March 6, 2000

Java 2D Graphics, Solid Color Fill

Java Programming, Lecture Notes # 312

by Richard G. Baldwin
baldwin@austin.cc.tx.us

Introduction

In an earlier lesson, I explained that the **Graphics2D** class extends the **Graphics** class to provide more sophisticated control over geometry, coordinate transformations, color management, and text layout. Beginning with JDK 1.2, **Graphics2D** is the fundamental class for rendering two-dimensional shapes, text and images.

I also explained that without understanding the behavior of other classes and interfaces such as **Shape**, **AffineTransform**, **GraphicsConfiguration**, **PathIterator**, and **Stroke**, it is not possible to fully understand the inner workings of the **Graphics2D** class.

What has been covered previously?

Earlier lessons have explained a number of Java 2D concepts, including **Shape**, **AffineTransform**, and **PathIterator**. I am saving **GraphicsConfiguration** until later because it is fairly complicated. Before, I can explain the **Stroke** class, I need to explain how to fill a **Shape**. The purpose of this lesson, and the next couple of lessons is to help you to understand how the *fill* process works in Java 2D.

Lessons build on one another

As you can see, each of the lessons in this series on Java 2D builds on the knowledge that you have gained by studying previous lessons. For that reason, I hope that you are taking the time to study the lessons in sequence.

Methods of the Graphics2D Class

The **Graphics2D** class has dozens of methods. In this lesson, I will be primarily concerned with the following three methods:

- setPaint(Paint paint)
- fill(Shape s)
- draw(Shape s)

I have used the **draw()** method in numerous previous lessons, so it isn't new to this lesson and won't merit much discussion. However, **setPaint()** and **fill()** are new to this lesson. I will discuss them both in detail

The setPaint(Paint paint) method

Here is part of what Sun has to say about the **setPaint()** method:

> Sets the **Paint** attribute for the **Graphics2D** context. Calling this method with a null **Paint** object does not have any effect on the current **Paint** attribute of this **Graphics2D**.
>
> Parameters:
> paint - the **Paint** object to be used to generate color during the rendering process, or null

The terminology here can be a little confusing, especially with respect to the use of the word *paint.*

First, the class named **Graphics2D** has a property named **paint**. The method **setPaint()** is a typical *setter* method used to set the **paint** property.

The parameter that is passed to this method must be a reference to an object of a class that implements the interface named **Paint**.

The Paint Interface

Here is what Sun has to say about the **Paint** interface.

> "This **Paint** interface defines how color patterns can be generated for **Graphics2D** operations. A class implementing the **Paint** interface is added to the **Graphics2D** context in order to define the color pattern used by the **draw** and **fill** methods."

This interface declares a single method named **createContext()** that returns a reference to an object of a class that implements the **PaintContext** interface. (I tell my Java students at least four or five times during each semester that if they don't understand the Java interface, they really don't understand Java.)

The PaintContext Interface

Here is what Sun has to say about the **PaintContext** interface.

> The **PaintContext** interface defines the encapsulated and optimized environment to generate color patterns in device space for **fill** or **stroke** operations on a **Graphics2D**. The **PaintContext** provides the necessary colors for **Graphics2D** operations in the form of a **Raster** associated with a **ColorModel**. The **PaintContext** maintains state for a particular paint operation. In a multi-threaded environment, several contexts can exist simultaneously for a single **Paint** object.

Obviously, I could continue tracking the path through the hierarchy, but that won't be necessary.

The Good News and the Bad News

The bad news is that this can all be very complicated. The good news is that unless you intend to define a class that implements the **Paint** interface, you don't need to be concerned about many of these details.

(See a later reference to Knudsen's book for an example of how to define your own class to implement the **Paint** interface.)

All you really need to know is the following:

> To set the **paint** property of a **Graphics2D** object, all that you must do is invoke the **setPaint()** method on that object, passing a reference to an object instantiated from a class that implements the **Paint** interface as a parameter.

Fortunately, there are several useful classes in the 2D API that implement the **Paint** interface so you may not need to define your own class that implements **Paint**.

The fill(Shape s) method

Why would you want to set the **paint** property of the **Graphics2D** object in the first place?

Because, when you invoke the **fill()** method, passing a reference to a **Shape** object to that method, the **Shape** object will be filled using the **Paint** object that was previously established by invoking the **setPaint()** method.

The bottom line

The bottom line is, if you want to *fill* a **Shape** object before you draw it, you accomplish this with the following two steps:

1. Invoke **setPaint()** on the **Graphics2D** object, passing a reference to an object of a class that implements the **Paint** interface as a parameter.
2. Invoke the **fill()** method on the **Graphics2D** object, passing a reference to the **Shape** object that you want to fill as a parameter.

As I mentioned in an earlier lesson, in his book, Java Foundation Classes in a Nutshell, David Flanagan tells us that the Java 2D definition of a **Shape** does not require the shape to enclose an area. In other words, a **Shape** object can represent an open curve. According to Flanagan, if an open curve is passed to a method that requires a closed curve (such as **fill()**), the curve is automatically closed by connecting its end points with a straight line.

The Three Paint Classes

The Java2D API provides at least three classes that implement the **Paint** interface:

- **Color**
- **GradientPaint**
- **TexturePaint**

(As of JDK 1.2.2, this is apparently all of the classes in the API that implement **Paint**, but you can always define your own.)

The Color class

The **Color** class can be used to fill a **Shape** object with a solid color. This will be the topic of the remainder of this lesson.

The GradientPaint class

The **GradientPaint** class can be used to fill a **Shape** with a color gradient. The gradient progresses from one specified color at one point to a different specified color at a different point. The two colors can be stabilized beyond the two points *(acyclic)* or the gradient can be caused to repeat in a cyclic fashion beyond the two points *(cyclic)*.

In his book entitled Java 2D Graphics, Jonathan Knudsen provides a sample program that produces a radial color gradient. This is a good example program to take a look at if you need to define your own class that implements the **Paint** interface.

Use of the **GradientPaint** class will be the topic of a subsequent lesson.

The TexturePaint class

The **TexturePaint** class can be used to fill a **Shape** with a tiled version of a **BufferedImage** object. This will also be the topic of a subsequent lesson.

Sample Program

The name of this program is **PaintColor01**. It illustrates the use of a **Paint** object to fill a Shape with a solid color. In this case, the **Paint** object is an instance of the **Color** class, which implements the interface named **Paint**.

The GUI is a Frame object

The program draws a four-inch by four-inch **Frame** on the screen. It translates the origin to the center of the **Frame**. Then it draws a pair of X and Y-axes centered on the new origin. So far, this is very similar to the sample programs that I have explained in previous lessons.

A circle in each quadrant

The program then draws one two-inch diameter circle in each quadrant. It fills the upper left circle with solid red, the upper right circle with solid green, the lower left circle with solid blue, and the lower right circle with solid yellow

The program was tested using JDK 1.2.2 under WinNT Workstation 4.0

This discussion of dimensions in inches on the screen depends on the method named **getScreenResolution()** returning the correct value. However, the **getScreenResolution()** method always seems to return 120 on my computer regardless of the actual screen resolution settings.

Will discuss in fragments

As is often the case, I will discuss this program in fragments. The controlling class and the constructor for the GUI class are essentially the same as you have seen in several previous

lessons, so, I won't bore you by repeating that discussion here. You can view that material in the complete listing of the program at the end of the lesson.

All of the interesting action takes place in the overridden **paint()** method, so I will begin the discussion there.

Overridden paint() method

The beginning portions of the overridden **paint()** method should be familiar to you by now as well. So, I am simply going to let the comments in Figure 1 speak for themselves.

```
//Override the paint() method
 public void paint(Graphics g){
  //Downcast the Graphics object to a
  // Graphics2D object
  Graphics2D g2 = (Graphics2D)g;

  //Scale device space to produce inches on
  // the screen
  // based on actual screen resolution.
  g2.scale((double)res/72,(double)res/72);

  //Translate origin to center of Frame
  g2.translate((hSize/2)*ds,(vSize/2)*ds);

  //Draw x-axis
  g2.draw(new Line2D.Double(
                      -1.5*ds,0.0,1.5*ds,0.0));
  //Draw y-axis
  g2.draw(new Line2D.Double(
                      0.0,-1.5*ds,0.0,1.5*ds));
```

Figure 1

The interesting part

That brings us to the interesting part, which, if you understand the previous discussion, you will find to be very straightforward.

The code in the next four fragments draws a circle in the upper left quadrant of the **Frame** and fills the circle with the color red.

An object that implements the Shape interface

The code in Figure 2 instantiates an object of the **Ellipse2D.Double** class. This is one of several classes in the API that can be used to produce geometric objects that implement the **Shape** interface.

```
  Ellipse2D.Double circle1 =
            new Ellipse2D.Double(
```

```
                -2.0*ds,-2.0*ds,2.0*ds,2.0*ds);
```

Figure 2

Recall that the parameters required for the constructor of this class specify a bounding rectangle for the ellipse. If that bounding rectangle describes a square, the ellipse turns into a circle.

Assuming that the variable **ds** contains the actual screen resolution, the parameters used in this fragment describe a bounding rectangle (square) that is two inches one each side, and whose upper left-hand corner is positioned two inches above and two inches to the left of the origin. This will produce a circle with a two-inch diameter, located in the upper left-hand quadrant of the **Frame.**

An object that implements the Paint interface

Figure 3 instantiates a new object of the class **Color**, initialized to the color red, and passes it to the **setPaint()** method of the **Graphics2D** object. (Recall that the **Color** class implements the **Paint** interface, so this satisfies the type requirements of the parameter to the **setPaint()** method.)

```
  g2.setPaint(new Color(255,0,0));//red
```

Figure 3

Filling the upper left-hand circle

Figure 4 invokes the **fill()** method on the **Graphics2D** object, passing the circle in the upper left-hand quadrant as a parameter. This causes the circle to be filled using the **Color** object discussed in the previous fragment. This, in turn, causes the circle to be filled with the color red.

```
  g2.fill(circle1);
```

Figure 4

It's time to render the circle

At this point, the red circle has not yet been rendered onto the screen. That happens in the Figure 5, which invokes the **draw()** method on the **Graphics2D** object, passing a reference to the red circle as a parameter.

```
  g2.draw(circle1);
```

Figure 5

When the circle is rendered, those attributes that have previously been established in the **Graphics2D** object (such as scaling, translation, rotation, etc.) will be used to render the circle.

Just for the record, recall that the **draw()** method requires a parameter that is a reference to an object of a class that implements the **Shape** interface. The **Ellipse2D.Double** class, of which this circle is an instance, is one of the geometric classes in the API that implements the **Shape** interface.

It's all downhill from here

Once you understand all of the above, the remainder of the program, shown in Figure 6, is completely straightforward.

```
//Upper right quadrant, Solid green fill
Ellipse2D.Double circle2 =
        new Ellipse2D.Double(
          0.0*ds,-2.0*ds,2.0*ds,2.0*ds);
g2.setPaint(new Color(0,255,0));//green
g2.fill(circle2);
g2.draw(circle2);

//Lower left quadrant, Solid blue fill
Ellipse2D.Double circle3 =
        new Ellipse2D.Double(
          -2.0*ds,0.0*ds,2.0*ds,2.0*ds);
g2.setPaint(new Color(0,0,255));//blue
g2.fill(circle3);
g2.draw(circle3);

//Lower right quadrant, Solid yellow fill
Ellipse2D.Double circle4 =
        new Ellipse2D.Double(
          0.0*ds,0.0*ds,2.0*ds,2.0*ds);
//yellow
g2.setPaint(new Color(255,255,0));
g2.fill(circle4);
g2.draw(circle4);
```

Figure 6

This remaining code simply creates, fills, and draws three more circles in the remaining three quadrants of the **Frame**. Except for variable names and parameter values, this code is the same as that shown in the previous several fragments, so I will let the comments speak for themselves.

You can view a complete listing of the program at the end of the lesson.

Summary

I have explained the basics of how *fill* is handled in Java 2D, and have shown you how to accomplish solid fill.

I will show you how to fill a **Shape** object with a color gradient, or with a **BufferedImage** object in subsequent lessons.

Complete Program Listing

A complete listing of the program is provided in Figure 7.

```
/*PaintColor01.java 12/12/99
 Copyright 1999, R.G.Baldwin

 Illustrates use of a Paint object to fill a Shape with
 a solid color.

 Draws a 4-inch by 4-inch Frame on the screen.

 Translates the origin to the center of the Frame.

 Draws a pair of X and Y-axes centered on the
 new origin.

 Draws one 2-inch diameter circle in each quadrant.

 Fills upper left circle with solid red.
 Fills upper right circle with solid green
 Fills lower left circle with solid blue
 Fills lower right circle with solid yellow

 Whether the dimensions in inches come out right or not
 depends on whether the method getScreenResolution()
 returns the correct resolution for your screen.

 Tested using JDK 1.2.2 under WinNT Workstation 4.0
**************************************************/
import java.awt.geom.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;

class PaintColor01{
  publicstaticvoid main(String[] args){
    GUI guiObj = new GUI();
  }//end main
}//end controlling class PaintColor01

class GUI extends Frame{
  int res;//store screen resolution here
  staticfinalint ds = 72;//default scale, 72 units/inch
  staticfinalint hSize = 4;//horizonal size = 4 inches
  staticfinalint vSize = 4;//vertical size = 4 inches

  GUI(){//constructor
    //Get screen resolution
    res = Toolkit.getDefaultToolkit().
                                getScreenResolution();
    //Set Frame size
    this.setSize(hSize*res,vSize*res);
    this.setVisible(true);
    this.setTitle("Copyright 1999, R.G.Baldwin");

    //Window listener to terminate program.
    this.addWindowListener(new WindowAdapter(){
      public void windowClosing(WindowEvent e){
        System.exit(0);}});
  }//end constructor
  //-------------------------------------------------------//
```

```
 //Override the paint() method
 public void paint(Graphics g){
  //Downcast the Graphics object to a Graphics2D
  // object
  Graphics2D g2 = (Graphics2D)g;

  //Scale device space to produce inches on the
  //screen based on actual screen resolution.
  g2.scale((double)res/72,(double)res/72);

  //Translate origin to center of Frame
  g2.translate((hSize/2)*ds,(vSize/2)*ds);

  //Draw x-axis
  g2.draw(new Line2D.Double(
                            -1.5*ds,0.0,1.5*ds,0.0));
  //Draw y-axis
  g2.draw(new Line2D.Double(
                             0.0,-1.5*ds,0.0,1.5*ds));

  //Upper left quadrant, Solid red fill
  Ellipse2D.Double circle1 = new Ellipse2D.Double(
                    -2.0*ds,-2.0*ds,2.0*ds,2.0*ds);
  g2.setPaint(new Color(255,0,0));//red
  g2.fill(circle1);
  g2.draw(circle1);

  //Upper right quadrant, Solid green fill
  Ellipse2D.Double circle2 = new Ellipse2D.Double(
                     0.0*ds,-2.0*ds,2.0*ds,2.0*ds);
  g2.setPaint(new Color(0,255,0));//green
  g2.fill(circle2);
  g2.draw(circle2);

  //Lower left quadrant, Solid blue fill
  Ellipse2D.Double circle3 = new Ellipse2D.Double(
                     -2.0*ds,0.0*ds,2.0*ds,2.0*ds);
  g2.setPaint(new Color(0,0,255));//blue
  g2.fill(circle3);
  g2.draw(circle3);

  //Lower right quadrant, Solid yellow fill
  Ellipse2D.Double circle4 = new Ellipse2D.Double(
                      0.0*ds,0.0*ds,2.0*ds,2.0*ds);
  g2.setPaint(new Color(255,255,0));//yellow
  g2.fill(circle4);
  g2.draw(circle4);

 }//end overridden paint()

}//end class GUI
//================================//
```

Figure 7

About the author

Richard Baldwin *is a college professor and private consultant whose primary focus is a combination of Java and XML. In addition to the many platform-independent benefits of Java*

applications, he believes that a combination of Java and XML will become the primary driving force in the delivery of structured information on the Web.

Richard has participated in numerous consulting projects involving Java, XML, or a combination of the two. He frequently provides onsite Java and/or XML training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Java Programming Tutorials, which has gained a worldwide following among experienced and aspiring Java programmers. He has also published articles on Java Programming in Java Pro magazine.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

baldwin@austin.cc.tx.us

-end-