More Java Drawing in 2D

Animations with Timer

Drawing Review

- A simple two-dimensional coordinate system exists for each graphics context or drawing surface such as a JPanel
- Points on the coordinate system represent single pixels (no world coordinates exist)
- **Top left** corner of the area is coordinate <0, 0>
- A drawing surface has a width and height
  - example: `JPanel` has `getWidth()` and `getHeight()` methods
- Anything drawn outside of that area is not visible
  - You see no errors or exceptions either

JComponent

- To begin drawing we first need a class which extends `JComponent`. Use `JPanel`.
- Once we subclass `JPanel` we can override the `paintComponent()` method to specify what we want the panel to paint when repainting
- When painting we paint to a `Graphics` context (which is given to us as an argument to `paintComponent`)

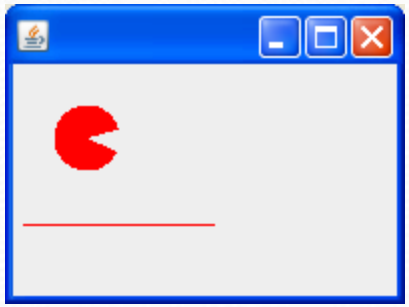
Extend JPanel

```
public class DrawingPanel extends JPanel {

  private Arc2D pieArc;

  public DrawingPanel() {
    pieArc = new Arc2D.Double(Arc2D.PIE);
    pieArc.setArc(20.0, 20.0, 33.0, 33.0, 15.0, 320.0, Arc2D.PIE);
  }

  @Override
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;
    g2.setPaint(Color.RED);
    g2.fill(pieArc);
    g2.drawLine(5, 80, 100, 80);
  }
}
```



paintComponent(Graphics g)

- `paintComponent()` is the method which is called when repainting a Component.
  - It should **never** be called explicitly, but instead `repaint()` should be invoked which will then call `paintComponent()` on the appropriate Components
- When overriding `paintComponent()` the first line should be `super.paintComponent(g)` which will clear the panel for drawing

Graphics vs. Graphics2D

- We are passed a `Graphics` object to `paintComponent()` which we paint to
- Cast the object passed to a `Graphics2D` which has much more functionality.
- `Graphics2D` has more sophisticated control over geometry, coordinate transformations, color management, and text layout. This is the fundamental class for rendering 2-D shapes, text and images on the Java(tm) platform
- Why didn't Java just pass `Graphics2D`?
  - Legacy support

Using Java Geometry

- We've seen Rectangle
- Classes within `java.awt.geom`

```
AffineTransform, Arc2D, Arc2D.Double, Arc2D.Float, Area,
CubicCurve2D, CubicCurve2D.Double, CubicCurve2D.Float,
Dimension2D, Ellipse2D,Ellipse2D.Double, Ellipse2D.Float,
FlatteningPathIterator, GeneralPath, Line2D,
Line2D.Double, Line2D.Float, Path2D, Path2D.Double,
Path2D.Float, Point2D, Point2D.Double, Point2D.Float,
QuadCurve2D, QuadCurve2D.Double, QuadCurve2D.Float,
Rectangle2D, Rectangle2D.Double, Rectangle2D.Float,
RectangularShape, RoundRectangle2D,
RoundRectangle2D.Double, RoundRectangle2D.Float
```

Using Java Geometry (cont.)

- Note: `Arc2D.Double` means that `Double` is an inner class contained in `Arc2D`

```
pieArc = new Arc2D.Double(Arc2D.PIE);
pieArc.setArc(20.0, 20.0, 33.0, 33.0, 15.0, 320.0, Arc2D.PIE);
```

- The difference between `g2.draw()` and `g2.fill()` is that draw will draw an outline of the object while fill will fill in the object

Alpha (Transparency)

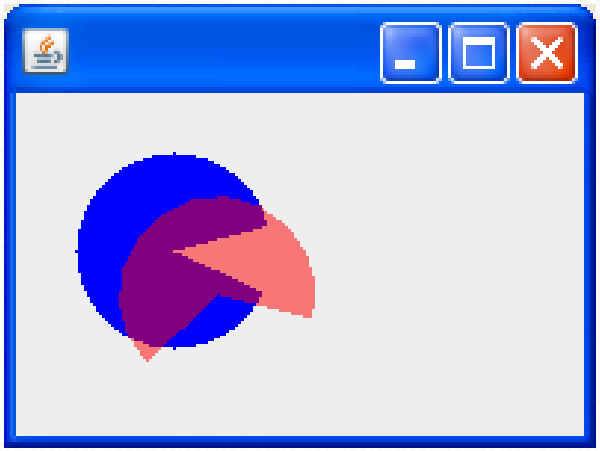
- We can assign transparency (alpha) values to drawing operations so that the underlying graphics partially shows through when you draw shapes or images.
- Create an `AlphaComposite` object using `AlphaComposite.getInstance` with a mixing rule
- There are 12 built-in mixing rules but we only care about `AlphaComposite.SRC_OVER`
  - See `AlphaComposite` API for more details
- Alpha values range from 0.0f to 1.0f, completely transparent and completely opaque respectively
- Pass the `AlphaComposite` object to `g2.setComposite()` so that it will be used in our rendering

Alpha (Transparency, cont.)

```
pieArc = new Arc2D.Double(Arc2D.PIE);
pieArc.setArc(20.0, 20.0, 66.0, 66.0, 15.0, 320.0, Arc2D.PIE);
pieArcRed = new Arc2D.Double(Arc2D.PIE);
pieArcRed.setArc(35.0, 35.0, 66.0, 66.0, -15.0, 240, Arc2D.PIE);
 // ....
g2.setPaint(Color.BLUE);
g2.fill(pieArc);
Composite originalComposite = g2.getComposite();
AlphaComposite alphaComposite =     // 0.1f would make red faint
      AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 0.5f);
g2.setComposite(alphaComposite);

g2.setPaint(Color.RED);
g2.fill(pieArcRed);
g2.setComposite(originalComposite);


// Run animation.DrawPanelMain
```



Animation with `javax.swing.Timer`

- At its simplest, animation is the time-based alteration of graphical objects through different states, locations, sizes and orientations
- This code set a timer to send an `actionPerformed` message to its listener(s) every 10 milliseconds

```
timer = new javax.swing.Timer(10, new TimerListener());
timer.start();
```

- The code in the `actionPerformed` message of class `TimerListener` that will be executed ever 10 ms

```
public class TimerListener implements ActionListener {
  public void actionPerformed(ActionEvent evt) {
   // Executes every so many ms the Timer has been set to
   // from a separate thread that it created for itself
  }
}
```

Animation: Simple Timer Example

```
public class RedBallPanel extends JPanel {
  private static Random generator;
  private Ellipse2D pellet;
  private int tick;
  private int x, y;
  private Timer timer;

  public RedBallPanel() {
    generator = new Random();
    startAnimating();
  }

  @Override public void
  paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;
    g2.setPaint(Color.RED);
    g2.fill(pellet);
  }

  private void startAnimating() {
    pellet = new Ellipse2D.Double();
    tick = 0;
    timer = new javax.swing.Timer(1000, new TimerListener());
    timer.start();
  }
```

executes every 1000 ms

```
public class TimerListener
        implements ActionListener {

  public void actionPerformed(
    x = generator.nextInt(Main.FR..
    y = generator.nextInt(Main.FR..
    pellet.setFrame(x, y, 10, 10);
      repaint();
      tick++;
      if (tick > 20)
        timer.stop();
    }
  }
}
```

Animation: Double Buffering

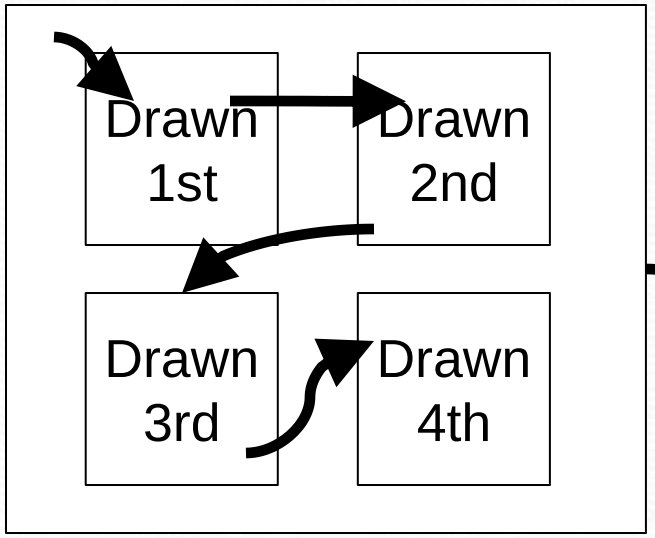
- Double buffering is the mechanism of using a second bitmap (or buffer) which receives all of the updates to a window during an update.
- Once all of the objects of a window have been drawn, then the bitmap is copied to the primary bitmap seen on the screen.
- This prevents the flashing from appearing to the user that occurs when multiple items are being drawn to the screen.

Animation: Double Buffering In Swing

- Swing now (as of Java 6) provides true double buffering. Previously, application programmers had to implement double buffering themselves
- The no-arg constructor of `JPanel` returns a `JPanel` with a double buffer (and a flow layout) by default
  - "...uses additional memory space to achieve fast, flicker-free updates."

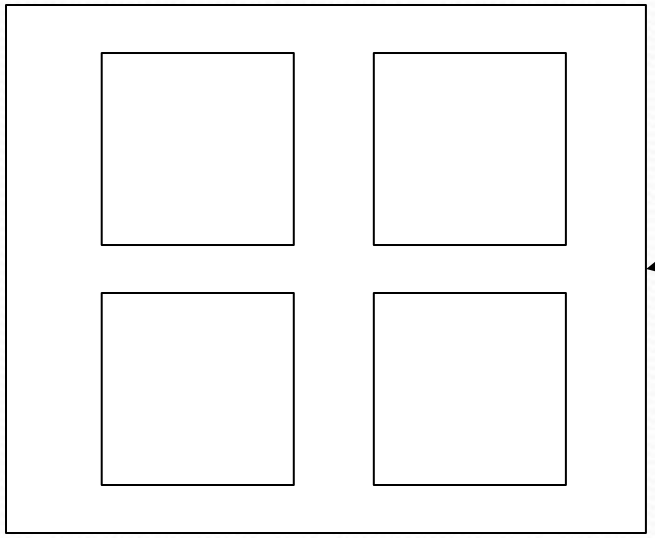
Animation: Double Buffering In Swing

Backing Bitmap
(not visible to user)



Each box drawn
in order might be
detectable by user.

Window Bitmap
(visible to user)



The entire backing
bitmap is copied at
once to the visible
bitmap, avoiding any
flicker.

Animated GIF's

- Using animated GIF's in your projects is okay, but there is a better way to animate things
- Problems with animated GIF's:
  - The timings between frames are set in stone and cannot be controlled by the programmer.
  - During the first animation of the GIF it will appear that your image is not loaded since the GIF loads each frame as it is displayed (looks ugly, but can be avoided using image loaders)
  - Programmers also can't control which frame to display or start/stop on since the GIF just keeps animating
- Instead of Animated GIF's, most 2D games will use what is called a **SpriteSheet** and then animate using a `Timer` or `Thread`

Sprite Sheets

- A sprite sheet is a depiction of various sprites arranged in one image, detailing the exact frames of animation for each character or object by way of layout.
- The programmer can then control how fast frames switch and which frame to display. Additionally the whole sheet is loaded at once as a single image and won't cause a loading glitch.
- In Java we can use the `BufferedImage` method:

```
getSubimage(int x, int y, int w, int h)
```

  - Returns a subimage defined by a specified rectangular region.