JVM Language and Compiler Design for Interactive Graphics

Nick Anderson
DGI 2016

November 3, 2016

1 Introduction

For Java, Scala, and other JVM-language programmers, programming for graphics proves far too difficult. One must learn the less intuitive JVM graphics libraries, interacting with drawing panels and frames, overriding classes to give simple functionality. For simple applications for Java beginners, the standard graphics libraries are simply out of reach without copious copying and pasting. Rather than approach graphics from a viewer's perspective, there is meta user research to be conducted about how others should be able to create graphics. Using a common tool of developers, a compiler, one may simplify interaction patterns with a new syntax that will also run on the JVM. A new language could be designed and implemented with a working compiler to better serve developers, especially those lacking experience.

2 Improving Standard Java Graphics

2.1 Current Java Libraries

As the most popular JVM language, Java has a Graphics API consisting of Frames and Canvases. The interface that a Java Graphics developer must work with has far too much depth for a beginner or any user merely trying to get a simple application running.

2.1.1 Canvas

Drawing takes place on a JPanel, which can then listen through the KeyListener interface. Several methods need to be understood and overrode to provide even the simplest interface. Rather than any simple import, developers must extends and implement a class rather than merely work with one. Many defaults must be set manually that new developers may expect. The canvas also lays upon a frame, which also must be created and demonstrates more barriers to building graphics applications quickly.

```
class SlaphicsCanvas extends JPanel implements KeyListener {

    Graphics2D g2d;
    Color paintColor;
    SlaphicsFrame frame;

    public SlaphicsCanvas() {
        super();
        this.setFocusable(true);
        this.addKeyListener(this);
    }

    public void setFrame(SlaphicsFrame frame) {
        this.frame = frame;
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        this.g2d = (Graphics2D) g;
    }

    @Override
    public void keyTyped(KeyEvent e) {

    }

    @Override
    public void keyPressed(KeyEvent e) {}

    @Override
    public void keyReleased(KeyEvent e) {}
}
```

2.1.2 Frame

Creating the frame for graphics presents a separation between canvas and frame, where many beginners would likely combine the two into a single drawing panel. Again, defaults such as the window closing behavior are set manually here, which would be expected by new developers.

```
class SlaphicsFrame extends Frame {

    public SlaphicsFrame() {
        super();
    }

    public void add(SlaphicsCanvas c) {
        super.add(c);
        c.setFrame(this);
        super.setVisible(true);
    }

    public SlaphicsFrame init(int width, int height) {
        super.setSize(width, height);
        super.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent windowEvent){
                System.exit(0);
            }
        });
        return this;
    }
}
```

2.2 Novel JVM Language

With a new JVM-compiled language, novice developers writing for the JVM could benefit from simpler interaction patterns. Implemented or extended classes should be kept to a minimum, only used to clarify the context of the program as within a graphics context. Creating a frame should be tightly coupled with the canvas, as in this example. The frame could merely have a set size and allow drawing with a set number of built in rectangles, ovals, strings, and lines.

The current mock up shows examples of the default paint through slacPaint. The artist then decides, on key press, to modify the original painted canvas with new lines, shapes, and new text. The interaction pattern with text is also simplified through discrete keySequence and keyEvent methods, where both are triggered in response to every key press, keyEvent receiving the last key, while keySequence receives every key pressed thus far.

```
class MySlaphicsCanvas <: SlaphicsCanvas {
   method slacPaint() : Unit = {
        self.draw(new SlaphicsString(100, 100, "Hello!", "black"));
        self.draw(new SlaphicsLine(25, 350, 350, 25, "orange"));
        self.draw(new SlaphicsRect(25, 25, 250, 200, "pink"));
        self.draw(new SlaphicsOval(250, 250, 75, 100, "gray"))
    }

    method slacPaintWithDiffColors() : Unit = {
        self.draw(new SlaphicsString(100, 100, "Hello in red!", "red"));
        self.draw(new SlaphicsLine(25, 350, 350, 25, "pink"));
        self.draw(new SlaphicsRect(25, 25, 250, 200, "black"));
        self.draw(new SlaphicsOval(250, 250, 75, 100, "blue"))
    }


    method keySequence(keyPressed : String) : Unit = {
        self.draw(new SlaphicsString(150, 150, "Pressed: " + keyPressed, "black"));
        self.slacPaintWithDiffColors()
    }

    method keyEvent(keyPressed : String) : Unit = {
        self.draw(new SlaphicsString(300, 300, "Pressed: " + keyPressed, "black"));
        self.slacPaintWithDiffColors()
    }
}

method main() : Unit = {
    var frame : SlaphicsFrame;
    frame = new SlaphicsFrame().init(500, 500);
    frame.add(new MySlaphicsCanvas())
}
```

3 Comparison to other Graphics API

3.1 PaperJS

From the PaperJS tutorial[Leh16], one defines the canvas on which to draw as the id of an HTML component, then has a natural interaction pattern of creating graphics primitives and interacting with them. Handlers can be set for key strokes and time events, using native javascript, naturally embedded. PaperJS demonstrates a more ideal interaction pattern, where the drawing surface is thrown anyway into the layout of a page, then primitives can be used more simply.

```
<!DOCTYPE html>
<html>
<head>
<!-- Load the Paper.js library -->
<script type="text/javascript" src="js/paper.js"></script>
<!-- Define inlined PaperScript associate it with myCanvas -->
<script type="text/paperscript" canvas="myCanvas">
        // Create a Paper.js Path to draw a line into it:
        var path = new Path();
        // Give the stroke a color
        path.strokeColor = 'black';
        var start = new Point(100, 100);
        // Move to start and draw a line from there
        path.moveTo(start);
        // Note the plus operator on Point objects.
        // PaperScript does that for us, and much more!
        path.lineTo(start + [ 100, -50 ]);
</script>
</head>
<body>
        <canvas id="myCanvas" resize></canvas>
</body>
</html>
```

4 Developer User Research

4.1 Developer Experience

The experience of a developer proves critical to his or her intuition regarding any API, where the current Java pattern may only prove unnatural for those used to other languages or just more natural, real world drawing experiences. Three distinct groups can quickly be identified: first, the most trivial, experienced developer who may quickly learn even less intuitive APIs, second, the inexperienced developer who has no intuitions regarding object-oriented programming or graphics programming, and third, the developer experienced in another language's graphics framework, such as Javascript's or Python's. However, for all three, more natural interaction patterns with a graphics library would best serve the beginner, feel familiar to the foreign developer, and prove unobtrusive to the experienced developer. Developers of all three groups should be asked for input on the implementation of a new JVM graphics library.

4.2 Feedback and Improvement

Developer feedback will inform improvements and changes to the new graphics API, as well as greater language design insights.

References

[Leh16] Jürg Lehni. Working with paper.js, 2016.