.getWidth();
        int h = g_display.getHeight();
        int iw = i.left + i.right;
        int ih = i.top + i.bottom;
        int aw = 85;
        int ah = 15;

        g_dataB.setRect(i.left, i.top, w - iw - aw, h - ih - ah);
        g_xlabB.setRect(i.left, h - ah - i.bottom, w - iw - aw, ah - 10);
        g_ylabB.setRect(i.left, i.top, w - iw, h - ih - ah);

        g_vis.run("update");
        g_vis.run("xlabels");
    }

    /*
     * internal class that handles counting the number of elements that are
     * visible in the current view
     */
    private class Counter extends GroupAction {

        public Counter(String group) {
            super(group);
        }

        public void run(double frac) {
            // counters for population and urban centres
            double totalPopulation = 0;
            int urbanCentreCount = 0;

            // iterate through all the visual items that are visible
            VisualItem item = null;
            Iterator items = g_vis.visibleItems("canUrban");
            while (items.hasNext()) {
                item = (VisualItem) items.next();
                // add the population data
                totalPopulation += item.getDouble("2006 Population");
                // increment the counter
                urbanCentreCount++;
            }

            // if there is only one urban centre being displayed, show its
information
            // in the counter display; otherwise show the number of urban
centres and
            // the total population
            if (urbanCentreCount == 1) {
                g_totalStr = item.getString("label");
            } else {
```

```
                g_totalStr =
NumberFormat.getIntegerInstance().format(urbanCentreCount) +
                        " Cities, Total Population: " +

NumberFormat.getIntegerInstance().format(totalPopulation);

            }
            // set the text in the interface element
            g_total.setText(g_totalStr);

        }
    }
}
```

7. Example 2: Network Layout

The following Java program is an adaptation of the one provided in the Prefuse documentation. It loads a data set in GraphML format, and provides a force-directed graph layout of the data.

socialnet.xml

SocialNetworkVis.java

```
package prefuse_tutorial;

import javax.swing.JFrame;

import prefuse.data.*;
import prefuse.data.io.*;
import prefuse.Display;
import prefuse.Visualization;
import prefuse.render.*;
import prefuse.util.*;
import prefuse.action.assignment.*;
import prefuse.Constants;
import prefuse.visual.*;
import prefuse.action.*;
import prefuse.activity.*;
import prefuse.action.layout.graph.*;
import prefuse.controls.*;

public class SocialNetworkVis {

    public static void main(String argv[]) {

        // 1. Load the data

        Graph graph = null;
        /* graph will contain the core data */
        try {
            graph = new GraphMLReader().readGraph("data/socialnet.xml");
        /* load the data from an XML file */
        } catch (DataIOException e) {
            e.printStackTrace();
```

```
            System.err.println("Error loading graph. Exiting...");
            System.exit(1);
        }

        // 2. prepare the visualization

        Visualization vis = new Visualization();
        /* vis is the main object that will run the visualization */
        vis.add("socialnet", graph);
        /* add our data to the visualization */

        // 3. setup the renderers and the render factory

        // labels for name
        LabelRenderer nameLabel = new LabelRenderer("name");
        nameLabel.setRoundedCorner(8, 8);
        /* nameLabel decribes how to draw the data elements labeled as "name"
*/

        // create the render factory
        vis.setRendererFactory(new DefaultRendererFactory(nameLabel));

        // 4. process the actions

        // colour palette for nominal data type
        int[] palette = new int[]{ColorLib.rgb(255, 180, 180),
ColorLib.rgb(190, 190, 255)};
        /* ColorLib.rgb converts the colour values to integers */


        // map data to colours in the palette
        DataColorAction fill = new DataColorAction("socialnet.nodes",
"gender", Constants.NOMINAL, VisualItem.FILLCOLOR, palette);
        /* fill describes what colour to draw the graph based on a portion of
the data */

        // node text
        ColorAction text = new ColorAction("socialnet.nodes",
VisualItem.TEXTCOLOR, ColorLib.gray(0));
        /* text describes what colour to draw the text */

        // edge
        ColorAction edges = new ColorAction("socialnet.edges",
VisualItem.STROKECOLOR, ColorLib.gray(200));
        /* edge describes what colour to draw the edges */

        // combine the colour assignments into an action list
        ActionList colour = new ActionList();
        colour.add(fill);
        colour.add(text);
        colour.add(edges);
        vis.putAction("colour", colour);
        /* add the colour actions to the visualization */

        // create a separate action list for the layout
        ActionList layout = new ActionList(Activity.INFINITY);
```

```
        layout.add(new ForceDirectedLayout("socialnet"));
        /* use a force-directed graph layout with default parameters */

        layout.add(new RepaintAction());
        /* repaint after each movement of the graph nodes */

        vis.putAction("layout", layout);
        /* add the laout actions to the visualization */

        // 5. add interactive controls for visualization

        Display display = new Display(vis);
        display.setSize(700, 700);
        display.pan(350, 350); // pan to the middle
        display.addControlListener(new DragControl());
        /* allow items to be dragged around */

        display.addControlListener(new PanControl());
        /* allow the display to be panned (moved left/right, up/down) (left-
drag)*/

        display.addControlListener(new ZoomControl());
        /* allow the display to be zoomed (right-drag) */

        // 6. launch the visualizer in a JFrame

        JFrame frame = new JFrame("prefuse tutorial: socialnet");
        /* frame is the main window */

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.add(display);
        /* add the display (which holds the visualization) to the window */

        frame.pack();
        frame.setVisible(true);

        /* start the visualization working */
        vis.run("colour");
        vis.run("layout");

    }
}
```

6. Links, APIs, etc.

Below are links to the Prefuse manual and the API documents. Another excellent source of information on Prefuse is the code itself. You can force NetBeans to generate the JavaDoc by selecting the original Prefuse project and selecting Build -> Generate JavaDoc.

There are also a number of examples in the `demos` directory of the project:

- Tabular Layout

  - Congress.java
  - ScatterPlot.java
- Network Layout
  - AggregateDemo.java
  - GraphView.java
  - RadialGraphView.java
  - TreeView.java

Perfuse Manual

Prefuse API