Java OOP: Scaling, Rotating, and Translating Images using Affine Transforms*

R.G. (Dick) Baldwin

Abstract

Learn how to scale images and how to rotate and translate images using the AffineTransform class.

1 Table of Contents

2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [1] .

*Version 1.3: Nov 14, 2012 12:18 pm -0600
†http://creativecommons.org/licenses/by/3.0/
[1]http://cnx.org/content/m44148/latest/

2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

2.1.1 Figures

2.1.2 Listings

3 Preview

In this module, you will learn how to scale images and how to rotate and translate images using the **AffineTransform** class.

Program specifications

Write a program named **Prob01** that uses the class definition shown in Listing 1 (p. 6) and Ericson's media library along with the image file named **Prob01.jpg** *(see Figure 1 (p. 3))* to produce the output images shown in Figure 2 (p. 4) and Figure 3 (p. 5) .

Input file named Prob01.jpg.

Figure 1: Input file named Prob01.jpg.

First output image.

Figure 2: First output image.

Second output image.

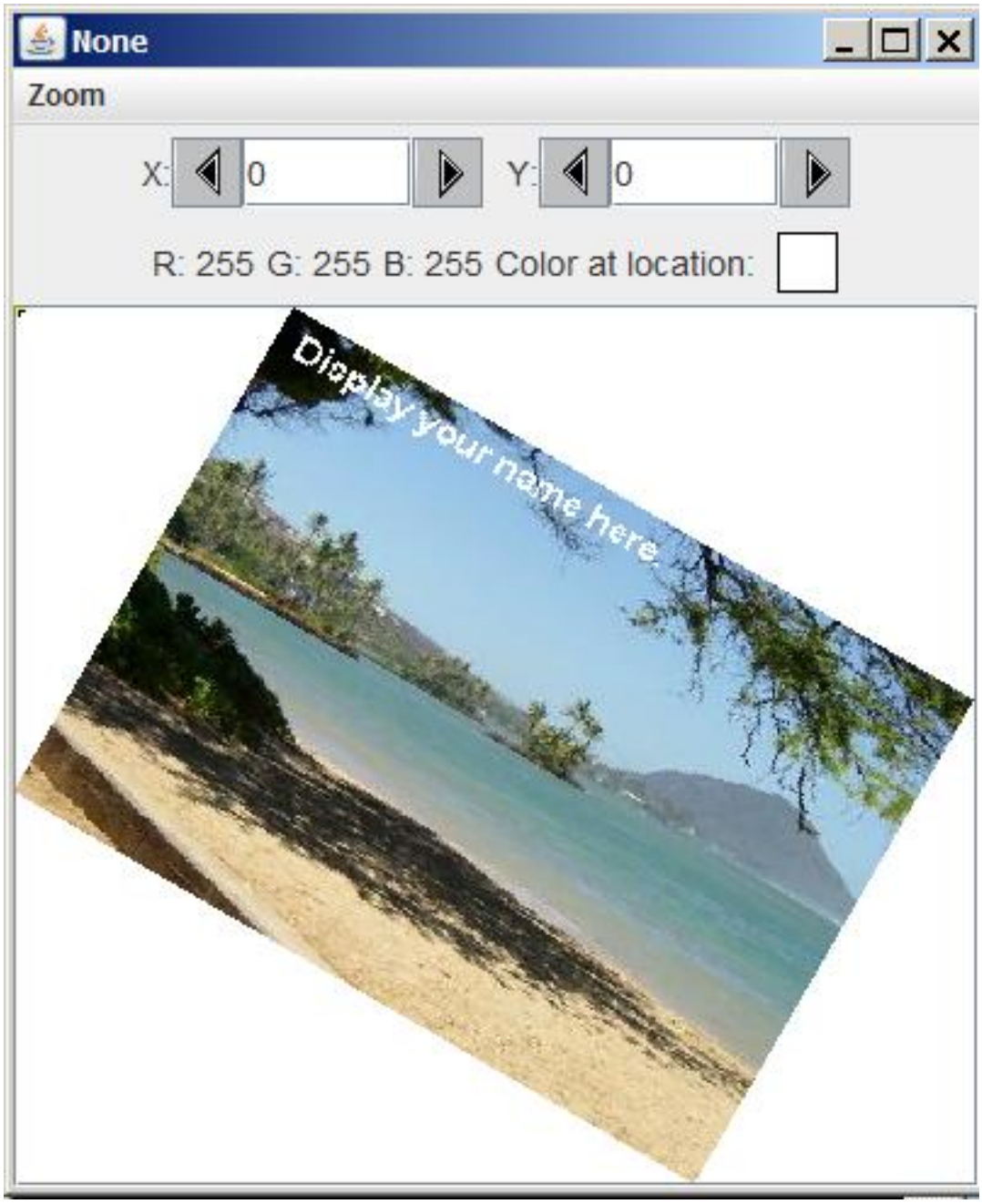

Figure 3: Second output image.

Scale and rotate

The image from the file named **Prob01.jpg** must be scaled and then rotated 30 degrees clockwise. A scale factor of 0.95 must be applied to the horizontal and a scale factor of 0.9 must be applied to the vertical.

New classes

You may define new classes as necessary to cause your program to behave as required, but you may not modify the class definition for the class named **Prob01** shown in Listing 1 (p. 6) .

Required output text

In addition to the two output images mentioned above, your program must display your name and the other line of text shown in Figure 4 (p. 6) on the command-line screen.

Required output text.

```
    Display your name here.
Picture, filename None height 360 width 394
```

Figure 4: Required output text.

---

4 General background information

Writing the code from scratch to rotate an image can be a daunting task. However, the task is made much easier through the use of the standard **AffineTransform** class, which is included in the standard Java library.

The **AffineTransform** class can also be used to scale and translate images.

5 Discussion and sample code

Will discuss in fragments

I will discuss and explain this program in fragments. A complete listing of the program is provided in Listing 10 (p. 13) near the end of the module.

The driver class named Prob01

The driver class containing the **main** method is shown in Listing 1 (p. 6) .

Listing 1: The driver class named Prob01.

```
    import java.awt.Graphics2D;
import java.awt.geom.AffineTransform;
import java.awt.geom.Rectangle2D;
import java.awt.Graphics;

public class Prob01{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
```

```
    new Prob01Runner().run();
  }//end main method
}//end class Prob01
```

Instantiate a new object and call its run method

As has been the case in several earlier modules, the code in the **main** method instantiates a new object of the class named **Prob01Runner** and calls the **run** method on that object.

When the **run** method returns, the **main** method terminates causing the program to terminate.

Beginning of the class named Prob01Runner

The class named **Prob01Runner** begins in Listing 2 (p. 7) .

Listing 2: Beginning of the class named Prob01Runner.

```
    class Prob01Runner{
  public Prob01Runner(){
    System.out.println("Display your name here.");
  }//end constructor
```

Listing 2 (p. 7) shows the constructor for the class, which simply displays the student's name on the command line screen as shown in Figure 4 (p. 6) .

The run method

The **run** method, which is called in Listing 1 (p. 6) , is shown in its entirety in Listing 3 (p. 7) .

Listing 3: The run method.

```
      public void run(){
    Picture pic = new Picture("Prob01.jpg");
    //Add your name and display the picture.
    pic.addMessage("Display your name here.",10,20);
    pic.explore();

    pic = pic.scale(0.95,0.9);
    pic = rotatePicture(pic,30);

    pic.explore();
    System.out.println(pic);
  }//end run
```

Mostly familiar code

You are already familiar with all of the code in Listing 3 (p. 7) except for the call to the **scale** method and the call to the **rotatePicture** method.

The scale method

The **scale** method is part of Ericson's library. However, it is not included in the library on the CD in the back of my copy of her textbook. It is included in the zip files containing later versions, which can be downloaded from Ericson's website. *(See Java OOP: The Guzdial-Ericson Multimedia Class Library* [2] *.)*

The scale method is straightforward

When the **scale** method is called on a **Picture** object, it creates and returns a reference to a new **Picture** object that is a scaled version of the original.

Parameters

[2] http://cnx.org/content/m44148/latest/

The **scale** method requires two parameters of type **double** . The first parameter is the scale factor that is applied to the horizontal dimension of the picture. The second parameter is the scale factor that is applied to the vertical dimension of the picture.

Replace original picture with scaled picture

The reference to the new **Picture** object returned by the **scale** method in Listing 3 (p. 7) is stored in the variable named **pic** overwriting the reference to the original **Picture** object. From this point forward, all operations are performed on the scaled version of the original picture.

Beginning of the method named rotatePicture

The method named **rotatePicture** begins in Listing 4 (p. 8) .

Listing 4: Beginning of the method named rotatePicture.

```
  private Picture rotatePicture(Picture pic,double angle){

//Prepare the rotation transform
AffineTransform rotateTransform =
                                new AffineTransform();
rotateTransform.rotate(Math.toRadians(angle),
                       pic.getWidth()/2,
                       pic.getHeight()/2);
```

Rotate and translate

The **rotatePicture** method accepts a reference to a **Picture** object along with a rotation angle in degrees.

It creates and returns a new **Picture** object that is of the correct size, containing the rotated version of the image as shown in Figure 3 (p. 5) .

The incoming image is rotated around its center by the specified rotation angle. Then it is translated to and drawn in the center of the new **Picture** object.

Affine transforms

The **rotatePicture** method uses *affine transforms* to rotate and translate the image. Affine transforms can also be used to scale images, but it is easier to scale images using Ericson's **scale** method.

However, the lack of complexity of the **scale** method is easily made up for by the complexity of affine transforms.

Google me

I have published several tutorials discussing and explaining the use of the **AffineTransform** class in Java. You can locate those modules by going to Google and searching for the following keywords:

richard baldwin affine transform

The AffineTransform class

The **AffineTransform** class is part of the standard Java library. Here is part of what the documentation [3] has to say about the class:

"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."

The ideas behind affine transforms

One of the ideas behind affine transforms is that you can create an affine transform object and apply it to an unlimited number of other objects. This might be useful in a game program, for example, where a large number of enemy ships need to be rotated, translated, and scaled in unison.

Concatenated affine transform objects

Another idea is that you can create two or more affine transform objects, concatenate them, and apply the concatenated transform object to an unlimited number of other objects.

[3] http://java.sun.com/javase/6/docs/api/java/awt/geom/AffineTransform.html

Application of concatenated transform objects

Applying a concatenated transform to an object is equivalent to applying one of the transform objects to the original object and then applying the other transform objects to the transformed objects in sequential fashion. Concatenation of transform objects can result in considerable computational savings in certain situations.

A larger Picture object is required

Looking back at Figure 2 (p. 4) and Figure 3 (p. 5) , you can see that the **Picture** object required to contain the rotated image must be larger than the **Picture** object required to contain the original image. You will learn how to compute the dimensions of the larger **Picture** object later in this module.

Behavior of the rotatePicture method

The **rotatePicture** method performs the following operations:

- Prepare an **AffineTransform** object that can be used to rotate the incoming image around its center by the specified angle.
- Get the dimensions of a rectangle of sufficient size to contain the rotated image.
- Prepare an **AffineTransform** object that will translate the rotated image to the center of a new, larger **Picture** object having the dimensions computed above.
- Concatenate the rotation transform object with the translation transform object.
- Create a new **Picture** object with the dimensions computed above.
- Apply the concatenated transform to the incoming image and draw the transformed image in the new **Picture** object.
- Return a reference to the new **Picture** object containing the rotated and translated image.

Prepare the rotation transform

Listing 4 (p. 8) begins by instantiating a new object of the **AffineTransform** class and saving the object's reference in the local reference variable named **rotateTransform** .

Call an overloaded rotate method

Then Listing 4 (p. 8) calls one of four overloaded **rotate** methods on the rotation transform object.

Three parameters are required

This version of the **rotate** method requires three parameters:

- theta - the angle of rotation measured in radians
- anchorx - the X coordinate of the rotation anchor point
- anchory - the Y coordinate of the rotation anchor point

To make a long story short...

The **rotate** method prepares the transform object to rotate an image around the point specified by the last two parameters.

The angle of rotation must be specified in radians.

Convert from degrees to radians

Listing 4 calls the static **toRadians** method of the **Math** class to convert the rotation angle from degrees to radians. The angle in radians is passed as the first parameter *(theta)* to the **rotate** method.

Compute the anchor point

Then the code in Listing 4 computes the coordinates of the center of the image and passes those coordinates to the **rotate** method as *anchorx* and *anchory* .

The dimensions of the new Picture object

How would you compute the dimensions of the new **Picture** object required to barely contain the rotated image shown in Figure 3 (p. 5) ?

The computation of those dimensions is not rocket science, but would certainly require you to know quite a lot about dealing with angles and triangles.

Fortunately, we don't have to perform that computation

Ericson provides a method named **getTransformEnclosingRect** that will perform that computation for us, returning the required dimensions in the form of a reference to a standard Java **Rectangle2D** object.

Compute the dimensions of the new Picture object

The code in Listing 5 (p. 10) calls the **getTransformEnclosingRect** method on the previously scaled **Picture** object passing a reference to the rotation transform object to get the required dimensions for a **Picture** object that will contain the rotated image.

Listing 5: Compute the dimensions of the new Picture object.

```
    Rectangle2D rectangle2D =
       pic.getTransformEnclosingRect(rotateTransform);

int resultWidth = (int)(rectangle2D.getWidth());
int resultHeight = (int)(rectangle2D.getHeight());
```

The dimensions of the rectangle

After getting a reference to the rectangle, Listing 5 (p. 10) gets and saves the **width** and **height** of the rectangle. These values will be used later to instantiate a new **Picture** object of the same size as the rectangle.

Prepare the translation transform

Listing 6 (p. 10) prepares a translation transform that can be used to translate the rotated image to the center of the new **Picture** object.

Listing 6: Prepare the translation transform.

```
    AffineTransform translateTransform =
                                new AffineTransform();
translateTransform.translate(
                  (resultWidth - pic.getWidth())/2,
                  (resultHeight - pic.getHeight())/2);
```

A new AffineTransform object

Listing 6 (p. 10) begins by instantiating a new object of the **AffineTransform** class and saving the object's reference in the local reference variable named **translateTransform** .

Call the translate method on the transform object

Then Listing 6 (p. 10) calls the **translate** method on the **AffineTransform** object.

According to the documentation [4] , the required parameters of the **translate** method are:

- tx - the distance by which coordinates are translated in the X axis direction
- ty - the distance by which coordinates are translated in the Y axis direction

Compute the translation distance components

Listing 6 (p. 10) computes the distance from the center of the image to the center of the new **Picture** object and passes the X and Y components of this distance to the **translate** method.

Two AffineTransform objects

At this point, we have two different **AffineTransform** objects. One is capable of rotating the image by a specified angle. The other is capable of translating the image by a specified amount.

We could apply the two transforms sequentially to the image being careful to rotate before we translate. *(The order of rotation and translation makes a huge difference.)*

[4] http://java.sun.com/javase/6/docs/api/java/awt/geom/AffineTransform.html#translate%28double,%20double%29

A more computationally economical approach

The preferred approach is to concatenate the two transform objects and apply only the concatenated transform object to the image. This is particularly important if the transforms are going to be applied to a large number of images such as in a game program for example.

Concatenate the transforms

Listing 7 (p. 11) calls the **concatenate** method on the translation transform passing a reference to the rotation transform as a parameter. This modifies the translation transform in such a way that it can be used to rotate the image around its center point and then translate it to the center of the new **Picture** object.

Listing 7: Concatenate the transforms.

```
translateTransform.concatenate(rotateTransform);
```

Instantiate the new Picture object

Listing 8 (p. 11) instantiates a new **Picture** object with the dimensions computed in Listing 5 (p. 10) . This **Picture** object will be used to contain and return the rotated image.

Listing 8: Instantiate the new Picture object .

```
Picture result = new Picture(
                        resultWidth,resultHeight);
```

Perform the concatenated transform

Listing 9 performs the rotation and translation, draws the modified image in the new **Picture** object, and returns a reference to the new **Picture** object.

Listing 9: Perform the concatenated transform .

```
    Graphics2D g2 = (Graphics2D)result.getGraphics();

   g2.drawImage(pic.getImage(),translateTransform,null);

   return result;
  }//end rotatePicture
}//end class Prob01Runner
```

Call the getGraphics method

Listing 9 (p. 11) begins by calling Ericson's **getGraphics** method on the new **Picture** object and casting the returned value to the standard type **Graphics2D** .

A cast is required

Ericson's **getGraphics** method returns a reference to the graphics context of the **Picture** object as type **Graphics** . That reference must be cast to type **Graphics2D** before the **drawImage** method can be called on the reference.

Call the drawImage method

Then Listing 9 (p. 11) calls the standard **drawImage** method on the reference to the graphics context passing three parameters to the method. This is one of two overloaded versions of the **drawImage** method defined in the standard **Graphics2D** class.

The first parameter

The first required parameter for this version of the **drawImage** method is a reference to an object of type **Image** containing the image that is to be drawn. In this case, Ericson's **getImage** method is called on the **Picture** object to get the image and pass it as the first parameter.

The second parameter

The second required parameter is a reference to an **AffineTransform** object that is to be applied to the image before it is drawn. Our concatenated transform object is passed as the second parameter.

The third parameter

The third required parameter is a reference to an object of the standard type **ImageObserver** or **null** . If you would like to know more about the use of an **ImageObserver** object, go to Google and search for the following keywords:

richard baldwin java imageobserver

We don't need an image observer in this case so Listing 9 (p. 11) passes **null** for the third parameter.

When the drawImage method returns

When the **drawImage** method returns, the image will have been rotated, translated, and drawn in the center of the new **Picture** object as shown in Figure 3 (p. 5) .

Return a Picture and terminate the method

Listing 9 (p. 11) returns a reference to the **Picture** object, *(which now contains the rotated image)* and terminates, returning control to the **run** method in Listing 3 (p. 7) .

Return to the run method

Returning to the **run** method in Listing 3 (p. 7) , we see that the remaining code in the **run** method:

- Calls Ericson's **explore** method on the returned **Picture** object producing the screen output shown in Figure 3 (p. 5) .
- Passes the returned **Picture** object to the **println** method producing the last line of text output shown in Figure 4 (p. 6) .
- Returns control the **main** method in Listing 1 (p. 6) , causing the program to terminate as soon as the user dismisses both images from the screen.

6 Run the program

I encourage you to copy the code from Listing 10 (p. 13) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. For example, try reversing the order of translation and rotation beginning with Listing 7 (p. 11) . Make certain that you can explain why your changes behave as they do.

Click Prob01.jpg [5] to download the required input image file.

7 Summary

You learned how to scale images and how to rotate and translate images using the **AffineTransform** class.

8 What's next?

In the next module, you will learn how to mirror images both horizontally and vertically.

9 Online video link

Select the following link to view an online video lecture on the material in this module.

- ITSE 2321 Lecture 11 [6]

[5] http://cnx.org/content/m44223/1.3/Prob01.jpg
[6] http://vimeo.com/channels/itse2321/21211960

10 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Scaling, Rotating, and Translating Images using Affine Transforms
- File: Java3022.htm
- Published: August 2, 2012
- Revised: November 14, 2012

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

11 Complete program listing

A complete listing of the program discussed in this module is shown in Listing 10 (p. 13) below.

Listing 10: Complete program listing.

```
    /*File Prob01 Copyright 2008 R.G.Baldwin
Revised 12/17/08
*******************************************************/
import java.awt.Graphics2D;
import java.awt.geom.AffineTransform;
import java.awt.geom.Rectangle2D;
import java.awt.Graphics;

public class Prob01{
  //DO NOT MODIFY THE CODE IN THIS CLASS DEFINITION.
  public static void main(String[] args){
    new Prob01Runner().run();
  }//end main method
}//end class Prob01
//=====================================================//

class Prob01Runner{
  public Prob01Runner(){
```

```
    System.out.println("Display your name here.");
  }//end constructor
  //-----------------------------------------------------//
  public void run(){
    Picture pic = new Picture("Prob01.jpg");
    //Add your name and display the picture.
    pic.addMessage("Display your name here.",10,20);
    pic.explore();
    pic = pic.scale(0.95,0.9);
    pic = rotatePicture(pic,30);

    pic.explore();
    System.out.println(pic);
  }//end run
  //-----------------------------------------------------//

  //This method accepts a reference to a Picture object
  // along with a rotation angle in degrees. It creates
  // and returns a new Picture object that is of the
  // correct size to contain and display the incoming
  // picture after it has been rotated around its center
  // by the specified rotation angle and translated to the
  // center of the new Picture object.
  private Picture rotatePicture(Picture pic,double angle){

    //Set up the rotation transform
    AffineTransform rotateTransform =
                                    new AffineTransform();
    rotateTransform.rotate(Math.toRadians(angle),
                           pic.getWidth()/2,
                           pic.getHeight()/2);

    //Get the required dimensions of a rectangle that will
    // contain the rotated image.
    Rectangle2D rectangle2D =
           pic.getTransformEnclosingRect(rotateTransform);
    int resultWidth = (int)(rectangle2D.getWidth());
    int resultHeight = (int)(rectangle2D.getHeight());

    //Set up the translation transform that will translate
    // the rotated image to the center of the new Picture
    // object.
    AffineTransform translateTransform =
                                    new AffineTransform();
    translateTransform.translate(
                     (resultWidth - pic.getWidth())/2,
                     (resultHeight - pic.getHeight())/2);

    //Concatenate the two transforms so that the image
    // will first be rotated around its center and then
    // translated to the center of the new Picture object.
```

```
    translateTransform.concatenate(rotateTransform);
    //Create a new Picture object to contain the results
    // of the transformation.
    Picture result = new Picture(
                               resultWidth,resultHeight);

    //Get the graphics context of the new Picture object,
    // apply the transform to the incoming picture and
    // draw the transformed picture on the new Picture
    // object.
    Graphics2D g2 = (Graphics2D)result.getGraphics();
    g2.drawImage(pic.getImage(),translateTransform,null);

    return result;
  }//end rotatePicture
  //-----------------------------------------------------//

}//end class Prob01Runner
```

-end-