Computer Graphics in Java

- 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, mainly because of its virtual machine abstraction

Choose your weapon (and this isn't an exhaustive list):

- 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 — Scene-based 3D graphics API
- Java Image I/O, Java Media Framework — libraries for handling image file formats and other media beyond still images, respectively
- SWT — a non-Sun alternative framework to AWT and Swing
- JOGL — the now-official OpenGL wrapper for Java (there also used to be GL4Java, but these days JOGL has pretty much taken over)

A Java Graphics System

- Application program
  - Written in Java
- Framework
  - One or more of those listed on the previous page
- Graphics engine
  - Frequently OpenGL under the hood, but not necessarily; typically whatever is available from the host operating system
- 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, etc. — anything for which the Java event manager can fire off Java events

As you can see, Java's architecture pretty much abstracts out individual hardware details (the bottom half of the graphics system)

Coding Details

- Make sure that the Java classes for the framework you are using are included in the Java classpath
  - AWT, Swing, and Java2D are built-in — no extra work or installs necessary
  - Other frameworks need additional files
    - *JAR (Java Archive) files* — stick these on your classpath, whether in the environment or IDE
    - *Native (JNI) libraries* — platform-specific "hook" files connecting Java to non-Java code (JOGL works this way)

- Once installed, refer to the frameworks in the usual Java way: import the needed classes, then instantiate objects or call methods as needed

- Troubleshooting — as always, the main idea is to determine the point of failure
  - *ClassNotFoundException*: Java can't find the class(es) your referring to — check your classpath
  - *UnsatisfiedLinkError*: a Java class has referenced native code that can't be found — check your JNI extension directories

Anatomy of a Java2D/Swing 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.*

Window initialization; instantiate as many as you need

Window classes such as JFrame have assorted properties that can be read/written via the standard getter/setter methods. Check out the API for details.

```
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();
}
```

JFrame has a content pane that defines "what's inside" — set this to get anything non-trivial.

Displays the window; initializes the whole graphics and event subsystem when invoked for the first time in a program.

*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

Component classes: view

For custom displays, start by overriding the *paintComponent()* method.

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

```
public class Fireworks extends JPanel {
    ...

    /**
     * Overridden method: painting starts here.
     */
    public void paintComponent(Graphics g) {
        super.paintComponent(g);

        paintBackground(g);
        paintSparks(g);
    }

    private void paintBackground(Graphics g) {
        g.setColor(Color.black);
        g.fillRect(0, 0, getWidth(), getHeight());
    }

    private void paintSparks(Graphics g) {
        Graphics2D g2 = (Graphics2D)g.create();
        g2.translate(getWidth() / 2.0, getHeight());
        for (int i = 0; i < _sparks.length; i++)
            paintOneSpark(_sparks[i], g2);
    }

   ...
}
```

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 a Java2D/Swing graphics program

Event listeners: controller

The javax.swing.Timer class triggers a designated action after some amount of real time passes. GLUT has a partial equivalent to this, which we didn't cover in class — *glutTimerFunc()*.

```
public class Fireworks extends JPanel {
    ...

    /**
     * Creates a new Fireworks panel.
     */
    public Fireworks() {
        ...

        // Set up the controller.
        _timer = new Timer(50, new SparkMover());
        _timer.setRepeats(true);

        // Set up the mouse adapter.
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent mevt) {
                initFireworks();
            }
        });
    }
```

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. Such listeners comprise a Java equivalent of sorts to GLUT's *Func designation functions.

```
    /**
     * 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;
}
```

SparkMover, specifically its *actionPerformed()* method, is roughly the equivalent of the specific function that is designated as the GLUT *glutTimerFunc()*. However, Swing's Timer class can repeat; *glutTimerFunc()* does not — you have to call it again once the designated time has elapsed.

Anatomy of a Java2D/Swing graphics program

Your code: the model and initialization

```
import ...Spark;

public class Fireworks extends JPanel {
    ...

    /**
     * Creates a new Fireworks panel.
     */
    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.