February 9, 2000

Java 2D Graphics, Simple Affine Transforms

Java Programming, Lecture Notes # 306

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.

This lesson is intended to give you the necessary understanding of the **AffineTransform** class.

What is an Affine Transform?

A system of device-independent coordinates (called *User Space*) is used to pass all coordinate information to the methods of a **Graphics2D** object. An **AffineTransform** object (see definition below) is contained in the **Graphics2D** object as part of its state. This **AffineTransform** object defines how to convert coordinates from user space to device-dependent coordinates in *Device Space.*

According to Sun:

> The **AffineTransform** class represents a 2D Affine transform that performs a linear mapping from 2D coordinates to other 2D coordinates that preserves the "straightness" and "parallelness" of lines. Affine transformations can be constructed using sequences of translations, scales, flips, rotations, and shears.

According to David Flanagan in his book Java Foundation Classes in a Nutshell,

> The coordinate system transformation described by **AffineTransform** have two very important properties:
>
> - Straight lines remain straight
> - Parallel lines remain parallel

Flanagan goes on to tell us:

> An **AffineTransform** is a linear transform, so the transformation can be expressed in the matrix notation of linear algebra. An arbitrary **AffineTransform** can be mathematically expressed by six numbers in a matrix like this:
>
> ```
>  sx shx tx
> shy  sy ty
> ```

(If you are a purist, you will recognize immediately that this matrix is missing the large square brackets [ ] that normally enclose a matrix. That is because I don't know how to create large square brackets in HTML.)

Don't Panic!

Please don't panic at the mention of matrix algebra. You don't need to know anything about matrix algebra to understand the material in this lesson. All that you need to know is a little about ordinary algebra.

In fact, you don't even have to understand ordinary algebra to implement the kinds of simple transforms that I am going to show you in this lesson, but such an understanding will help you to better understand how it all works.

What Do These Symbols Represent?

In this lesson, I am going to show you how to use the **AffineTransform** to cause your graphics to be transformed in the following ways before being displayed on the output device:

- Scaling
- Translation
- Shear
- Rotation

In the notation shown above, the symbols or factors **sx** and **sy** are scale factors that accomplish scaling. The factors **tx** and **ty** are scale factors that accomplish translation. The factors **shx** and **shy** are scale factors that accomplish shear.

According to Flanagan, "...rotation is a combination of scaling and shearing, so there are not separate **rx** and **ry** numbers."

How Are They Used?

"So what?" you say. "So what if I have six numbers?" How do they accomplish scaling, translation, and shear?

With regard to these three, all that you need to remember is the following pair of equations, where the asterisk (*) indicates multiplication just as it does in Java programming syntax. (Actually, you don't even need to remember these equations, but it is helpful in understanding what happens when we modify the default **AffineTransform** object.)

```
newX = sx*x + shx*y + tx
newY = shy*x + sy*y + ty
```

In other words, given the values for an **x** and **y** coordinate in user space, these multiplicative factors are used calculate new values for those coordinates for use in device space, thereby accomplishing scaling, shear, and translation.

As we will see later, if the factors representing shear and translation have a value of zero, then that type of transform is simply not performed. If the factors representing scaling have a value of unity (which is the default case), then there is no change in the coordinate value between user space and device space. Any other values for any of the factors will cause some amount of scaling, shear, or translation to take place.

For all three types of transform, the value used to transform the **x** coordinate is independent of the value used to transform the **y** coordinate. Therefore, you could, for example, translate by a large value in **x** and translate by a small value in **y**.

Before getting into the details of performing these transforms, it will probably be useful to provide a few words of explanation about each.

Scaling

Scaling is perhaps the easiest of the four types of transforms to understand. This simply means that if a point is located at a horizontal coordinate value of **x** in user space, it will be located at **sx*x** in device space, where **sx** is a numeric multiplier that can be either positive or negative.

Translation

The purpose of translation is to move the origin of the coordinate system in device space.

For example, the default position of the origin is the upper left-hand corner of the component on which the graphic is being displayed. Assume that the component is a **Frame** object that is four inches on each side. You might like for the origin to be in the center of the **Frame** instead of at the top left-hand corner. You could accomplish this by translating the origin by two inches in both the horizontal and vertical directions.

Or, you might like for the origin to be just barely inside the borders of the **Frame** object instead of outside the borders as is the default. This can be accomplished by getting the widths of the top border and left border by invoking **getInsets()** on the **Frame**, and then using those values to translate the origin to a location that is just barely inside the borders.

Another use of translation (in combination with scaling) is to flip the default positive direction of the vertical axis so that increasing positive values go up instead of down, which is the default. I will leave the implementation of this as an exercise for the student.

Shear

I like the way that Jonathan Knudsen describes shear in his book entitled Java 2D Graphics. He describes it something like this. Take a stack of paper (like you might place in your laser printer) and draw a picture on the side of the stack. Then deform the stack so that the side of the stack on which you drew the picture takes the form of a parallelogram. Be especially careful to keep the opposite sides parallel. Then look at your picture. It will have been subjected to shear in one dimension.

Rotation

Rotation is also fairly easy to visualize (but the combination of rotation and shear can be very difficult to visualize). To visualize rotation, draw a picture on a piece of paper. Use a single tack to attach it to a bulletin board. Then rotate the paper a few degrees around the tack. You

will have rotated the picture around the coordinates of the tack by the specified number of degrees.

Java 2D works the same way. In the process of transforming a graphic from user space to device space, you can cause it to be rotated by a specified angle around a specified coordinate position.

However, in Java 2D, the angle must be specified in radians instead of degrees. If you are unfamiliar with the use of radian measure to specify an angle, just remember the following identity:

PI radians = 180 degrees

Where PI is the numeric value that you learned about in your geometry class.

PI has a value of 3.14159..... However, you don't have to remember the value of PI. It is available as a static final constant in the **Math** class. You can access the value as **Math.PI**.

If this sounds confusing, you might also want to remember the following identities:

PI/2 = 90 degrees
PI/4 = 45 degrees
PI/8 = 22.5 degrees
PI/16 = 11.25 degrees, etc.

This last identify is used in the sample program in the following section to rotate a simple graphic by 11.25 degrees.

Another interesting thing about the rotation transform is that positive angles of rotation proceed in a clockwise direction. This may be backwards from what you are accustomed to in various science and engineering endeavors where counter-clockwise rotation is often thought of as the positive direction of rotation.

Sample Program

With that introduction, lets look at some code and learn just how simple it is to accomplish Scaling, Translation, Shear, and Rotation with Java 2D.

The name of the sample program is **Affine01.java**. This program illustrates the use of convenience method of the **Graphics2D** class that make it very easy to apply transforms of the following types:

- Scaling
- Translation
- Shear
- Rotation

The methods that I will be discussing here are simply convenience methods, and there are other, more general, ways to apply Affine transforms as well. If you understand the process in sufficient detail, you can create your own matrix of six numeric values and cause those six values to be used to implement the transform. But, that is a topic for another lesson. In this lesson, I will concentrate on the convenience methods.

You can use these convenience methods to apply transform of the four types listed above with no concern for the six numeric values in the transform matrix. However, for your education (and mine as well), I have written the program such that when a transform is applied using a convenience method, the six values are displayed on the command-line screen. That way, we can both see what a particular transform produces in terms of the transform matrix (if we care about such things).

Experimentation is encouraged

This program is designed for experimentation. Four individual statements similar to the following are used to apply each of the four types of transforms (as you can see, this is the statement that applies translation):

```
g2.translate(0.25*ds,0.25*ds);
```

You should experiment with these statements by disabling them, enabling them, and rearranging them, and then observing the graphic output when you run the program.

Cumulative but not commutative

You will find that the result of executing two or more of the statements in succession is *cumulative*, but not *commutative*. In other words, applying translation and rotation, for example, causes the output graphic to be translated and also rotated. However, depending on your parameters, you might not get the same results if you switch the order of translating and rotating.

The six numeric values

Finally, before getting into the details of the code, I am going to show you the six numeric values contained in the default Affine transform and the six values contained in the transform after applying each of the four types of transforms in succession.

```
Default Transform
1.0 0.0 0.0
0.0 1.0 0.0

Add Scale Transform
1.6666666666666667 0.0 0.0
0.0 1.6666666666666667 0.0

Add Translate Transform
1.6666666666666667 0.0 30.0
```

```
0.0 1.6666666666666667 30.0

Add Shear Transform
1.6666666666666667 0.08333333333333334 30.0
0.16666666666666669 1.666666666666666730.0

Add Rotate Transform
1.6508996608400615 -0.2434184299932779
79.32270275806317
0.4886147500940855 1.6021270803360292 -
7.066823581936546
```

The default values

As you can see, the default transform has non-zero values only in the two scale factors for a scaling transform, and these scale factors are both unity. Thus, the default transform doesn't apply any scaling between user space and device space.

The scaling transform

After invoking the **scale()** methodto compensate for the difference in the default coordinate values (at 72 units per inch) and the actual resolution of my screen at 120 pixels per inch, the scale factor increased to 1.666... This is the ratio of 120 to 72.

See the caveat in an earlier lesson that the **getScreenResolution()** method always seems to return 120 on my computer regardless of the actual screen resolution settings.

The translation

After I applied a translation of 0.25 inches in both dimensions, each of the translation scale factors changed from zero to 30.0. This is the product of 0.25 and the resolution of my screen.

The shear

Then I applied a shear of 0.05 inches in **x** and 0.1 inches in **y**. The two shear factors changed from zero to 0.08333... and 0.1666... respectively. If you are interested, I will leave it as an exercise for you to go back to the equations that I presented earlier to deduce how these factors produce the desired amount of shear.

The rotation

Finally, adding rotation by 11.25 degrees caused all six values to change from their previous values. As mentioned earlier, this is probably the most complex aspect of the standard

translations insofar as understanding what the numbers mean. You will need some understanding of trigonometry to understand these values.

Don't worry about the six numeric values

Again, as I mentioned earlier, you can use the convenience methods to apply the four standard types of transforms without ever giving any thought to these numeric values. They are presented here only for the benefit of those students who have a desire to go further and to produce custom transforms from a numeric basis.

Finally, Some Code!

After all of that, I am finally going to show you some code.

As usual, I will discuss the program in code fragments. A complete listing of the program is provided at the end of the lesson.

The first fragment (Figure 1) instantiates an object of a class named **GUI** and causes it to be displayed on the screen. This code is very similar to code that I have presented in earlier lessons, so I won't discuss it in detail.

```
/*Affine01.java 12/12/99
Copyright 1999, R.G.Baldwin

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

class Affine01{
  publicstaticvoid main(String[] args){
    GUI guiObj = new GUI();
  }//end main
}//end controlling class Affine01

class GUI extends Frame{
  int res;//store screen resolution here
  staticfinalint ds = 72;//default scale, 72 units/inch

  GUI(){//constructor
    //Get screen resolution
    res = Toolkit.getDefaultToolkit().
                        getScreenResolution();

    //Set Frame size to four-inches by four-inches
    this.setSize(4*res,4*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
```

Figure 1

Basically, this code causes a **Frame** object that is four inches on each side to be displayed on the screen. Beyond this, all of the action in the program is provided by an overridden **paint()** method, which I will discuss shortly.

Overridden paint() method

Figure 2 shows the beginning of the overridden **paint()** method. As mentioned in an earlier lesson, the **paint()** method always receives a reference to an object of the **Graphics** class. It is necessary to downcast this reference to the **Graphics2D** class in order to gain access to the methods of the **Graphics2D** class (**Graphics2D** extends **Graphics**).

```
publicvoid paint(Graphics g){
  Graphics2D g2 = (Graphics2D)g;
```

Figure 2

Displaying the six numeric values

Figure 3 displays the words "**Default Transform"** on the screen and then invokes a method named **displayMatrix()** to cause the six values of the default **AffineTransform** object to be displayed. I will discuss the method named **displayMatrix()** later.

```
  System.out.println("Default Transform");
  displayMatrix(g2.getTransform());
```

Figure 3

What does the program really do?

Fundamentally, this program makes four updates to the default **AffineTransform** object to apply scaling, translation, shear, and rotation to the transform. Then the program creates and draws a graphic inside the **Frame** object. When the graphic is drawn, it is transformed according to the current state of the **AffineTransform** object.

As indicated earlier, you are encouraged to disable, enable, and rearrange these four transform update operations, recompile, and run the program in order to get a good feel for the effect of the different types of transforms.

The default case with no transforms enabled

For example, if you disable all four transform updates and run the program, you should see five concentric squares located in the upper left-hand quadrant of the **Frame** object. (Can squares be concentric? Maybe nested squares would be a better description.)

The dimensions of the outer square should be 72 pixels on each side. Of course, this will produce different size squares, depending on the resolution of your screen in terms of pixels per inch. If your screen actually has 72 pixels per inch, the outer square will be one inch on each side.

The text string "**Exit->**" should be superimposed on the squares with the bottom left-hand corner of the letter "**E"** being located at the center of the innermost square.

Enable the scaling transform

If you then enable the scaling update shown in Figure 4 (while leaving the other three updates disabled) recompile, and rerun the program, you should see the same five concentric squares, but this time they should be centered in the four-inch by four-inch **Frame** object.

```
System.out.println("Add Scale Transform");
g2.scale((double)res/72,(double)res/72);
displayMatrix(g2.getTransform());
```

Figure 4

The outer square should be one inch on each side, and the distance between the squares should be 0.1 inch.

This should be true regardless of the resolution of your screen because the parameters being passed to the **scale()** method are designed to compensate for the difference between actual screen resolution and the default resolution of 72 pixels per inch.

The convenience method named **scale()** makes it possible to provide scaling between user space and device space without having to be concerned as to how the scaling is actually accomplished from a matrix algebra viewpoint. The same is true for each of the other four convenience methods that I will discuss.

Note that you have independent control over the scaling to be applied to the horizontal axis and the scaling to be applied to the vertical axis. Each of these is a different parameter that you pass to the **scale()** method.

Enable the translation transform

If you leave the above scaling enabled and then enable the translation update shown in Figure 5 (with shear and rotation still disabled), you should see that the entire pattern of concentric squares and the text has been translated to the right and down by 0.25 inch.

```
System.out.println("Add Translate Transform");
g2.translate(0.25*ds,0.25*ds);
displayMatrix(g2.getTransform());
```

Figure 5

Although it isn't obvious, what has actually happened is that the origin has been translated from the upper left-hand corner of the **Frame** object to a location that is 0.25 inches to the right and 0.25 inches below the upper left-hand corner of the **Frame** object.

Then when the squares are drawn with respect to the new origin, they appear to have moved in a similar fashion relative to the outer boundaries of the **Frame** object. It is important to keep in mind, however, that the squares are still being drawn in the same position relative to the origin. It is the origin that has moved relative to the outer boundaries of the **Frame**.

Enable the shear transform

Next, I recommend that you disable translation, enable the shear update shown in Figure 6, and leave scaling enabled. If you recompile and run the program in that configuration, you should see something like Mr. Knudsen's shear example except that in this case, it occurs in both the horizontal and vertical dimensions.

```
System.out.println("Add Shear Transform");
g2.shear(0.05,0.1);
displayMatrix(g2.getTransform());
```

Figure 6

For example, the squares have all been converted to parallelograms. The bottom ends of what used to be vertical lines are now further to the right than the top ends of the same lines. The right ends of what used to be horizontal lines are now lower than the left ends of the same lines. The shapes of the characters in the text string are modified accordingly.

This picture also confirms what Mr. Flanagan has to say about Affine transforms:

- Straight lines remain straight
- Parallel lines remain parallel

You might also want to modify the values of the parameters being passed to the **shear()** method and observe the output. Be careful and don't make the values too large. If you do, you will push the entire graphic out of the **Frame** and won't be able to see it.

A bad case of the "jaggies"

Depending on your screen resolution, you may have noticed that the lines in the graphic output have a bad case of the "jaggies." In other words, the lines are not smooth, but rather have a stair step appearance. This is more properly referred to *aliasing*. A subsequent lesson will explain how to use the *anti-aliasing* feature of Java 2D to reduce the undesirable visual effect of this phenomenon.

Enable the rotation transform

Next, I recommend that you disable the shear update and enable the rotation update shown in Figure 7, while keeping the scaling update enabled. For me, at least, it is very difficult to separate the visual results of combined shear and rotation, so it is better to view them separately.

```
System.out.println("Add Rotate Transform");
//Rotate 11.25 degrees about center
g2.rotate(Math.PI/16,2.0*ds, 2.0*ds);
displayMatrix(g2.getTransform());
```

Figure 7

Now you should see the same five concentric squares plus the superimposed text string, rotated clockwise by 11.25 degrees around the center of the innermost square.

As mentioned earlier, you have the ability to specify the angle of rotation in radians as well as the **x** and **y** coordinate values of the point around which the rotation will take place.

These are only suggestions

The scenarios that I have been suggesting are simply that, suggestions. At this point, you might want to go back and experiment further with different combinations of transform types. Also, don't forget to experiment with the order in which the transforms take place because the same transforms executed in a different order will often produce different results.

Drawing the picture

None of the code discussed so far has actually caused anything to be displayed on the screen (except for perhaps an empty **Frame** object). Rather, that code was executed for the purpose of establishing the behavior of the transform that is applied to any graphic that is subsequently drawn in the **Frame**.

Figure 8 causes five concentric squares to be drawn in the **Frame** object. If the scaling transform described above is applied (with no shear or translation), the outer-most square will be one-inch on each side. The squares will be separated by 0.1 inch, and the entire group of squares will be centered at a coordinate position of two inches horizontally and two inches vertically.

```
double delta = 0.1;
for(int cnt = 0; cnt < 5; cnt++){
  g2.draw(new Rectangle2D.Double(
     (1.5+cnt*delta)*ds, (1.5+cnt*delta)*ds,
     (1.0-cnt*2*delta)*ds, (1.0-cnt*2*delta)*ds));
}//end for loop
```

Figure 8

The application of different scaling factors, the application of shear, the application of translation, or the application of rotation will cause different results to be produced.

Figure 9 superimposes the text string "**Exit->**" on the squares with the bottom left-hand corner of the letter "**E**" located at the center of the innermost square. I will have a lot more to say about how **Graphics2D** handles text in subsequent lessons. The main reason I put it here is to provide orientation information to help you keep track of what's going on if you implement large rotation angles.

```
  g2.drawString("Exit ->",2.0f*ds,2.0f*ds);
 }//end overridden paint()
```

Figure 9

That ends the definition of the overridden **paint()** method.

Displaying the six numeric values

Figure 10 shows the beginning of the definition of a convenience method that I wrote for the purpose of displaying the six values stored in an **AffineTransform** object.

```
void displayMatrix(
                AffineTransform theTransform){
 double[] theMatrix = newdouble[6];
 theTransform.getMatrix(theMatrix);
```

Figure 10

This method receives a reference to an **AffineTransform** object, on which it invokes the **getMatrix()** method.

When the **getMatrix()** method is invoked, a reference to a **double** array object having at least six elements is passed as a parameter. The **getMatrix()** method places the six values of interest in the first six elements of that array in a particular order and then returns.

I'm not going to go into the order in which the six values are placed in the array. I will simply refer you to the Sun documentation for that information.

```
  //Display first row of values by displaying every
  // other element in the array starting with element
  // zero.
  for(int cnt = 0; cnt < 6; cnt+=2){
    System.out.print(theMatrix[cnt] + " ");
  }//end for loop

  //Display second row of values displaying every
  // other element in the array starting with element
```

```
    // number one.
    System.out.println();//new line
    for(int cnt = 1; cnt < 6; cnt+=2){
      System.out.print(theMatrix[cnt] + " ");
    }//end for loop
    System.out.println();//end of line
    System.out.println();//blank line

  }//end displayMatrix
}//end class GUI
```

Figure 11

However, you can probably figure it out from Figure 11 that uses two sequential **for** loops to extract the information from the array and display it in the following order.

An arbitrary **AffineTransform** can be mathematically expressed by six numbers in a matrix like this:

```
 sx shx tx
shy  sy ty
```

Note that this is a partial reproduction of information that I provided earlier in this lesson and you will find the definition of each of the terms, such as **sx** and **shy** provided there.

Conclusion

Because of the availability of the convenience methods named **scale()**, **translate()**, **shear()**, and **rotate()**, it is very easy to implement Affine transformations of these four standard types.

These are linear transforms, and the end result is the cumulative effect of applying the transforms in a sequential fashion. Keep in mind that the order in which you apply the transforms may have a significant impact on the results.

If you need to implement more complex transforms, you can do so by creating your own six-element matrix of values and using that matrix to set the transform for a particular application.

Complete Program Listing

A complete listing of the program is provided in Figure 12.

```
/*Affine01.java 12/12/99
Copyright 1999, R.G.Baldwin
```

```
Illustrates use of the Affine transforms, and the
methods of the Graphics2D class that allow for applying
transforms of the following types:

Scale
Translate
Rotate
Shear

Also gets and displays the values in the AffineTransform
object after the transform has been applied by invoking
methods with the same names.

In addition to displaying a GUI that visually
illustrates the effects of the transforms, the program
also displays the following information on the screen.

Default Transform
1.0 0.0 0.0
0.0 1.0 0.0

Add Scale Transform
1.6666666666666667 0.0 0.0
0.0 1.6666666666666667 0.0

Add Translate Transform
1.6666666666666667 0.0 30.0
0.0 1.6666666666666667 30.0

Add Shear Transform
1.6666666666666667 0.08333333333333334 30.0
0.16666666666666669 1.6666666666666667 30.0

Add Rotate Transform
1.6508996608400615 -0.2434184299932779 79.32270275806317
0.4886147500940855 1.6021270803360292 -7.066823581936546

The visual display consists of five concentric squares
with the text string "Exit->" superimposed on the
squares.  This visual display illustrates the effects of
the transforms pretty well.

This program is intended to be used to experiment with
Affine transforms.  By using comment indicators "//"
to disable statements of the form:

    System.out.println("Add Translate Transform");
    g2.translate(0.25*ds,0.25*ds);
    displayMatrix(g2.getTransform());

and by rearranging those statements, the student can see
the individual and cumulative effects of applying the
transforms.

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

class Affine01{
  publicstaticvoid main(String[] args){
    GUI guiObj = new GUI();
  }//end main
}//end controlling class Affine01

class GUI extends Frame{
  int res;//store screen resolution here
  staticfinalint ds = 72;//default scale, 72 units/inch
```

```
GUI(){//constructor
  //Get screen resolution
  res = Toolkit.getDefaultToolkit().
                      getScreenResolution();

  //Set Frame size to four-inches by four-inches
  this.setSize(4*res,4*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 to draw and manipulate a
// square.
publicvoid paint(Graphics g){
  //Downcast the Graphics object to a Graphics2D object
  Graphics2D g2 = (Graphics2D)g;

  //Display contents of default AffineTransform object
  System.out.println("Default Transform");
  displayMatrix(g2.getTransform());

  //Update transform to include a scale component,
  // and display the values.
  System.out.println("Add Scale Transform");
  g2.scale((double)res/72,(double)res/72);
  displayMatrix(g2.getTransform());

  //Update transform to include a translate component,
  // and display the values.
  System.out.println("Add Translate Transform");
  g2.translate(0.25*ds,0.25*ds);
  displayMatrix(g2.getTransform());

  //Update transform to include a shear component,
  // and display the values.
  System.out.println("Add Shear Transform");
  g2.shear(0.05,0.1);
  displayMatrix(g2.getTransform());

  //Update transform to provide rotation and display,
  // the transform values.
  System.out.println("Add Rotate Transform");
  //11.25 degrees about center
  g2.rotate(Math.PI/16,2.0*ds, 2.0*ds);
  displayMatrix(g2.getTransform());

  //Draw five concentric squares and apply the transform
  // that results from the above transform updates.  If
  // the above scale transform is applied, the outer
  // square is one inch on each side, and the squares
  // are separated by 0.1 inch
  double delta = 0.1;
  for(int cnt = 0; cnt < 5; cnt++){
    g2.draw(new Rectangle2D.Double(
       (1.5+cnt*delta)*ds, (1.5+cnt*delta)*ds,
       (1.0-cnt*2*delta)*ds, (1.0-cnt*2*delta)*ds));
  }//end for loop

  //Superimpose some text on the squares with the
  // lower left corner of the first character at the
  // center of the squares.
  g2.drawString("Exit ->",2.0f*ds,2.0f*ds);
}//end overridden paint()

//This method receives a reference to an AffineTransform
```

```
  // and displays the six controllable values in the
  // transform matrix
  void displayMatrix(AffineTransform theTransform){
    double[] theMatrix = newdouble[6];
    theTransform.getMatrix(theMatrix);

    //Display first row of values by displaying every
    // other element in the array starting with element
    // zero.
    for(int cnt = 0; cnt < 6; cnt+=2){
      System.out.print(theMatrix[cnt] + " ");
    }//end for loop

    //Display second row of values displaying every
    // other element in the array starting with element
    // number one.
    System.out.println();//new line
    for(int cnt = 1; cnt < 6; cnt+=2){
      System.out.print(theMatrix[cnt] + " ");
    }//end for loop
    System.out.println();//end of line
    System.out.println();//blank line

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

Figure 12

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-