A Java/2D/3D Graphics System

- Application program
  - Written in Java
- Framework
  - AWT/Swing: Java’s windowing and user interface toolkit
  - Java2D: Java’s base 2D graphics library; built-in
  - Java Advanced Imaging: augments Java2D with image processing routines; add-on
  - Java3D: Scene-based 3D API; add-on
- Graphics engine
  - Frequently OpenGL under the hood, but not necessarily
- Operating system
  - Put your favorite Java-capable operating system here
- Device driver
  - Put your favorite nVidia, ATI, or other driver here
- Video hardware
  - Ditto, but for graphics cards
- Input devices
  - Keyboards, mice, trackballs, gloves, tablets, oh my!

Getting into Java Graphics

- Java is actually a general-purpose programming language, so when looking at it in terms of graphics, it exists at the same level as C and C++, not OpenGL.
- Like C and C++, Java is host to a number of graphics-related APIs. Unlike C and C++, these Java APIs are more uniform and standardized than their equivalents in C, C++, and other languages.
- Choose your weapon:
  - AWT — user interface framework that is a thin wrapper to the host operating system’s user interface facilities; generally not used these days
  - Swing — the portable (and official) Java user interface framework; features pluggable look-and-feels and more types of components
  - Java2D — Java’s 2D graphics API; co-exists with AWT and Swing, both “below” and “above” them
  - Java Advanced Imaging (JAI) — Java’s image processing plug-in API; builds image-specific functions on top of whatever is already in Java2D
  - Java3D — Java’s 3D graphics API, using a scene-based approach; recently went open source
- Other graphics-related Java APIs exist, both from Sun and not, such as:
  - Java Image I/O, Java Media Framework: additional Sun libraries
  - SWT, an alternative to the “official” AWT and Swing as a user interface framework; the Eclipse IDE is a very conspicuous user of SWT
  - jogl and GL4Java: more direct OpenGL wrappers for Java
  - and many more, including layers on top of Swing such as Buoy
Coding Details

• Graphics APIs in Java are just like any other Java library: they are typically defined by a package hierarchy
  – AWT: java.awt.*
  – Swing: javax.swing.* (also uses many AWT classes)
  – Java2D: java.awt.* also, more specifically subpackages java.awt.geom, java.awt.image
  – No strict boundaries among AWT, Swing, and Java2D because they’re all standard anyway — they’re always available in Java

• Other libraries are optional, but once installed are also just accessed by package
  – Java3D: javax.media.j3d.*

• Ditto for everything else, whether from Sun or not

Anatomy of Java graphics program

Starting up the event thread

The Java graphics “innards” are less explicit than in OpenGL/GLUT. Graphics routines take place in the context of the event thread, which in turn is activated whenever a window is displayed anywhere in the code.

Though the sample code below takes place in a main() method, in general windows can be opened anytime.*

```
public static void main(String[] args) {
    JFrame frame = new JFrame("Fireworks!");
    frame.setSize(500, 500);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setContentPane(new Fireworks());
    frame.show();
}
```

* There is actually a little loophole here that is related to the multithreaded nature of Java but the single-threaded nature of its core graphics routines... if you really want the gory details, start by reading “Threads and Swing” at http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html
Anatomy of Java graphics program

**Component classes: view**

```java
public class Fireworks extends JPanel {
    ...  
    /**
     * Overridden method: painting starts here.
     */
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        // Set up the controller.
        _timer = new Timer(90, new SparkMover());
        _timer.setRepeats(true);
        // Set up the mouse adapter.
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent mevt) {
                initFireworks();
            }
        });
    }  
    /**
     * The SparkMover class serves as the controller for the model; it is triggered
     * at fixed intervals by the timer to "advance" the model forward in time.
     */
    private class SparkMover implements ActionListener {
        public void actionPerformed(ActionEvent aevt) {
            for (int i = 0; i < _sparks.length; i++)
                _sparks[i].move();
            repaint();
        }
    }
    private Timer _timer;
    ...  
    public Fireworks() {
        ...  
        initFireworks();
    }
}
```

**Event listeners: controller**

```java
public class Fireworks extends JPanel {
    ...  
    /**
     * The javax.swing.Timer class is very similar in function to OpenGL's idleFunc(), with slightly
     * different semantics.
     */
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(Color.black);
        g.fillRect(0, 0, getWidth(), getHeight());
        private void paintSparks(Graphics g) {
            Graphics2D g2 = (Graphics2D)g.create();
            g2.translate(0, 0, getWidth(), getHeight());
            for (int i = 0; i < _sparks.length; i++)
                paintOneSpark(_sparks[i], g2);
        }
    }
    private void paintBackground(Graphics g) {
        g.setColor(Color.black);
        g.fillRect(0, 0, getWidth(), getHeight());
    }
    private void paintOneSpark(Spark s, Graphics g) {
        g.drawLine((int)s.getLocation().getX() - (radius / 2), (int)s.getLocation().getY() - (radius / 2), radius, radius);
        g.drawLine((int)s.getLocation().getX(), (int)s.getLocation().getY()),
            (int)(s.getLocation().getX()) + s.getVelocity().getX(), (int)(s.getLocation().getY() + s.getVelocity().getY()));
    }
    ...  
    public Fireworks() {
        ...  
        initFireworks();
    }
}
```

**Anatomy of Java graphics program**

If you are painting anything beyond standard user interface controls, start here.

Break up your painting code as you please; most of the time you need to pass the Graphics object around.

The Graphics class has tons of painting commands as well as OpenGL-like "state," properties. Some methods are only available from the more powerful Graphics2D class. Knock yourself out.

Anatomy of Java graphics program

As you probably already know, Java represents user activity as a set of listener/event methods, which you register with the appropriate object in order to be notified of desired user activity. This is equivalent to GLUT's *Func designation functions.

SparkMover, specifically its actionPerformed() method, is roughly the equivalent of the specific function that is designated as the GLUT idleFunc().
Anatomy of Java graphics program

Your code: the model and initialization

```java
import ...Spark;

public class Fireworks extends JPanel {
    
    public Fireworks() {
        // Create the model.
        _sparks = new Spark[1000];
        initFireworks();
    }
    ...

    private void initFireworks() {
        for (int i = 0; i < _sparks.length; i++)
            _sparks[i] = new Spark();
    }
    ...

    private Spark[] _sparks;
}
```

There are a million and one ways to define and implement your model. In this specific case, the role of model is divided between the Spark class and a subset of the Fireworks code. Spark defines an individual Spark, while Fireworks holds an array of 1000 Sparks.

You don’t really need a separate initialization function, but it’s good practice anyway. Sound familiar? Just like in OpenGL, model management and initialization is the part of a Java graphics program that is most dependent on your design skills.

Java3D Specifics

- Model: Java3D is based on a scene graph that you construct, starting with a VirtualUniverse and gradually branching out until you hit leaf nodes such as Shape3D. Viewing parameters and transforms are integrated into this scene graph.

- View: Java3D uses Canvas3D, which is essentially a customized AWT component. It takes a VirtualUniverse scene graph and renders it within its given space. In essence, Canvas3D replaces a Swing component with an overridden `paintComponent()`.

- Controller: Boils down to manipulation of the scene graph through the usual Java event listeners. Java3D provides additional classes for facilitating these manipulations.

- Performance considerations: A final side set of Java3D functions are geared toward optimizing the rendering and manipulation of the scene graph.
Look It Up

- Virtually everything you need to know about these libraries can be found online. A few useful addresses (but by no means the only ones):
  - Java tutorial: http://java.sun.com/docs/books/tutorial
  - Java API reference (current final version is 1.4.2): http://java.sun.com/j2se/1.4.2/docs/api/index.html
  - Java3D tutorial — PDF files to get you started: http://java.sun.com/products/java-media/3D/collateral/index.html#tutorial
  - Java3D home site — recently went open source: https://java3d.dev.java.net (note the https protocol)

- …and many many more. There is no shortage of Java documentation on the Web. Take full advantage of it.