Understanding the JUNG Visualization System

Danyel Fisher (edited by Joshua O'Madadhain)

JUNG 1.7.0 features a largely-revamped visualization system, thanks to a lot of work from Tom Nelson. While the JUNG visualization system is much more flexible than it once was, it also has a couple of tricks and a more sophisticated model.

I'll go through the major features of the JUNG 1.7.0 visualization system; when applicable, I'll refer to the demonstration code that shows how this works.

Let's walk though what's going on.

SHORT VERSION:

The correct and simplest way to create a visualization of a graph is

```
Graph g ...
Layout l = new FRLayout( g );
Renderer r = new PluggableRenderer();
VisualizationViewer vv = new VisualizationViewer( layout, renderer );
JFrame jf = new JFrame();
jf.getContentPane().add ( vv );
```

BASIC CONCEPTS

As with past JUNG systems, there is a distinction between the graph, the layout, and the renderer.

- A `Graph` knows what Vertexes are linked to each other by what Edges, and is implemented by a `Graph` class. The basic `Graph` class is `SparseGraph`.
- A `Layout` specifies where vertices are to be drawn in a visualization; it may also embody some calculation. (At the present time, edges' positions are determined by the positions of their endpoint vertices.)
- A `Renderer` is responsible for drawing vertices and edges (and thus must know how they are supposed to look). Some `Renderers` as well as any vertex/edge labels, if applicable.

As of version 1.7, we now are moving more toward a model-view-controller model. The model is responsible for knowing what to display--that is, the `Graph` and the Layout; the view/controller is responsible for knowing how to display it.

1. A `VisualizationModel` takes control of the layout process. It controls a thread that allows animated layouts to move forward; it contacts the View when its state changes. It controls the current layout (which would, in turn, have a reference to a relevant `Graph`.). The generic (and so far only) implementation is the `DefaultVisualizationModel`.

2. A `VisualizationViewer` extends a `JPanel`, and thus is the thing that does the painting. It forms the center of the Visualization complex, and thus has a number of different tasks:
    1. It tracks the Model
    2. It tracks the `Renderer`, including the PreRenderers (which paint under the topology) and the PostRenderers (which paint over the topology)
    3. Handles `ToolTipFunctions`, `PickSupports`, and `GraphMouse`, if any.
    4. If appropriate, it maintains an offscreen buffer.
    5. Applies `Transformers`, if any, to either the View or the Layout: the ViewTransformer and the LayoutTransformer.

    At initialization time:

    1. A renderer and a layout must be created and supplied to the `VisualizationViewer`
    2. A size is chosen (default: 600x600)
    3. A model is built (default: `DefaultVisualizationModel` based on the layout and the size). See `MultiViewDemo`, among others, to see examples where the same Model might be shared by several `VisualizationViewers`

IMPORTANT CHANGES FROM 1.6

ADDED:

- `VisualizationModel` (see above)
- `VisualizationViewer` has a notion of both View Scale and Layout Scale (see below)
- Improvements to `GraphMouse`, `PickSupport`, and `ToolTip` (see below)

DEPRECATED:

- `GraphDraw`. `GraphDraw` was a convenience technique for creating a `JPanel` that contained a graph. As we've refactored and cleaned up, we've realized that it was causing more confusion than needed. Most of your old code should work just fine by manually creating a `VisualizationViewer`.
- `Layout.filter()`. As we've improved the dynamic graph code, the idea that the layout should be responsible for maintaining "filters" has gotten increasingly difficult to sustain. In addition, support was uneven. It is deprecated now, and should be removed by version 1.8.
- `NewGraphDraw` was an experimental system to learn more about visualization. Most of the features that were available through `NewGraphDraw` are now available in the new JUNG visualization system.

DELETED:

- `FadingVertexLayout` has been deprecated since version 1.3, and has gotten no emails. Time to let it go.

RENDERING: Painting images, writing unicode, and curving lines

The core rendering code is the `PluggableRenderer`. While it's possible to write your own `Renderer`, the PluggableRenderer alone has a tremendous amount of flexibility; it's also possible that just inheriting from the `PluggableRenderer` will cover you.

`PluggableRenderer`'s behavior can be changed by supplying functions which supply `PluggableRenderer` with information on the properties to use for each vertex and edge that `PluggableRenderer` is asked to draw. These properties include (but are not limited to):

- vertex/edge `Shape`: `{Edge, Vertex}ShapeFunction`
  shapes provided include various geometric shapes for vertices and various curves, lines, and shapes for edges
- vertex/edge `Stroke`: `{Edge, Vertex}StrokeFunction`
- vertex/edge `Paint` (includes `Color`): `{Edge, Vertex}PaintFunction`
  outline (draw) and interior (fill) `Paint` each specified separately; these can be solid colors, gradients, ...
- vertex/edge `String` label: `{Edge, Vertex}Stringer`
- vertex/edge label `Font`: `{Edge, Vertex}FontFunction`
- vertex/edge inclusion: `Predicate`
  only those vertices/edges which satisfy the specified `Predicate` will be drawn
- edge label positioning: `NumberEdgeValue`
  value in [0,1] that specifies where edge labels are to be drawn: 0 = at source, 1 = at destination
- edge arrow type: `EdgeArrowFunction`
- edge arrow inclusion: `Predicate`
  only those edges satisfying the predicate will have arrows
- vertex `Icon` (`VertexIconFunction`)
  allows the user to supply pictures for each vertex; see `UnicodeLabelDemo` and `VertexImageShaperDemo` for examples.
- Non-rectangular image shaping
  If your images have a non-rectangular opaque part on a transparent background, you can use the `FourPassImageShaper` to extract tne non-rectangular shape from each image. This effect is demonstrated in the `VertexImageShaperDemo` where you can see that the edge arrow placement and the shape pick range both conform to the non-rectangular opaque part of the images.

Changing the functions that `PluggableRenderer` uses is very simple:

```
PluggableRenderer pr = new PluggableRenderer();
VertexStringer vs = ...
pr.setVertexStringer(vs);
pr.setEdgeShapeFunction(new EdgeShape.QuadraticCurve());
```

The `PluggableRendererDemo` shows how to use these various functions to customize your visualization. Also see the `PluggableRenderer` Javadoc for more information.

MOUSE INTERACTION and PICKING

JUNG now allows a wide variety of behaviors with the mouse. In particular, a mouse can be used to indicate or control any part of the graph, including clicking on both edges and vertices. Various among the demos show how to use the mouse to select (known in JUNG as "Pick") vertices and edges and choose which mouse event occurs when a click occurs.

- `PickSupport`

  A `VisualizationViewer` may be associated with a `PickSupport` object (this association is created via the call `vv.setPickSupport( ... )`). A `PickSupport` returns a `Vertex`, or `Edge`, based on a specified (x,y) location in a `Layout`'s coordinate space. (Some of these methods are inherited from `GraphElementAccessor`.) Generally (but not necessarily), these locations are generated by mouse events. Implementations include those that return...

  - ...the nearest `Vertex` (`ClassicPickSupport`, the default)
  - ...the nearest object closer than a certain distance (`RadiusPickSupport`)
  - ...the object (if any) whose defining Shape contains the specified point (`ShapePickSupport`)
  - ...there is even a mode that automatically centers the picked vertex with an animated transform when you CTRL-click it.

  As usual, the interface can be extended for your own use.

- `GraphMouse`

  The real truth of picking comes with a `GraphMouse`. This class catches mouse events, uses the `PickSupport` to change them into an event on the graph, and then sends them onward. The `GraphMouse` is responsible for adjusting the (x,y) location associated with the mouse event into the appropriate location in graph coordinates. A `GraphMouse` replaces a `MouseListener`, `MouseMotionListener`, and `MouseWheelListener`. The default `GraphMouse` calls pick on the current `PickedState` object for clicks. There are, however, `GraphMouse` implementations designed for a variety of tasks:

  - `PluggableGraphMouse` allows a user to select the behavior of the mouse by linking sets of modifiers to plugins. See documentation associated with `PluggableGraphMouse`.
  - `DefaultModalGraphMouse` creates a menu that lets the user choose between modes: pick (click to select; shift-click to multi-select) and transform (drag, shift-drag, and control-drag to pan, rotate, and shear; scroll wheel to zoom).

  Also check out the demo of `SatelliteViewDemo` and, in particular, its help section.

  `vv.SetGraphMouse(gm)` replaces all `GraphMouse` instances currently in use with the specified `GraphMouse` instance `gm`.

- `PickedState`, `PickEventListener`

  `PickedState` classes store the "picked" status of vertices and edges. The details of their implementation define policies such as the number of elements that may be

simultaneously selected. `PickedState` instances also allow `PickEventListener` instances to be registered with them; when elements are picked or unpicked, the `PickedState` notifies any registered `PickEventListener` of the specific event (picked or unpicked) and the affected element (vertex or edge).
`MultiPickedState`, the default implementation of `PickedState`, allows the system to maintain a set of picked points.
Historically, a `Layout` would maintain mouse information, which is why `Layouts` must implement `PickEventListener`; however, this requirement is likely to be removed in future versions.

- `ToolTipFunction`

  A ToolTipFunction specifies the text that is shown when mousing over a vertex or edge. Several of these functions are made simpler by extending `ToolTipFunctionAdapter` and overriding methods as they seem appropriate. `ZoomDemo` demonstrates the tooltip functionality. The tooltip function is set thus:

  ```
  vv.setToolTipFunction( ... )
  ```

TRANSFORMATIONS: TRANSLATION, SCALING, ROTATION, DISTORTION

In JUNG 1.6.0, we had begun to experiment with the idea that a view might be panned or zoomed. Fortunately, Java 2 makes this fairly easy, and so we've greatly expanded coverage. JUNG now supports several types of transformations, including translation (panning), scaling (zooming), rotating, shearing, and even nonlinear transformations such as hyperbolic projections.

JUNG, as of version 1.7, now offers two different types of scaling:

- **view scaling**: scaling the coordinate system in which the network is drawn. This is analogous to pointing a camera lens at the screen and zooming in or out: as you zoom in, vertex images get bigger, edges get thicker, and text gets larger; in addition, vertices get farther apart and edges become longer. See `ViewScalingGraphMousePlugin`.
- **model scaling**: scaling the coordinate system used by the `Layout`. This is analogous to stretching or compressing the surface on which the network is drawn: as the scale factor is increased, vertices get farther apart and edges get longer, but they remain the same size. See `ScalingGraphMousePlugin`.

These scaling methods are combined by the `CrossoverScalingGraphMousePlugin`, which has a `crossover` parameter that specifies a boundary (by default, set to 1.0, that is, no scaling). It works as follows:

- If the scale factor is greater than `crossover`, then layout scaling is used: the vertices stay the same size (that is, if `crossover == 1.0`, they are not scaled) and become farther apart.
- If the scale factor is less than `crossover`, then view scaling is used: vertices become smaller and closer together.

This "crossover scaling" is used in several demos, including GraphZoomScrollPaneDemo. This and other types of transformations can be seen in SatelliteViewDemo (scaling, panning, shearing, rotating) and HyperbolicLensDemo (hyperbolic projection).

Instead of using a JScrollPane...

JScrollPane will not give expected behavior as a container for the VisualizationViewer. Instead, use the GraphZoomScrollPane, a custom container that both responds to, and controls transformations of the VisualizationViewer; it provides vertical and horizontal scrollbars when the entire graph does not fit in the window. It listens for changes to the VisualizationViewer (zooming, panning) and adjusts the scrollbars' size and position accordingly.

Specifying Transformations

If you want to specify the transformers to use directly (as opposed to through a GraphMousePlugin), call vv.setLayoutTransformer() or vv.setViewTransformer() with an appropriate implementation of Transformer. In general, a Transformer is responsible for mapping points in one coordinate system to points in another. Because this interface is fairly generic, it even supports non-linear transformations. Check out the hyperbolic view defined in HyperbolicTransformer, which supports hyperbolic transformations over an affine 'delegate'.

Multiple Views of a Graph

The SatelliteVisualizationViewer is intended to provide an overview of the graph. It links to a VisualizationViewer and can share that VV's layout and use a similar renderer. This allows the system to display a small copy of the same graph for navigation; see SatelliteViewDemo for an example of how that works.

PRE-RENDER and POST-RENDER

Sometimes, you want to put text over top of a graph, or lay something out underneath it. The VisualizationViewer interface Paintable allows you to define a surface to be painted; UseTransform says whether it should zoom with the graph or should stay constant. Demonstrations for this include:

- GraphZoomScrollPaneDemo uses it to paint the texture on the background and the banner label on the foreground
- SatelliteViewDemo uses it to paint the rectangular lens and the optional grid.

SUBLAYOUTS

While Layouts cover an entire graph, a SubLayout picks locations for a subset of vertices and lays them out in a tight grouping that serves as a visual proxy for all the vertices.

While traditionally a Layout can store its data in a variety of places, and uses a whole graph to

decorate, the `SubLayoutDecorator` decorates a `Layout` with the knowledge that it may have to store a `SubLayout`. The design for this mechanism is still in progress.

Check out `SubLayoutDemo` to manually select vertices, or `ClusteringDemo` to see how automatic sublayouts may be chosen as appropriate.

WHEN THE GRAPH CHANGES

`ChangeEventSupport` and `ChangeListener` are now standard mechanisms for recognizing when either the underlying graph or the underlying model changes: that is, when a vertex gets moved or the graph changes. Both of these should trigger `ChangeEvents` which should in turn, trigger transformations and repaints as appropriate.

CAPTURING THE MOMENT: DUMPING TO PNG, JPG, or EPS

For JUNG 1.5 and before, there was no problem simply rendering the screen to a Buffer, and then saving that as a PNG or a JPG. As of JUNG 1.6, we added offscreen buffers that partially accelerated drawing--but didn't support this output modality. (And it wrecked the ability to swap in a new Graphics object).

JUNG 1.7 allows the best of both worlds. Double-buffering is off by default, but can be turned on for to increase performance. When the graph looks like you want, turn off double-buffering and then call a PNG, JPG, or EPS dumper. A GPL EPS dumper can be found at http://jibble.org /epsgraphics/; sample JPG and PNG dumping code is below.

```
// use double buffering until now

// turn it off to capture
visualizationViewer.setDoubleBuffered( false )

// capture: create a BufferedImage
// create the Graphics2D object that paints to it
visualizationViewer.paintComponent( replaced_graphics2D )
// and save out the BufferedImage
ImageIO.write(bi, "jpg", new file( ... ));

// turn double buffering back on
visualizationViewer.setDoubleBuffered( true )
```