THE GLYPHICS HIERARCHY

By

Ben Funnell

A thesis submitted for the degree of

Bachelor of Computing Science (Honours)

of The University of New England

2004

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many applications require an easy to use interactive graphics package to display information to the screen. An interactive graphics package is perfect for displaying internally complex data, so less skilled users can analyse and manipulate the data.

There is already a package available called `Ezd`, written by Joel Bartlett in Java 1.0, using features available then [1]. `Ezd` lacks some of the functionality required to create graphics objects in the new Java 2 environment. It is also unable to easily implement aspects required to create and manipulate graph objects, like translation and rotation of compound shapes.

A solution is `Glyphics` which is loosely based upon Joel Bartlett's Ezd, with some changes to improve the structural design and utilise new Java features. `Glyphics` is designed to be easily extended, so custom graphical objects can be created. These custom graphics objects can be used to display all sorts of information.

This package was produced with some specific applications in mind, but has been developed so it can be effectively used in other graphical applications. The main application of `Glyphics` is to be combined with `JLambda`, an easily parsed, run time interpreted, scheme like language [2], to enable dynamic creation of graphical objects and advanced graphical user interfaces.

1.1 Joel Bartlett's `Ezd`

Joel Bartlett's `Ezd` is an easy to use graphics drawing package. With `Ezd` arbitrary graphical elements can be created, displayed to a screen, and react to events, and user input.

In `Ezd`, drawing and event capturing is handled by an `EzdView` object. An `EzdView` draws it's contained `Glyphs` by calling each `Glyph's` paint method. In `Ezd`, a `Glyph` is responsible for drawing itself, typically as a sequence of shapes. This painting method implements the painters algorithm, colors are opaque and new objects may obscure previously drawn objects. `Ezd's` implementation of compound glyphs is limited, and needs to mimic the behaviour of Java's Collection interface, with methods like; add, remove, clear, size, etc.

The `EzdView` object is responsible for passing the event to the appropriate `Glyph`. Once the event has been captured, it is up to the `Glyph` object to handle it. The `Ezd` package was developed using the deprecated Java 1.0 event model, which is based on inheritance. For a program to catch and process GUI events, it must subclass the GUI components and override their event methods. Events are always delivered to components regardless of whether the components handle them or not. This is considered a severe performance problem.

1.2 Java Developments

Since Java 1.0, enhancements have been made to the language which can be used to improve Joel Bartlett's Ezd package.

1. Java's new event system [3].
2. Swing and the Graphics2D package [4].
3. Affine Transforms (`java.awt.geom.AffineTransform`) [5].

The `java.awt.Graphics2D` class extends the `Graphics` class to provide more sophisticated control over geometry, coordinate transformations, color management, and text layout. Forming the fundamental class for rendering 2-dimensional shapes, text and images [6].

With the newer 2D implementation of the `java.awt.Shape` interface, and Java's implementations of the `java.awt.Shape` interface, many geometric and compound shapes can be created. Shapes can be drawn to the screen with a range of `java.awt.Strokes` and `java.awt.Colors`. These shapes support affine transforms, which allows them to be easily manipulated. `Glyphics` also provides many `java.awt.RenderingHints` which can be used to tweak the performance and quality of rendered objects.

`Java.awt.Graphics2D` can easily and clearly display text to the screen. The `java.awt.font` package has been improved to support `Graphics2D`. High detail `java.awt.Shape` objects can be created from `Strings` which can be rendered to the screen.

When combined with the `KEY_ANTIALIASING` rendering hint, `Graphics2D` can display clear, unpixelated text. Images are also easily rendered to the screen as a `BufferedImage`. Although use of Java's IO classes is required to read an images from file.

With the `java.awt.geom.AffineTransform` class, Shape objects, Image objects, and even the graphics object itself can be transformed. In fact almost anything you can render using `Graphics2D` can be manipulated by Affine Transforms. Objects can be translated, rotated, sheared, and scaled. Affine transforms are simple and efficient, making it simple to manipulate a complex compound shapes, which is otherwise complicated.

The `java.awt.Graphics2D` package uses double precision coordinates and values for all its components. This is an improvement on Joel Bartlett's `Ezd` package, which used integer coordinates, since it was implemented in Java 1.0. This was frustratingly inaccurate, requiring calculations to be rounded. This rounding causes a lack of accuracy and caused errors in geometry calculations. For example, rotating an arrowhead in integer precision causes the coordinates of the arrowhead to deform and contort. Using double precision in calculations reduces the magnitude of rounding errors.

Chapter 2

Glyphics

This is the next generation `Ezd` package. Glyphics is a hierarchal graphics display library, all operations upon Glyphics objects are through this hierarchy. This library is simple use, because all graphics objects behave in the same manner. Glyphics takes advantage of the improvements to the Java language since `Ezd` was implemented. With this package, some custom Glyphics objects are implemented to handle more complicated graphics objects like text and images.

2.1 The Glyphish Hierarchy

In our approach the root class of all things glyph-like is the `abstract` class `Glyphish`. It has three main immediate concrete subclasses: the `Glyph`, the `GlyphList`, and the `ClosureGlyph`.

The `Glyphish` class defines the interface that each of the concrete subclasses responds to. Creating this sub-type hierarchy makes it simple to use, as all things of type `Glyphish` behave the same. This hierarchy is the means for all `Glyph` communication and manipulation.

There are three related but distinct aspects to the `Glyphish` class:

1. How a `Glyphish` instance depicts or portrays itself graphically.

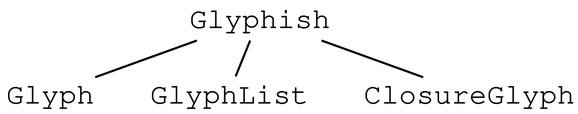

Figure 2.1: The Glyphics hierarchy

2. How a `Glyphish` instance handles input events from the keyboard and mouse.

3. How a `Glyphish` instance positions or transforms itself.

All these operations are controlled by the encompassing containment hierarchy, we will postpone discussion of the component hierarchy until a chapter 3. The concrete subclasses differ in their implementation of these common features, and we will touch on the differences, once we have explained the similarities.

2.1.1 Glyphish

We will begin by describing the base `Glyphish` class, this class is the basis of all the other Glyph classes.

`Glyphish` is composed of four abstract methods which control the way objects of type `Glyphish` are painted, transformed and respond to events. These methods must be implemented in any concrete sub-class of `Glyphish`. The most important of the abstract methods is:

```
void abstract paint(Graphics2D g2d)
```

Which is responsible for painting an instance of a `Glyphish` sub-class to the screen. Since many objects are drawn differently, this is where these differences will be implemented.

Event detection in an instance of `Glyphish` is achieved with:

```
boolean abstract inside(Point2D point)
```

Which returns true if the point is contained within the instance of `Glyphish`.

For ease of manipulation, `Glyphish` instances have an associated `Rectangle2D` object which decribes their size and position

```
public abstract Recangle2D getBounds()
```

This `Rectangle2D` can be used in transform calculations, event handling, or if the size and position of a `Gyphish` instance needs to be known.

The method used for moving and transforming `Glyphish` instances is:

```
void abstract transform (AffineTransform a)
```

Manipulating `Glyphish` instances is achieved by using this method to apply affine transformations (e.g. translating, rotating, shearing, and scaling) to the internal representation of the `Glyphish` instance. To simplify affine transforms, five convenience methods are provided:

```
void rotate(double thetaRad)
void rotate(double thetaRad, double x, double y)
void scale(double x, double y)
void shear(double x, double y)
void translate(double x, double y)
```

which enable one to transform a `Glyphish` instance without having to first construct the corresponding `AffineTransform` object. The rotate methods are used to either rotate a `Glyphish` instance around its centre or around a point. A scale and shear of one does no transforming. Translating a `Glyphish` instance is relative to to its position. All transforms are relative.

As well as the four abstract methods, `Glyphish` class implements each of the `Input` event listener interfaces: `MouseListener`, `MouseMotionListener`, and `KeyListener`, all of the package `java.awt.event`. Any other events could easily be implemented in sub-classes.

For each method in the listener interface a `Glyphish` instance has a `Closure` object associated with it. For example, in the case of the

```
public void mouseClicked(java.awt.event.MouseEvent e);
```

method of the `MouseListener` class, the `Glyphish` class has the field

```
private g2d.jlambda.Closure  mouseClickedAction;
```

This closure will have arity 2, and uses Luca Cardelli's trick of having a self argument to implement Java's `this` pointer. Calling the `mouseClicked` method would result in the `clickedAction` closure being applied with

```
clickedAction.applyClosure(this, e);
```

where the `this` pointer is the `Glyphish` instance that is responding to the
`java.awt.event.MouseEvent` instance `e`.

These two methods help `Glyphish` instances associate closures to events in `JLambda`:

```
void setKeyAction(int type, Closure fun)
void setMouseAction(int type, Closure fun)
```

which dynamically sets an action (`fun`) to an event (`type`). The `type` argument corresponds to the events identifiers, i.e. `MouseEvent.MOUSE_CLICKED` and `MouseEvent.MOUSE_DRAGGED`.

2.1.2 Glyphish's direct sub-classes

The direct sub-classes of `Glyphics` are `Glyph`, `GlyphList` and `ClosureGlyph`. The `Glyph` class is an atomic subclass of `Glyphish`. It can draw a single `Shape` object, according to `Glyph's` private fields `fillColor`, `borderColor` and `Stroke`. A `GlyphList` is a composite subclass of `Glyphish`, it consists of an ordered list of `Glyphish` instances. This is `GlyphList's` only field, the actual drawing is achieved by the contents of the list. `ClosureGlyph` is the most dynamic, it requires closures [2] to implement all the methods required by the `Glyphish` API. It provides, in essence, a way of defining `Glyphish` instances whose methods are defined at runtime, rather than at compile time.

Painting

`Glyph` objects draw themselves according to their private fields; `java.awt.Shape`, fill colour, border colour and `Stroke`. Any change to these fields will appear when the instance is repainted. All painting is applied to the `Graphics2D` instance. A `Graphics2D` object draws by setting its `Stroke` and colour, then the `Shape` is drawn or filled.

`Glyph's` implementation of the abstract paint method follows:

```
public void paint(Graphics2D g2d) {
    if((fill != null) && (shape != null)) {
        g2d.setColor(fill);
        g2d.fill(this.shape);
    }
    if(border != null) {
        g2d.setColor(border);
        g2d.setStroke(stroke);
        if(shape != null){
            g2d.draw(this.shape);
        }
    }
}
```

A `GlyphList` iterates through and paints each `Glyphish` element in its list, hence implementing the painters algorithm.

The `ClosureGlyph` responds by applying it's

```
private g2d.jlambda.Closure paintClosure;
```

field to the appropriate `java.awt.Graphics2D` object.

Events

To handle events, the `Glyphish` hierarchy requires an encompassing object to listen for events, this is implemented as `IOPView`. For the `IOPView` to determine whether a `Glyphish` instance is the desired target of an `Input` event the `abstract` method

```
public abstract boolean inside(java.awt.geom.Point2D p);
```

of the `Glyphish` class is used.

The `Glyph` class implements this by using it's `java.awt.Shape`'s field corresponding

```
public boolean contains(java.awt.geom.Point2D p);
```

method.

The `GlyphList` instance implements this by iterating through its list of `Glyphish` elements calling each element's `inside` method, returning `true` if one returns `true`, else it returns `false`.

The `ClosureGlyph` responds by applying it's

```
private g2d.jlambda.Closure insideClosure;
```

field to the appropriate `Point2D` object.

Transformation

`AffineTransforms` can be applied to subclasses of `Glyphish`

`Glyph` instances apply the `AffineTransform` to it's `Shape` object. This creates a new `Shape` object which is stored in the `Glyph` instance.

For `GlyphList` instances, the `AffineTransform` is applied to each `Glyphish` element conatined in its list. While the `ClosureGlyph` simply applies the transform to

```
private g2d.jlambda.Closure transformClosure;
```

2.2 How to use Glyphics effectively

There are a number of ways to create displayable objects with the `Glyphics` hierarchy. If the object is simple and does not require custom events or other custom behaviour, it is best to create an instance of `Glyph` or `GlyphList`. Many geometric shapes can be created with the `java.awt.Shape` interface, and many more can be created by combining `Glyphs` in a `GlyphList`. If the displayable object needs extra properties, it is best to sub-class `Glyph` or `GlyphList`.

This simple example extends the `Glyph` class to create a `RectangleGlyph`.

```
public class RectangleGlyph extends Glyph {
    private Color color;

    public RectangleGlyph(double x, double y, double width,
                          double height, Color color) {
        this.color = color;
        Rectangle2D rect = new Rectangle2D.Double(x, y, width, height);
        setGlyph(rect, this.color, null);
    }

    public static void main(String[] args){
        IOPView view = new IOPView();
        IOPFrame frame = new IOPFrame("RectangleGlyph", view);
        RectangleGlyph rect = new RectangleGlyph(20, 30, 100, 40, Color.red);
        view.add(rect);
        frame.setVisible(true);
        view.repaint();
    }
}
```

The main of this example can be replicated in `JLambda` as follows (the path of the `RectangleGlyph` must be known to the `JLambda` interpreter):

```
(let (
    (red java.awt.Color.red)
    (view (object ("g2d.swing.IOPView")))
    (frame (object ("g2d.swing.IOPFrame"
            "Rectangle example" view)))
    (rect (object ("RectangleGlyph" (double 20)
                                    (double 30)
                                    (double 100)
                                    (double 40)
                                    red))))
```

```
(seq
    (invoke view "add" rect)
    (invoke frame "setVisible" (boolean true))
    (invoke view "repaint")))
```

2.3 Specialised Glyphs

Some specialised glyphs are provided to handle more complicated graphical tasks. These specialised glyphs deviate slightly in the way they are painted from regular glyphs. However the differences keep in line with the structure and design of the base Glyphics hierarchy.

2.3.1 TextGlyph

Almost all graphical applications need text to be able add labels and other written information to the display. `TextGlyph` is a specialized `Glyph` object (see hierarchy image 2.2) which displays aligned, anti-aliased[1] text, which can be seen in images 2.4 and 2.3. `TextGlyph` works by overriding the `Glyph` class's paint method. In a normal glyph the outline of the contained `Shape` is drawn and possibly filled with a colour. But in `TextGlyph` this causes the text to be unreadable. `TextGlyph's` paint method fills the text's `Shape`, and does not draw its outline. Although this creates a fast implementation of a text glyph, the text is still pixelated, and becomes difficult to read at small sizes. Anti-aliasing the whole `Graphics2D` object was not practical, since this slowed down each redraw to an unacceptable level. But, anti-aliasing just the text was a viable solution, this option provided clean, easy to read text without a huge cost to performance. `TextGlyph's` performance is affected by applying the anti-aliasing option, the cost of the performance slow down is acceptable, because of how clean the text looks. The performance issue may only become a problem when large masses of text are displayed. A demonstration of `TextGlyph's` performance can be seen in the affineTransform demo [7], in this animation, a `TextGlyph` is warped, translated and rotated via `Glyphish's` affineTransform interface. The speed of the animation, even as the text string gets large, shows the efficiency and speed of `TextGlyph's` paint method.

Here is the code for the modified paint method.

```
public void paint(Graphics2D g2d){
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
```

[1] Anti-aliasing is the removal of on-screen pixelation by drawing partially filled pixels lighter than normal pixels

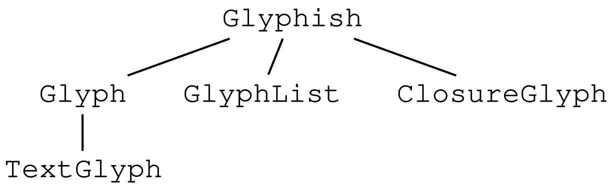

Figure 2.2: Glyphics hierarchy with `TextGlyph`

Figure 2.3: Anti-aliased text.

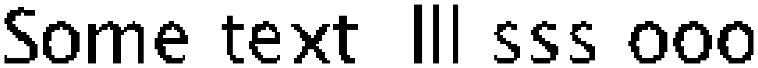

Figure 2.4: The same text without anti-aliasing.

```
                              RenderingHints.VALUE_ANTIALIAS_ON);
        Color filling = getFill();
        if(filling != null) {
            g2d.setColor(filling);
            g2d.fill(getShape());
        }
        g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                             RenderingHints.VALUE_ANTIALIAS_OFF);
    }
```

Each `TextGlyph` can be aligned relative to a bounding box, or point. The bounding rectangle of the `TextGlyph` is used to position the text relative to the inside or outside of a bounding box. Alignment operations can be applied together to form compound alignment, for example, a `TextGlyph` can be positioned in the center of, but on top of a given bounding box by applying both the `alignCenter` and the `alignOnTop` operations. The `alignCenter` operation aligns text to the center of the `Rectangle` horizontally vertically. The bounding box can be the bounds of another `Glyph` which makes it simple to align text relative to other `Glyphish` objects. `TextGlyph's` font, size, and face can be set with appropriate setters, providing full featured text control.

In the following example, some text is going to be aligned so it looks like image 2.5. The text

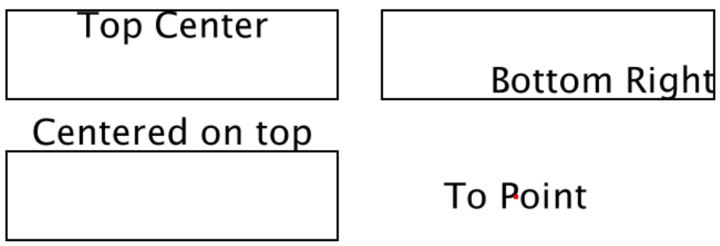

Figure 2.5: Some aligned text

and box objects have already been initialized. As you can see, alignments can be combined and the text can be aligned relative to the inside of the box, outside of the box or centered at a point.

In this example `topcenter` and `topoint` are `TextGlyphs` and boxes are objects with a `public Rectangle2D getBounds()` method, which could be `Glyphish` instances.

```
(invoke topcenter "alignTop" (invoke box1 "getBounds"))
(invoke topcenter "alignCenter" (invoke box1 "getBounds"))

(invoke topoint "alignToPoint" (object ("java.awt.geom.Point2D$Double"
                                         (double 250) (double 300))))
```

2.3.2 SubTextGlyph

Some applications need sub-scripts and super-scripts to properly display mathematical text. `SubTextGlyph` is an extension of `TextGlyph` (see hierarchy image 2.6) which displays anti-aliased text with a TeX like syntax for subscripts and superscripts. For example, `A^{23}` displays A^{23} and `G_{12}` displays G_{12}. They can be nested like so, `A^{G_{12}}`. The input string is parsed with a single recursive method, which can easily handle nested subscripts.

Like TeX, `SubTextGlyph` does not support subscripts of subscripts (i.e. `A_B_C`). When this error is encountered, `SubTextGlyph` stops parsing the string, displays an error message to standard error, and creates and displays the glyph up-to the error. A demonstration is available [8] showing `SubTextGlyph` in its recursive glory, it also shows the handling of the subscript of subscript error. There is a soft limit on how many nested subscripts a `SubTextGlyph` can have, since text sizes below one point are not supported by Java's `Font` class.

This is the JLambda form of the `SubTextGlyph` constructor. The created `SubTextGlyph` looks like image 2.7.

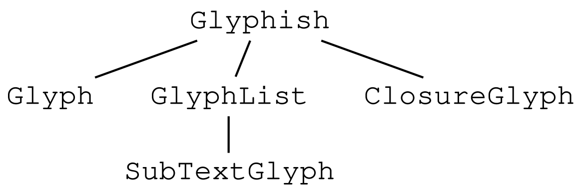

Figure 2.6: Glyphics hierarchy with `SubTextGlyph`

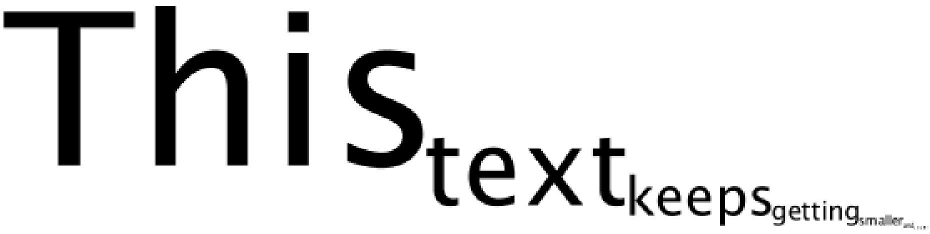

Figure 2.7: SubTextGlyph can display nested sub-scripts.

```
(text (object ("g2d.glyph.SubTextGlyph"
    "This_{text_{keeps_{getting_{smaller_{and_{smaller}}}}}}"
    (int 64))))
```

This is the java form of the `SubTextGlyph` constructor. The created `SubTextGlyph` looks like image 2.7.

```
SubTextGlyph text;
text = new SubTextGlyph(
             "This_{text_{keeps_{getting_{smaller_
               {and_{smaller}}}}}}", 64);
```

Once constructed these objects need to be added to an `IOPView` to be displayed.

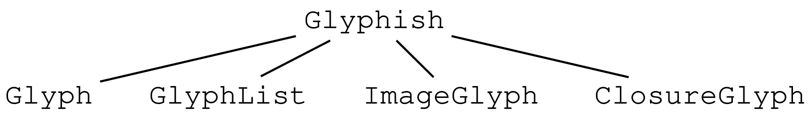

Figure 2.8: Glyphics hierarchy with `ImageGlyph`

2.3.3 ImageGlyph

Images can be added to a graphical application to display objects which otherwise could not be created with a `Shape` object. `ImageGlyph` allows images to be used to create `Glyphish` instances, which can be drawn by the Glyphics hierarchy (see hierarchy image 2.8). `ImageGlyphs` are different from other `Glyphs`, as the internal storage type of `ImageGlyph` is not a Shape object, but a `java.awt.image.BufferedImage`. This requires a different concept of painting, transforms, and bound management. Each `ImageGlyph` maintains its own transform object, this helps in calculating the image bounds once it has been transformed. The `ImageGlyph` implementation of `Glyphish's` abstract paint method draws the `BufferedImage` to the `Graphics2D` object. An `AffineTransformOp` linearly maps the image to the coordinates of the `Graphics2D` object. Linear mappings include nearest neighbour and bilinear interpolation. Nearest neighbour interpolation causes the image to become pixelated (see image 2.9), while bilinear interpolation blurs image pixels together (see image 2.10), similar to JPEG images.

Here is the source for the modified paint method.

```
public void paint(Graphics2D g2d){
    AffineTransformOp ao;
    ao = new AffineTransformOp(transform, interpolationType);
    g2d.drawImage(bImage, ao, 0, 0);
}
```

This is the JLambda form of the ImageGlyph constructor.

```
(image (object ("g2d.glyph.ImageGlyph"
            "Images/FlightControlPanel.gif")))
```

This is the Java form of the ImageGlyph constructor.

```
ImageGlyph image;
image = new ImageGlyph("Images/FlightControlPanel.gif");
```

2.3.4 AnimatedGlyph

`AnimatedGlyph` is a multi-state image object. `AnimatedGlyph` is very similar to `ImageGlyph` (see hierarchy image 2.11). The animation is achieved by storing a list of BufferedImage objects, and painting certain elements of the list when requested. The current implementation has a linear sequence, but could be used where a multiple state `ImageGlyph` is required. Creating an `AnimatedGlyph` is similar to `ImageGlyph`, except the path is to a directory of images.

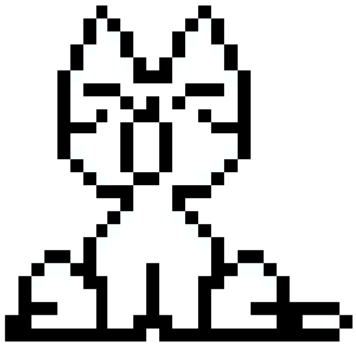

Figure 2.9: Nearest neighbour interpolation pixelates the image.

Figure 2.10: Bilinear interpolation blurs each pixel.

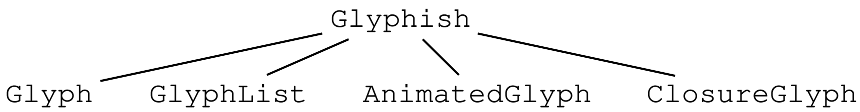

Figure 2.11: Glyphics hierarchy with `AnimatedGlyph`

Chapter 3

Containment Components

For `Glyphish` objects to handle painting and events, there needs to be an enclosing component. `IOPView` is the top level container of the underlying Glyphish objects. It directly utilises Glyphish's paint methods, the view controls when and how the contained Glyphish hierarchy is painted. The `IOPView` can zoom its display with an `AffineTransform` instance. It controls how mouse and key events are directed to the underling hierarchy. `IOPView` is composed of many sub-components, the core compontent is the `IOPComponent`, which actually contains and draws `Glyphish` instances. `IOPView` can include an optional `IOPScrollPane`, and `IOPToolBar` for added functionality. Creation of an `IOPView's` toolbar and scrollpane can be achieved with boolean flags in `IOPView's` constructor:

```
public IOPView(boolean scrollbars, boolean toolbar)
```

`IOPView` forwards most operations to the underlying `IOPComponent` independently of whether a toolbar or scroll pane exists.

3.1 Painting

Although `Glyphics` has a well defined painting hierarchy, it cannot display itself to the screen without an enclosing object to control when the painting will occur. The containment hierarchy is responsible for painting its contents to the screen. `IOPView` contains a `GlyphList` instance variable which stores the objects to be displayed to the screen. `IOPComponents` are extensions of `javax.swing.JComponents` and its subclasses, so drawing is controlled by the `protected void paintComponent(Graphics g)` method. To utilise `JComponent's` double buffered painting hierarchy this method needs to be overridden.

```
public void paintComponent(Graphics g) {
    Graphics2D g2d = (Graphics2D)g;

    g2d.setColor(getBackground());
    g2d.fillRect(0,0, getWidth(), getHeight());
    g2d.setColor(getForeground());

    g2d.transform(zoomTrans);
    glyphList.paint(g2d);
}
```

In our `IOPComponents`, this method paints the background, this is explicitly required, otherwise the background stays grey. After the background is painted, the zoom transform is applied to the `Graphics2D` object. Then the `GlyphList` is painted. This uses the custom Glyphics hierarchy to efficiently draw any number of `Glyphs` to the screen.

3.2 Zooming

In the `JComponent` hierarchy objects are displayed to the screen using a `Graphics2D` object. When an object is painted, it is painted on this `Graphics2D` object. Controlling this object directly controls what is displayed to the screen. The `Graphics2D` object can be transformed with affine transforms.

Zooming scales the `Graphics2D` object, rather than scaling each `Glyph`, this does not change the original `Glyphish` instance, only how it is displayed. Zooming this way requires a mapping from the stored representation of the `GlyphList` to the displayed representation of the `Graphics2D` object. This is easily achieved with the use of affine transforms and their inverses. The zoom transform is stored so it can be applied to the `Graphics2D` object every time the view is painted. This is a very efficient method to zoom the view because it does not require a double traversal of the `GlyphList` (one for the transform, and one for the redraw).

When zooming, the view ensures that mouse events get mapped to the correct `Glyphish` instance. This mapping is implemented by creating an inverse of the zoom transform, which maps the mouse event's coordinates to the original `Glyphish` instance.

When the view is zoomed, it is necessary to maintain the `preferredSize` of the `IOPComponent`. The `preferredSize` of a component is used to calculate the range of the scrollbars in a scroll pane. This is implemented in `IOPComponents checkSize()` method, which is called each time the scrollbars need to be updated. The implementation is simple, the new size of the component is calculated, and set with `setPreferredSize(...)` and the scrollbars are updated with `revalidate()`. Mouse wheel zooming is built into the `IOPComponent` class, providing a fast and easy way to zoom in and out (provided the user has an appropriate mouse).

3.3 Sub-Components

3.3.1 ToolBar

A basic toolbar was required to handle `IOPView's` built-in zooming capabilities (zoom-in, zoom-out, and zoom-to-fit). This toolbar can be added to the view with a boolean value passed to the constructor. `IOPView` has the ability to add `IOPButtons` to the basic tool bar, either after the existing buttons, or at an specific location. Which enables a straight forward way to add extra buttons as they are needed. These buttons can provide application specific features. There are two good examples [9] [10] showing the use of the toolbar and other `IOPComponent` features.

3.3.2 Scroll Pane

When the view is zoomed, or the view is large, sections of the view can be outside of the window, so scrollbars are needed to navigate around the view. Scrollbars can be added to a view with a boolean value passed to the constructor. While zooming with scrollbars, it may be necessary to center the view upon a specific graphical item, this can be achieved with;
`public void centerOn(Glyphish g)` or
`public void centerOn(Rectangle2D r)`.
These methods will center the view upon a `Glyphish` object or to a `Rectangle2D`.

3.3.3 Buttons

`g2d.swing.IOPButton` is a convenience class, `IOPButton's` constructor is simple;

```
IOPButton(String image, String tooltip)
```

`image` is the path to an image contained in the `Glyphics` package, or a regular path to an image, or the buttons label if the image does not exist in the previously checked locations.

Chapter 4

Graphs

The graph package is a specialised application of `Glyphics`, which provides an interactive graphical representation of an underlying structure. The displayed graph is drawn cleanly and efficiently to the screen with the new Glyphics library. The graph object, like all glyphs, is displayed in an `IOPView`, which is a class designed to provide two-way interaction with the contained glyph object. The graph is displayed in a pleasing, easy to read manner, and is capable of accepting events to control, query, and manipulate the graph. The combination of the graph hierarchy and `IOPView`, allows the dynamic creation of interactive graphs, wrapped in an interactive graphical user interface.

4.1 Object Model

Object Oriented languages are perfectly suited to representing graphs. The concepts and forms of theoretical graph concepts can be applied directly to the Object Oriented paradigm. Each node and edge can be represented as an unique object with its own specifications. Different object representations of graphs can be used, depending on what is required of the graph object.

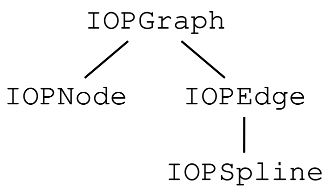

Figure 4.1: A ”composed of” hierarchy for the graph classes

An adjacency list or matrix can be used to represent the graph, which provides fast access of connected nodes for use in traversal algorithms. This cannot be the only internal representation of the graph, because in this form the edge objects are not specified, so how can the edges be drawn? This can be an alternate form of the graph to improve efficiency of graph traversal or path finding algorithms.

In an object oriented language like Java, when a complicated component is created out of many objects, the relationships between the individual objects will form a graph structure. Which could be the conceptual basis of the structure of graph objects, this structure is much slower to traverse than a single list of graph elements. In its current form, the `ArrayLists` used for the `GlyphList` could be replaced by an object of interface type `Collection`, then a `GlyphList` structure which mimics the structure of the graph could be implemented. Traversal of the `GlyphList` will need to be completed every time `IOPView` requests a redraw, or other global graph operation. If the structure is complicated, traversal could be slow.

In our case where the graph object needs to be drawn in an `IOPView` with the `Glyphics` package and processed by a layout renderer, the best representation for a graph is a linearly accessable `Collection` of nodes and edges. This way, `Collections` of nodes and edges can be used directly to create the dot input file. The dot output file is also in node/edge list form, so the nodes and edges can have position and other attributes set when the graph is layed out. Drawing compound glyph objects with the Glyphics library requires the glyphs be in a linearly accessable list.

4.2 Graph Components

4.2.1 Nodes and Edges

The node and edge classes are only really used for control of events and the graphical representation of graphs. Nodes and edges have attributes to control the way they are drawn. The attributes are Strings to be compatible with dot and the layout parser.

When an edge is drawn, with dot [11] as the renderer, it is drawn as a series of cubic spline curves. `IOPSpline` creates a single curve from a series of cubic spline curves. The spline curve is sent to `IOPSpline` as an array or vector of points in the form:

```
[<spline segment startpoint> <ctrl pt 1> <ctrl pt 2>
<spline segment endpoint> <ctrl pt 1> ....]
```

Note that the endpoint of one cubic spline curve segment is the start point for the next, one spline curve flows seamlessly into the next. This allows any number of complicated curves to be drawn as graph edges. So our graphical representation of the graph can display the edges generated by dot, which bend around nodes, other edges or anything else in the way.

4.2.2 Graph

The internal representation of the graph objects are stored as two `HashMaps`, one for nodes, the other for edges. Each node or edge is referenced in the graph's hashmap's by its unique name. This unique name is very important since it is how the layout renderer distinguishes between and locates graph elements.

Creation of an `IOPGraph` object is via its default constructor. Nodes and edges can be added and removed with:

```
void addNode(IOPNode node)
void addEdge(IOPEdge edge)
void rmNode(IOPNode node)
void rmEdge(IOPEdge edge)
```

Once the graph is created and contains some nodes and edges, the graph can be rendered with:

```
void doLayout()
```

This method positions the nodes and edges by creating an input file for dot, and processes this file with dot.

The graph object is sent to the layout parser, and the dot output file is parsed. When this method is completed the nodes and edges should be positioned as dot specified.

Nodes can be selected and de-selected by using:

```
void toggle(IOPNode node)
```

which toggles the nodes colour, and adds or removes it from the list of selected nodes.

Selection of graph elements enables the graph to be edited. The implementation of graph editing is designed to be used with mouse events. Adding a node is the simplest of the graph editing operations, create a node at the position specified, and add it to the graph. Creating an edge requires two nodes to be selected, and a call to:

```
public void createEdge()
```

which will create an edge between from the first selected node to the second selected node. This method checks the number of selected nodes, so an edge can only be created if and only if there are two selected nodes. Removal of nodes can be achieved with;

```
public void removeSelectedNodes()
```

Which will remove the selected node and its connected edges.

Selection also provides the means for zoom control, the `zoomtofit()` method of `IOPView` uses the graph's `getBounds()` method to calculate the region to zoom to. The graphs bounds is either the whole bound of the graph, or the bounds of the selected node(s).

After altering a graph elements attribute, the graph needs to be redrawn to display the changes.

See [12] for a simple example of graph creation and layout. Also see [10] for a more complex example involving the graph editing capabilities.

4.3 Display

The graph class is an extension of Glyphics' `GlyphList` class, which is a compound glyph that can contain many `Glyphs` which are drawn as one. Whenever any graph element is added to the graph the internal `GlyphList` object is updated so the new element will be drawn.

Nodes and Edges are also `GlyphLists`. A node is composed of the node base, which can be any shape defined in `java.awt.Shape` interface, an `ImageGlyph`, an `AnimatedGlyph`, or any other `Glyphish` object. An edge is composed of a cubic spline curve and one (or two) arrowheads. Each graph component is responsible for drawing itself; this is the behaviour of all `Glyphish` objects.

Care is required when designing how Glyphics draws the graph objects, so as to not create an overly complex Glyphics list structure. If the list structure mimics the structure of the graph, traversal of the graphs `GlyphList` will be slow and inefficient. Since the `GlyphLists` paint method will iterate into lists of lists of lists... etc, this nested traversal would be very slow for large graphs. Having lists of lists creates list overhead, as each list object contains some data which identifies it as a list. As a result, flattening the graph's `GlyphList` into a list of nodes and edges, provides sufficient usability and efficiency.

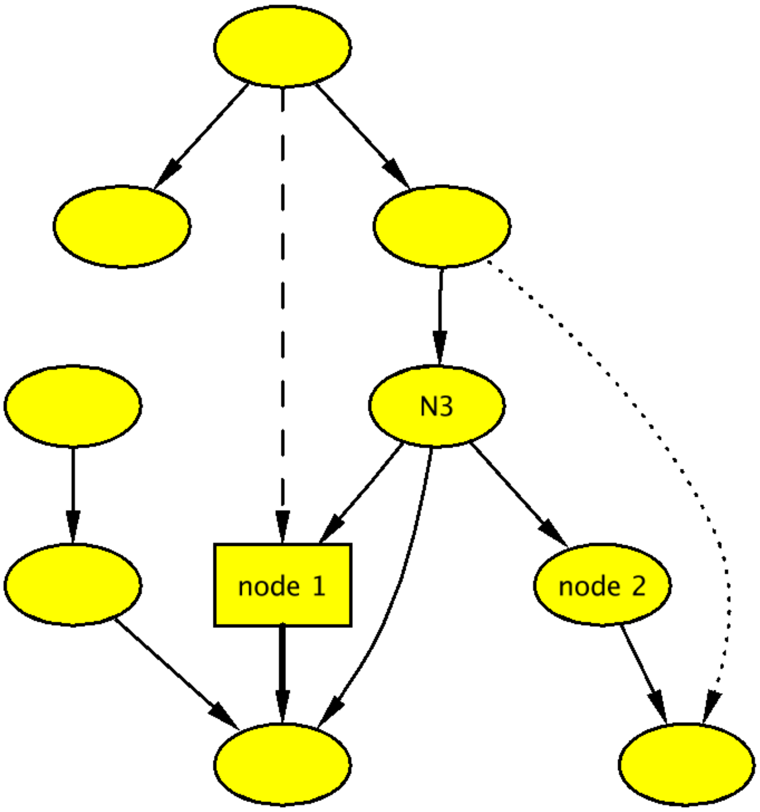

Figure 4.2: A graph with nodes and edges positioned by dot.

4.4 The Layout Renderer

Dot is a third party graph layout program [11] [13], which implements an efficient four pass algorithm for laying out directed and undirected graphs. Image 4.2 shows dot's layout in action. To use dot, a dot input file needs to be created which represents the graph, and then the file is processed by dot. This creates an output file which has been attributed with position and layout information for each node and edge. To be able to layout graphs defined by the graph hierarchy, we need to create a layout parser.

To implement a java-cup [14] based parser a LALR (Look-Ahead Left to Right) [15] grammar needs to be produced. This grammar must not be ambiguous. The terminal symbols need to be identified, as do the non-terminal symbols.

The parser uses a simple scanner generated by JLex [16] to read through the input file . As it encounters graph elements (nodes, edges, or sub-graphs), they are matched to the corresponding

IOPGraph element, the position, colour, size, etc attributes are updated and drawn as dot specifies. Nodes are simply moved to their correct position. Edges are drawn at render time, in the case of dot being the renderer, the edges are created as cubic spline curves.

Semantic actions need to augment the grammar. These semantic actions will define how graph elements will be handled. For our application as a graph renderer, the parser's semantic actions are going to be involved with setting graph, node and edge attributes. A semantic action which is not part of the grammar is the creation of the attribute list. The attribute list matches attribute names to attribute values for a graph element. It is used to extract the values of graph element attributes from the dot output file.

LALR Grammar For Dot with Semantic Actions

```
DotGraph    ::= OptStrict GraphType OptName LBRACE StmtList RBRACE
OptStrict   ::= STRICT | null
OptName     ::= ID | null
GraphType   ::= GRAPH | DIGRAPH
StmtList    ::= Stmt OptSemi StmtList| null
Stmt        ::= AttrStmt
                | NodeStmt
                | EdgeStmt
                | ID EQUALS ID
OptSemi     ::= SEMI | null
AttrStmt    ::= GRAPH AttrList
                semantic action: create graph if necessary,
                    set graph's width and height
                | NODE AttrList
                | EDGE AttrList
AttrList    ::= MakeAMap LSQUARE OptAList RSQUARE
OptAttrList ::= AttrList | null
MakeAMap    ::=  null
                semantic action: create hashmap
AList       ::= OptAList OptComma ID EQUALS ID
                semantic action: add element to hashmap
OptAList    ::= AList | null
OptComma    ::= COMMA | null
NodeStmt    ::= ID OptAttrList
                semantic action: create or move node,
                    setting attributes if necessary
EdgeStmt    ::= ID TO ID OptAttrList
                semantic action: draw edges,
                    setting attributes if necessary
```

The semantic actions need to be implemented as embedded java code within the java-cup parser implementation. These semantic actions will be added to the generated parser by java-cup, to provide the graph creation and layout capabilities of the parser.

4.4.1 Problems

There are some problems involved with dot being the graph renderer. If graph is dynamically created and after each addition to the graph the graph is re-layed-out the graph will change form, quite drastically in some cases. This will become particularly confusing to users when they are trying to analyse a graph. The graph will continually change form, and any benefit gained by using this software will be lost, since the graph does not maintain its layout.

Specifically positioning the nodes in the input file has no effect, as these values are ignored. Using subgraphs to try and fix the position of a portion of the graph has no effect either, as the layout algorithm dot uses must be global. Even breaking the graph into adjacently connected subgraphs, has no effect in maintaining the graphs intermittent layout. In dot, a nodes rank can be set relative to other node(s), this provides very limited methods to control the graph's layout. An edge's weight can be set, which will only have the effect of placing a node closer to another node

Another graph renderer, Da Vinci cannot be used to renderer our graph object, since it cannot return an attributed graph representation. It could be used as our graphics actor, but it would not provide the capabilities and features that we require.

4.5 Event Handling

This graph utilises the event system from `JLambda` and Glyphics, which enables dynamic creation of event handlers to pass events to the appropriate `Glyphish` instance. Events can be created and handed completely from within the `JLambda` code. Or the event can be handled by the glyph itself.

4.6 JLambda

The graph package was designed to be used with `JLambda`, as well as being used as normal objects. Just as `JLambda` can create any Java object at runtime, `JLambda` can create a GUI with toolbar, menubar, and containing graph object. Creation, manipulation, and event handling methods are all designed to be as simple as possible in order to minimise the number of methods which need to be invoked to achieve a goal.

Chapter 5

Applications

Although the `Glyphics` package was designed with a specific purpose in mind, it can be used in many applications. As well as the pathway logic actor, a few extra examples were produced to test the functionality of the graphics package during design. These extra examples not only show how to use the `Glyphics` package in the `JLambda` environment, but have helped in tracking bugs and other strange behaviour. This example will need to be run with the `JLambda` interpreter.

```
$ jlambda path/to/script.lsp
```

5.1 Pathway Logic Assistant

The Pathway Logic Assistant (PLA) [17] is the main application of the `Glyphics` package, and is being produced by Carolyn Talcott. PLA utilises the graph package and the component hierarchy to create an interactive graphical front end to the IOP and IMaude system to analyze biological reaction networks.

5.2 Sketchpad

The sketchpad [9] is a simple scribble program written in `JLambda`. The colour of the line can be changed by clicking on the drawing canvas, or with the button. The size of the line can be increased and decreased with the buttons. A small info pop-up containing the width of the line can be displayed by clicking the info button. All of these aspects makes it a good example to demonstrate the functionality of `JLambda` and the `Glyphics` package.

5.3 Affine Transform Demo

The affine transform demo [7] is written in `JLambda`. In this demo a TextGlyph is animated with a sequence of affine transforms. This demo can be used to test the affine transform interface of any new Glyphs, the demo can be altered to transform a simple shape, or an AnimatedGlyph Affine transforms can be combined to create compound transforms, with this range of manipulation Glyphs can be altered and moved easily around the screen.

Chapter 6

Conclusion

The Glyphics hierarchy is an effective way to produce high quality interactive graphics objects.

Glyphics is a clear improvement upon Joel Bartlett's Ezd. Glyphics proves a simple interface for producing graphics in the Java's Graphics2D environment. It also implements Java's new event system to easily create graphical objects which respond to events. Glyphics implements text and images to enable a wide range of graphical objects to be created and displayed.

Glyphics' transform API provides a fast and efficient way to manipulate Glyphics instances in double precision.

Glypyhics' components provide a simple and effective way to display and interact with Glyphics instances. These components have built in zooming capabilities, which improve the way large graphical objects are viewed by the user. The view automatically captures events, making it straight forward to add events to a Glyphish instance.

The applications in this thesis show a wide range of applications for the Glyphics hierarchy. And many more are possible, since the Glyphics hierarchy is easily extended to exhibit any behaviour that is required.

Bibliography

[1] Joel Bartlett. Ezd – easy-to-use structured graphics for Java. `http://research.compaq.com/wrl/projects/Ezd/home.html`.

[2] Ian A. Mason, David Porter, and Carolyn Talcott. The `JLambda` Language, 2004. `http://mcs.une.edu.au/~iop/Data/Papers/jlambda.pdf`.

[3] Java 1.4.2 API. Class AWTEvent. `http://java.sun.com/j2se/1.4.2/docs/api/java/awt/AWTEvent.html`.

[4] Deborah Adair, Jennifer Ball, and Monica Pawlan. 2D Graphics. `http://java.sun.com/docs/books/tutorial/2d/index.html`.

[5] Java 1.4.2 API. Class AffineTransform. `http://java.sun.com/j2se/1.4.2/docs/api/java/awt/geom/AffineTransform.h%tml`.

[6] `http://java.sun.com/products/java-media/2D/forDevelopers/java2dfaq.html`. The Java 2D FAQ.

[7] An animated affine transform demonstration, 2004. `http://mcs.une.edu.au/~iop/Data/JLambda/Misc/affineDemo-do.lsp`.

[8] A demonstration of `SubTextGlyph`, 2004. `http://mcs.une.edu.au/~iop/Data/JLambda/Glyph/text.lsp`.

[9] A simple sketchpad program, 2004. `http://mcs.une.edu.au/~iop/Data/JLambda/Misc/sketchpad.lsp`.

[10] A graph editing program, 2004. `http://mcs.une.edu.au/~iop/Data/JLambda/Graph/editGraph.lsp`.

[11] Emden Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with *dot*. `http://www.research.att.com/sw/tools/graphviz/dotguide.pdf`.

[12] A simple graph layout program, 2004. `http://mcs.une.edu.au/~iop/Data/JLambda/Graph/graph.lsp`.

[13] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. `http://www.graphviz.org/Documentation/TSE93.pdf`.

[14] Scott E. Hudson. CUP Parser Generator for Java. `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[15] Wikipedia. LALR parser. `http://en.wikipedia.org/wiki/LALR_parser`.

[16] Elliot Berk. JLex: A Lexical Analyzer Generator for Java(TM). `http://www.cs.princeton.edu/~appel/modern/java/JLex/`.

[17] I. A. Mason and C. L. Talcott. IOP: The InterOperability Platform & IMaude: An Interactive Extension of Maude. In *International Workshop on Rewriting Logic and its Applications (WRLA 2004)*, Electronic Notes in Theoretical Computer Science. Elsevier Science, 2004.