Java 2D Graphics, Texture Fill

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

Java Programming, Lecture Notes # 316

March 19, 2000

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**. Before, I can explain the **Stroke** class, I need to explain how to fill a **Shape**. Earlier lessons showed you how to fill a **Shape** with a solid color and with a color gradient.

How to fill, in general

I explained in an earlier lesson that if you want to *fill* a **Shape** object before you draw it, you can accomplish this with the following two steps:

- Invoke **setPaint()** on the **Graphics2D** object, passing a reference to an object of a class that implements the **Paint** interface as a parameter.

- Invoke the **fill()** method on the **Graphics2D** object, passing a reference to the **Shape** object that you want to fill as a parameter.

This lesson will show you how to fill a **Shape** with a tiled version of an image, otherwise known as a texture.

The Three Paint Classes

In a previous lesson, I explained that the Java2D API in JDK 1.2.2 provides three classes that implement the **Paint** interface (and you can also define your own):

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

The Color class

The **Color** class can be used to fill a **Shape** object with a solid color. That was the topic of an earlier lesson.

The GradientPaint class

The **GradientPaint** class can be used to fill a **Shape** with a color gradient. That also was the topic of an earlier lesson.

The TexturePaint class

The **TexturePaint** class can be used to fill a **Shape** with a tiled version of a **BufferedImage** object. That is the topic of this lesson.

At this point, we need to learn a little more about the class named **TexturePaint**. Here is what Sun has to say about the class.

"The **TexturePaint** class provides a way to fill a **Shape** with a texture that is specified as a **BufferedImage**. The size of the **BufferedImage** object should be small because the **BufferedImage** data is copied by the **TexturePaint** object.

At construction time, the texture is anchored to the upper left corner of a

Rectangle2D that is specified in user space. Texture is computed for locations in the device space by conceptually replicating the specified **Rectangle2D** infinitely in all directions in user space and mapping the **BufferedImage** to each replicated **Rectangle2D**."

So, we see immediately that we need to know something about the **BufferedImage** class.

The BufferedImage Class

The **BufferedImage** class extends the **Image** class in JDK 1.2.Here is what Sun has to say about **BufferedImage**.

"The **BufferedImage** subclass describes an **Image** with an accessible buffer of image data. A **BufferedImage** is comprised of a **ColorModel** and a **Raster** of image data. The number and types of bands in the **SampleModel** of the **Raster** must match the number and types required by the **ColorModel** to represent its color and alpha components. All **BufferedImage** objects have an upper left corner coordinate of (0, 0). Any **Raster** used to construct a **BufferedImage** must therefore have minX=0 and minY=0."

All in all, this looks pretty complicated. Fortunately, we don't need to be too concerned about many of the technical details.

Getting a BufferedImage

There are a couple of ways that you can get a **BufferedImage** to use to fill your **Shape**. One way is to read a file containing an image. I plan to cover that approach in subsequent lessons that deal with image processing.

Another approach is to invoke the **createImage()** method on any object that extends the **Component** class. This will return a reference to an object of type **Image,** which can be downcast to **BufferedImage**. This is the approach used in this lesson,

The createImage() method

Here is what Sun has to say about the **createImage()** method.

public Image **createImage**(
  int width,
  int height)

Creates an off-screen drawable image to be used for double buffering.

Parameters:
  width - the specified width.
  height - the specified height.

Returns: an off-screen drawable image, which can be used for double buffering.

As mentioned above, the **BufferedImage** class extends the **Image** class in JDK 1.2.

You can cast the Image to a BufferedImage

Here is what Java Foundation Classes in a Nutshell, by David Flanagan, has to say about the **createImage()** method.

"This method was first introduced in Java 1.0; it returns an **Image** object. In Java 1.2, however, the returned **Image** object is always an instance of **BufferedImage** so you can safely cast it.

After you have created an empty **BufferedImage** you can call its **createGraphics()** method to obtain a **Graphics2D** object.
...
Anything you can draw on the screen, you can draw into a **BufferedImage**."

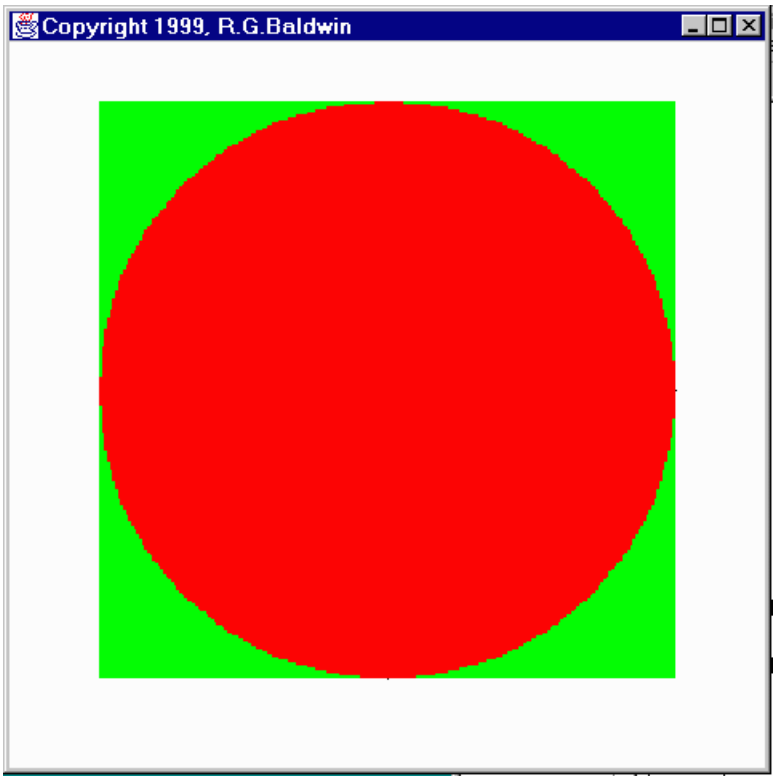
The first sample program shows how to create and display a **BufferedImage** object.

The second program shows how to create a **BufferedImage** program and use it to fill a **Shape** object.

Sample Program BufferedImage01

This program shows how to create and display a **BufferedImage** object without a requirement for an external file containing an image.

A significantly reduced screen shot of the output of this program is shown below.



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.

Then the program creates a **BufferedImage** object that is three inches on each side. It gets a **Graphics2D** context on the **BufferedImage**.

Draw a green rectangle on the BufferedImage

Once the **Graphics2D** context is available, the program draws a rectangle on the context the same size as the **BufferedImage**. Then it fills the rectangle with the solid color green.

Draw a red circle on the BufferedImage

Then the program draws a filled circle on the context that just fits inside the dimensions of the **BufferedImage** object. The circle is filled with the solid color red. The circle covers the green rectangle leaving some green exposed around the edges.

Draw the BufferedImage on the Frame

Finally, the program draws the **BufferedImage** object on the **Frame**, centered on the origin.

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

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 repeat 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 going to let the comments in the Figure 1 speak for themselves.

```
publicvoid 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

Get a BufferedImage object

Figure 2 invokes the **createImage()** method on the **Frame** (this) to get a reference to an object of type **Image**. The **Image** is a square, three inches on each side.

```
double size = 3.0;//size of BufferedImage object
BufferedImage bufImg = (BufferedImage)
              this.createImage((int)(size*ds),
                               (int)(size*ds));
```

Figure 2

This reference is immediately downcast to **BufferedImage** and stored in a reference variable of type **BufferedImage** named **bufImg**.

Get a Graphics2D context on the BufferedImage

Figure 3 invokes the **createGraphics()** method on the **BufferedImage** to get a **Graphics2D** context on the **BufferedImage**.

```
Graphics2D g2dImage =
                    bufImg.createGraphics();
```

Figure 3

Once I have the **Graphics2D** context, I can draw pictures on the **BufferedImage** just as though I am drawing on the screen.

Draw a green square on the BufferedImage

Figure 4 uses familiar code to draw a square on the **BufferedImage**. The square is the same size as the **BufferedImage**, and is filled with solid green color.

```
Rectangle2D.Double theRectangle =
    new Rectangle2D.Double(
                       0.0,0.0,size*ds,size*ds);
g2dImage.setPaint(new Color(0,255,0));//green
g2dImage.fill(theRectangle);
```

```
g2dImage.draw(theRectangle);
```

Figure 4

Draw a red circle on the BufferedImage

Continuing with familiar code, Figure 5 draws a filled red circle on the **BufferedImage**. This circle covers the green square giving us a green square containing a red filled circle.

```
Ellipse2D.Double theCircle =
            new Ellipse2D.Double(
                        0.0,0.0,size*ds,size*ds);
g2dImage.setPaint(new Color(255,0,0));//red
g2dImage.fill(theCircle);
g2dImage.draw(theCircle);
```

Figure 5

Render the BufferedImage on the Frame

It is important to understand that up to this point, I have simply been creating a virtual image in the computer's memory. Nothing has been done so far to cause this image to be visible to a human observer.

One common use of **BufferedImage** is to build virtual images in memory and to blast them to the screen as quickly as possible to produce smooth animation. This can sometimes eliminate the flicker that often occurs when we attempt to produce animation by drawing directly on the screen.

In this case, all I want to do is to see the image that I have constructed in the computer's memory. I accomplish that in Figure 6, which invokes the **drawImage()** method of the **Graphics2D** class to render the **BufferedImage** on the screen.

```
g2.drawImage(bufImg,null,(int)(-(size/2)*ds),
                         (int)(-(size/2)*ds));
```

Figure 6

A red filled circle on a green square

If you compile and run this program, you should see a filled red circle on a green square. The outermost edges of the circle coincide with the edges of the square. If the actual resolution of your screen is given by **ds**, the size of the square should be three inches on each side.

How does this differ from previous programs

Although it may not be obvious from the visual presentation, this is significantly different from the previous programs that have drawn circles on the screen and filled them.

This program did not draw a circle on the screen. Rather, the circle was drawn and filled in the computer's memory and transferred to the screen in the final rendering process.

Why go to all of this trouble

This does seem like a hard way to get a filled red circle on the screen. However, the purpose of this program was not to display a filled red circle on the screen. The purpose was to show you how to create a **BufferedImage** under program control and to draw something on that **BufferedImage**.

The rendering of the **BufferedImage** on the screen was simply a little icing on the cake.

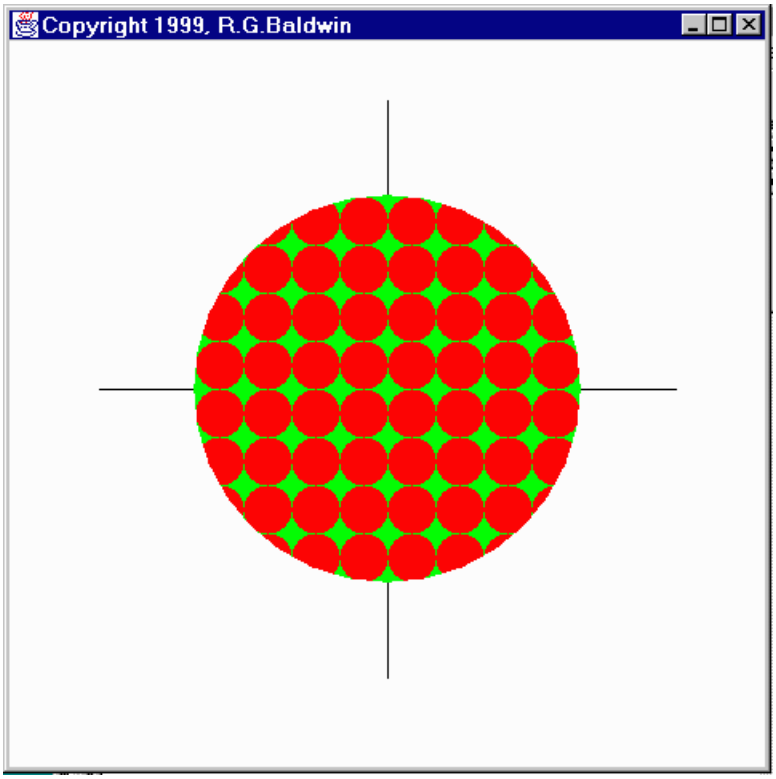
The next sample program will need a **BufferedImage** object, and you now know how to create one without the need for an external file containing an image.

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

Sample Program PaintTexture01

This program illustrates the use of a **TexturePaint** object to fill a relatively large circle with small tiled instances of a **BufferedImage** object.

A significantly reduced screen shot of the output of this program is shown below.



The **BufferedImage** object has a red filled circle on a green square the same as in the previous program.

The program uses code to create the **BufferedImage** object to avoid the need to provide an auxiliary image file to accompany this lesson.

As usual, 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.

Then it fills a two-inch diameter circle with small tiled versions of the **BufferedImage** object described above.

Finally, the program draws the filled two-inch diameter circle on the **Frame**, centered on the origin.

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

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 repeat 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 going to let the comments in Figure 7 speak for themselves.

```
publicvoid 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 7

The main Shape object

Figure 8 instantiates the main **Shape** object. This is a circle, two inches in diameter. This circle will be filled with a tile pattern constructed from the **BufferedImage** object, which is a red ball on a green square background.

```
   Ellipse2D.Double theMainCircle =
            new Ellipse2D.Double(
              -1.0*ds,-1.0*ds,2.0*ds,2.0*ds);
```

Figure 8

The interesting part

That brings us to the interesting part. Figure 9 declares and initializes a **double** variable named **tileSize** that will be used to establish the size of the tiles, in inches, used to fill the circle.

```
double tileSize = 0.25;//size of tiles in the fill

Rectangle2D.Double anchor =
 new Rectangle2D.Double(
                          0,0,(int)(tileSize*ds),
                             (int)(tileSize*ds));
```

Figure 9

Controlling the size of the tiles

This variable is used to instantiate a new **Rectangle2D.Double** object that is a square, 0.25 inches on each side. As we will see later, the **BufferedImage** object will be automatically scaled to fit this rectangle, and a large number of such rectangles will be used to fill the circle.

The TexturePaint class

At this point, we need to learn a little more about the **TexturePaint** class. This is what Sun has to say about the constructor for this class.

public **TexturePaint**(
BufferedImage txtr,
Rectangle2D anchor)

Constructs a TexturePaint object.

Parameters:

- txtr - the BufferedImage object with the texture used for painting
- anchor - the Rectangle2D in user space used to anchor and

replicate the texture

As you can see, the constructor takes two parameters. The first parameter is a reference to a **BufferedImage** object that will be used to produce the tiles to fill the **Shape**.

The second parameter is a rectangle that is used to anchor and replicate the texture. Sun has this to say about the actual process of filling the **Shape.**

"Texture is computed for locations in the device space by conceptually replicating the specified **Rectangle2D** infinitely in all directions in user space and mapping the **BufferedImage** to each replicated **Rectangle2D**."

The bottom line

The bottom line is that the circle is filled with rectangles of the specified size, and the **BufferedImage** is scaled down and drawn automatically on each rectangle.

Originally, I created a small **BufferedImage** objectthat was the same size as the rectangle. When I did that, I got a very ragged looking circle in the **BufferedImage**. I found that the results were much better when I created a fairly large **BufferedImage** and allowed it to be scaled down automatically to fit the rectangle.

Instantiate a TexturePaint object

Figure 10 instantiates the new **TexturePaint** object, passing the **BufferedImage** and the anchor rectangle to the constructor as parameters.

```
TexturePaint theTexturePaintObj =
      new TexturePaint(
                  getBufferedImage(),anchor);
```

Figure 10

The **BufferedImage** is actually produced by invoking a method named **getBufferedImage()** that I will discuss shortly.

There is nothing special about the **getBufferedImage()** method. It is simply a convenience method that I wrote to separate the code that generates the **BufferedImage** from the code that uses the **BufferedImage**.

Filling and drawing the circle

Figure 11 shows the three statements that you have come to expect for filling and drawing **Shape** objects.

```
//set fill object
g2.setPaint(theTexturePaintObj);
g2.fill(theMainCircle);//do the fill
g2.draw(theMainCircle);//draw the filled circle
```

Figure 11

Compile and run the program

If you compile and run this program, you should see a two-inch diameter circle centered on a **Frame**. The circle is filled with small green squares, and each green square contains a red filled circle.

The getBufferedImage() method

Figure 12 contains the entire method named **getBufferedImage()**.

```
BufferedImage getBufferedImage(){
  //Larger images produce better quality.
  double imageSize = 10.0;
  BufferedImage theBufferedImage =
           (BufferedImage)this.createImage(
                        (int)(imageSize*ds),
                       (int)(imageSize*ds));

  //Get a Graphics2D context on
  // the BufferedImage object
  Graphics2D g2dImage =
  theBufferedImage.createGraphics();

  //Draw a rectangle on the
  // Graphics2d context on the
  // BufferedImage and fill it with
  // the color green.
  Rectangle2D.Double theRectangle =
                     new Rectangle2D.Double(
                         0.0,0.0,imageSize*ds,
                                 imageSize*ds);
  g2dImage.setPaint(new Color(0,255,0));
  g2dImage.fill(theRectangle);
  g2dImage.draw(theRectangle);

  //Draw a circle on the Graphics2d
  // context on the BufferedImage and
  // fill it with the color red.
```

```
 // This circle will cover the green rectangle.
 Ellipse2D.Double circleOntheBufferedImage
                    =  new Ellipse2D.Double(
                        0.0,0.0,imageSize*ds,
                               imageSize*ds);
 //red
 g2dImage.setPaint(new Color(255,0,0));
 g2dImage.fill(circleOntheBufferedImage);
 g2dImage.draw(circleOntheBufferedImage);

 return theBufferedImage;

  }//end getBuffered Image

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

Figure 12

As mentioned earlier, this is a convenience method that I wrote to separate the code that creates the **BufferedImage** from the code that uses it. The method creates and returns a **BufferedImage** object consisting of a filled red circle on a green square background.

This is essentially the same code as in the previous program, except that this time, instead of rendering the **BufferedImage** on the screen, the method simply returns a reference to the **BufferedImage** object. Since there is nothing new in the fragment, I won't discuss it further.

Summary

In this lesson, I have shown you how to create a **BufferedImage** object under program control without the requirement for an external image file.

I have also shown you how to use a **BufferedImage** object to fill a **Shape** with a texture produced by tiling the **BufferedImage**.

Complete Program Listings

A complete listing of both programs is provided in Figure 13 and Figure 14.

```
/*BufferedImage01.java 12/12/99
 Copyright 1999, R.G.Baldwin

 Illustrates:
 1. Creating a BufferedImage object from code as
    an alternative to importing from an image file.
 2. Drawing geometric shapes on the
    BufferedImage object.
 3. Drawing the BufferedImage object on a
    Frame object.
```

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

Translates the origin to the center of the Frame.

Creates a BufferedImage object 3.0 inches on
each side.

Gets a Graphics2D context on the BufferedImage

Draws a green filled rectangle on the context
that is the same size as the BufferedImage object.

Draws a red filled circle on the context that just fits
inside the dimensions of the BufferedImage object.
The circle covers the green rectangle.

Draws the BufferedImage object on the Frame,
centered on the origin.

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, WinNT Workstation 4.0
**********/

```
import java.awt.geom.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;

class BufferedImage01{
  publicstaticvoid main(String[] args){
    GUI guiObj = new GUI();
  }//end main
}//end controlling class BufferedImage01

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(){
      publicvoid windowClosing(WindowEvent e){
        System.exit(0);}});
  }//end constructor
  //-------------------------------------------------------//

  //Override the paint() method
  publicvoid 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));

    double size = 3.0;//size of BufferedImage object
    //Get a BufferedImage object
    BufferedImage bufImg =
          (BufferedImage)this.createImage(
                                  (int)(size*ds),
                                  (int)(size*ds));

    //Get a Graphics2D context on the
    //  BufferedImage object
    Graphics2D g2dImage =
                              bufImg.createGraphics();

    //Draw a rectangle on the BufferedImage and
    // fill it with the color green.
    Rectangle2D.Double theRectangle =
          new Rectangle2D.Double(
                               0.0,0.0,size*ds,size*ds);
    g2dImage.setPaint(new Color(0,255,0));//green
    g2dImage.fill(theRectangle);
    g2dImage.draw(theRectangle);

    //Draw a circle on the BufferedImage and fill
    //  it with the color red.
    Ellipse2D.Double theCircle =
           new Ellipse2D.Double(
                               0.0,0.0,size*ds,size*ds);
    g2dImage.setPaint(new Color(255,0,0));//red
    g2dImage.fill(theCircle);
    g2dImage.draw(theCircle);

    //Now draw the BufferedImage on the Frame
    g2.drawImage(bufImg,null,(int)(-(size/2)*ds),
                               (int)(-(size/2)*ds));

  }//end overridden paint()

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

Figure 13

```
/*PaintTexture01.java 12/12/99
 Copyright 1999, R.G.Baldwin

 Illustrates use of a TexturePaint object to fill a
 relatively large circle with small tiled instances of a
 BufferedImage object.

 The BufferedImage object has a red filled circle on
 a square green background.

 Uses code to create the BufferedImage object to
 avoid the need to provide an auxiliary image file.

 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.

Fills a 2-inch diameter circle with small tiled
versions of the BufferedImage object.

Draws the filled 2-inch diameter circle on the
Frame, centered on the origin.

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, WinNT Workstation 4.0
**********************************************/
import java.awt.geom.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;

class PaintTexture01{
  publicstaticvoid main(String[] args){
    GUI guiObj = new GUI();
  }//end main
}//end controlling class PaintTexture01

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(){
      publicvoid windowClosing(WindowEvent e){
        System.exit(0);}});
  }//end constructor
  //-------------------------------------------------------//

  //Override the paint() method
  publicvoid 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));

    //Instantiate the main Shape object which is a
    //  circle
    Ellipse2D.Double theMainCircle =
      new Ellipse2D.Double(
                        -1.0*ds,-1.0*ds,2.0*ds,2.0*ds);

    double tileSize = 0.25;//size of tiles in the fill

    //Instantiate the anchor rectangle. This
    // determines the size of the tiles containing the
    // BufferedImage when the circle is filled.
    Rectangle2D.Double anchor =
           new Rectangle2D.Double(
                                  0,0,(int)(tileSize*ds),
                                      (int)(tileSize*ds));
    //Instantiate the TexturePaint object and
    // populate it with a BufferedImage object.
    TexturePaint theTexturePaintObj =
      new TexturePaint(getBufferedImage(),anchor);

    g2.setPaint(theTexturePaintObj);//set fill object
    g2.fill(theMainCircle);//do the fill
    g2.draw(theMainCircle);//draw the filled circle

  }//end overridden paint()
  //-------------------------------------------------------//

  //Method to create and return a BufferedImage
  //  object
  //Returns a BufferedImage consisting of a filled
  //  red circle on a green square background.
  BufferedImage getBufferedImage(){
    //Larger images produce better quality.
    double imageSize = 10.0;
    BufferedImage theBufferedImage =
      (BufferedImage)this.createImage(
                              (int)(imageSize*ds),
                                (int)(imageSize*ds));

    //Get a Graphics2D context on the
    //  BufferedImage object
    Graphics2D g2dImage =
                  theBufferedImage.createGraphics();

    //Draw a rectangle on the Graphics2d context
    // on the BufferedImage and fill it with the color
    // green.
    Rectangle2D.Double theRectangle =
             new Rectangle2D.Double(
                                    0.0,0.0,imageSize*ds,
                                            imageSize*ds);
    g2dImage.setPaint(new Color(0,255,0));
    g2dImage.fill(theRectangle);
    g2dImage.draw(theRectangle);

    //Draw a circle on the Graphics2d context on the
    // BufferedImage and fill it with the color red.
    // This circle will cover the green rectangle.
    Ellipse2D.Double circleOntheBufferedImage =
              new Ellipse2D.Double(
                                  0.0,0.0,imageSize*ds,
                                          imageSize*ds);
    g2dImage.setPaint(new Color(255,0,0));//red
    g2dImage.fill(circleOntheBufferedImage);
    g2dImage.draw(circleOntheBufferedImage);

    return theBufferedImage;
```

```
  }//end getBuffered Image

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

Figure 14

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.

-end-