

Affine Transformation for BufferedImage

1. Affine Transformation

Affine transformation is a linear mapping method that preserves points, straight lines, and planes. Sets of parallel lines remain parallel after an affine transformation.

The affine transformation technique is typically used to correct for geometric distortions or deformations that occur with non-ideal camera angles. For example, satellite imagery uses affine transformations to correct for wide angle lens distortion, panorama stitching, and image registration. Transforming and fusing the images to a large, flat coordinate system is desirable to eliminate distortion. This enables easier interactions and calculations that don't require accounting for image distortion.

Table 1 illustrates the different affine transformations: translation, scale, shear, and rotation. The matrix in Table 1 is used in principle to calculate the conversion destination (x', y') of the point (x, y) with the following expression.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = A \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

where **A** is the matrix.

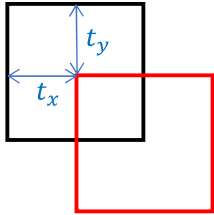
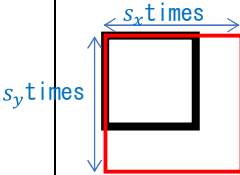
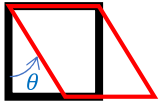
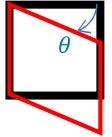
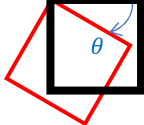
2. AffineTransform class

One of the easiest ways to operate an AffineTransform object is to use one of the static methods defined by AffineTransform. For example, getScaleInstance method returns an instance of AffineTransform that represents a simple scaling transformation.

Another way to get an AffineTransform is with the AffineTransform constructor. The no argument version of the constructor returns an AffineTransform that represents the identity transform – that is no transform at all. We can modify this empty transform with a number of methods.

This text introduces this method.

Table 1. Affine Translation.

affine Transform	Example	Transformation Matrix
Translation		$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$ <p>t_x specifies the displacement along the x axis t_y specifies the displacement along the y axis.</p>
Scale		$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$ <p>s_x specifies the scale factor along the x axis s_y specifies the scale factor along the y axis.</p>
Shear		$\begin{bmatrix} 1 & \tan \theta & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ <p>θ specifies the angle of shear along the x axis</p>
		$\begin{bmatrix} 1 & 0 & 0 \\ \tan \theta & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ <p>θ specifies the angle of shear along the y axis</p>
Rotation		$\begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$ <p>θ specifies the angle of rotation</p>

The following is the procedure to set 2D affine transformation with “AffineTransform” and execute with “AffineTransformOp”. The constructor of “AffineTransformOp” is as follows.

```
AffineTransformOp(AffineTransform xform, int interpolationType);
```

xform: The AffineTransform to use for the operation.

interpolationType: One of the integer interpolation type constants defined by this class:

TYPE_NEAREST_NEIGHBOR, TYPE_BILINEAR, TYPE_BICUBIC.

TYPE_BICUBIC is used for smooth interpolation.

It is assumed that the source image is assigned to the BufferedImage variable “bimage1”.

(1) Prepare a BufferedImage variable containing the converted image.

Example:

```
BufferedImage bimage2 = new BufferedImage(width, height, bimage1.getType());
```

- (2) Prepare the “AffineTransform” object.

Example:

```
AffineTransform affine = new AffineTransform();
```

- (3) Register the transformations on the “AffineTransform” object (for example, “affine”). The methods shown in Table 2 can be used to register the transformations.
- (4) Register the “AffineTransform” object in “AffineTransformOp” and create an object.

Example:

```
AffineTransformOp operator =  
    new AffineTransformOp(affine, AffineTransformOp.TYPE_BICUBIC);
```

- (5) Convert from source image to the destination image using “filter” method of “AffineTransformOp”.

Example:

```
operator.filter(bimage1, bimage2);
```

Table 2. The main methods for affine transformation.

Method & Description for affine transformation
void rotate(double <i>theta</i>) Rotates <i>theta</i> radian clockwise around the origin (0, 0).
void rotate(double <i>theta</i>, double <i>anchorx</i>, double <i>anchory</i>) Rotates <i>theta</i> radian clockwise around an anchor point (<i>anchorx</i> , <i>anchory</i>).
void scale(double <i>sx</i>, double <i>sy</i>) Expands (reduces) by <i>sx</i> in the x-axis direction and by <i>sy</i> in the y-axis direction.
void shear(double <i>shx</i>, double <i>shy</i>) Shears using <i>shx</i> and <i>shy</i> . <i>shx</i> : the multiplier by which coordinates are shifted in the direction of the positive X axis as a factor of their Y coordinate. <i>shy</i> : the multiplier by which coordinates are shifted in the direction of the positive Y axis as a factor of their X coordinate.
void translate(double <i>tx</i>, double <i>ty</i>) Translates by <i>tx</i> in the x-axis direction and by <i>ty</i> in the y-axis direction.

The program “AffineMap0” in Figure 1 is the basic pattern of the affine transformation program.

```
AffineMap0.java
1  /*
2   * Basic pattern of AffineTransform
3   * by Hitoshi Ogawa on December 13th, 2021
4   */
5  import java.awt.Graphics;
6  import java.awt.geom.AffineTransform;
7  import java.awt.image.AffineTransformOp;
8  import java.awt.image.BufferedImage;
9  import java.io.File;
10 import javax.imageio.ImageIO;
11 import javax.swing.JFrame;
12
13 public class AffineMap0 extends JFrame {
14     BufferedImage bimage1, bimage2;
15     String filename = "image3.jpg";
16
17     public AffineMap0() {
18         setSize(500, 810);
19         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20         setVisible(true);
21         initialization();
22         affineTransExample();
23         repaint();
24     }
25
26     public void paint(Graphics g){
27         super.paint(g);
28         int hposition = 440;
29
30         if(bimage1 != null){
31             g.drawImage(bimage1, 10, 70, this);
32         }
33         if(bimage2 != null){
34             g.drawImage(bimage2, 10, hposition, this);
35         }
36     }
37
38     void initialization() {
39         try {
40             bimage1 = ImageIO.read(new File(filename));
41         } catch (Exception e) {
42             System.out.println("error " + e);
43         }
44     }
45
46     void affineTransExample() {
47         /* the procedure of affine transformation */
48     }
49
50     public static void main(String[] args) {
51         new AffineMap0();
52     }
53 }
```

Figure 1. The basic pattern of the affine transformation program.

In the field, `BufferedImage` variable “bimage1” to which the loaded the source image is assigned and `BufferedImage` variable “bimage2” to substitute the image whose length is halved are prepared (line 14). The String variable “filename” is initialized with the file name of the original image (line 15).

In the constructor, a window with a width of 500 pixels and a height of 810 pixels is prepared (line 18). This is because the size of the image is assumed to be 480 X 360 and the processed image is drawn below.

In the paint method, “bimage1” and “bimage2” are drawn if their contents are assigned. If an image is assigned to bimage1, bimage1 is drawn in lines 29 to 31. The same processing is performed for bimage2. Alignment must be considered to draw two images that are the same size as the original image. Furthermore, in order to be able to place buttons in the upper part, the upper left coordinates of bimage1 and bimage2 are set to (10, 70) and (10, 440), respectively.

The file “image3.jpg” is assigned to the variable “bimage1” in the initialization method. The affine transformation procedure is described in the “affneTransExample” method. This method name may be changed according to the content.

2. 1 Scale

The “scale” method is used instead of the `affineTransExample` method. This method reduces the image to 80%. Figure 2 shows the “scale” method. In this case, “`affineTransExmple()`” on the lines 45 to 47 of the basic system is rewritten to “`scale()`”. The 22nd line of “AffineMap0.java” is also rewritten to “`scale()`;”.

In the scale method, variables “width” and “height” are assigned values that are 80% of the width and height of “bimage1”. `BufferedImage` variable “bimage2” for substituting an image whose length is half is prepared. The image type of “bimage1” is used as the image type described in the third argument of the `BufferedImage` constructor (line 50 in Figure 2).

The object of `AffineTransform` is assign to the variable “affine”. The scale method that sets the image length to 80% is set to variable affine (line 52 in Figure 2).

```
affine.scale(scale, scale);
```

where scale = 0.8

The image1 is converted to image2 by filter method of `AffineTransformOP` (lines 53 to 55 in Figure 2).

The execution result of the “scale” method is shown in Figure 3.

```

45  void scale() {
46      double scale = 0.8;
47
48      int width = (int)(bimage1.getWidth() * scale);
49      int height = (int)(bimage1.getHeight() * scale);
50      bimage2 = new BufferedImage(width, height, bimage1.getType());
51      AffineTransform affine = new AffineTransform();
52      affine.scale(scale, scale);
53      AffineTransformOp operator =
54          new AffineTransformOp(affine, AffineTransformOp.TYPE_BICUBIC);
55      operator.filter(bimage1, bimage2);
56  }

```

Figure 2. The scale method to make image 80% length.

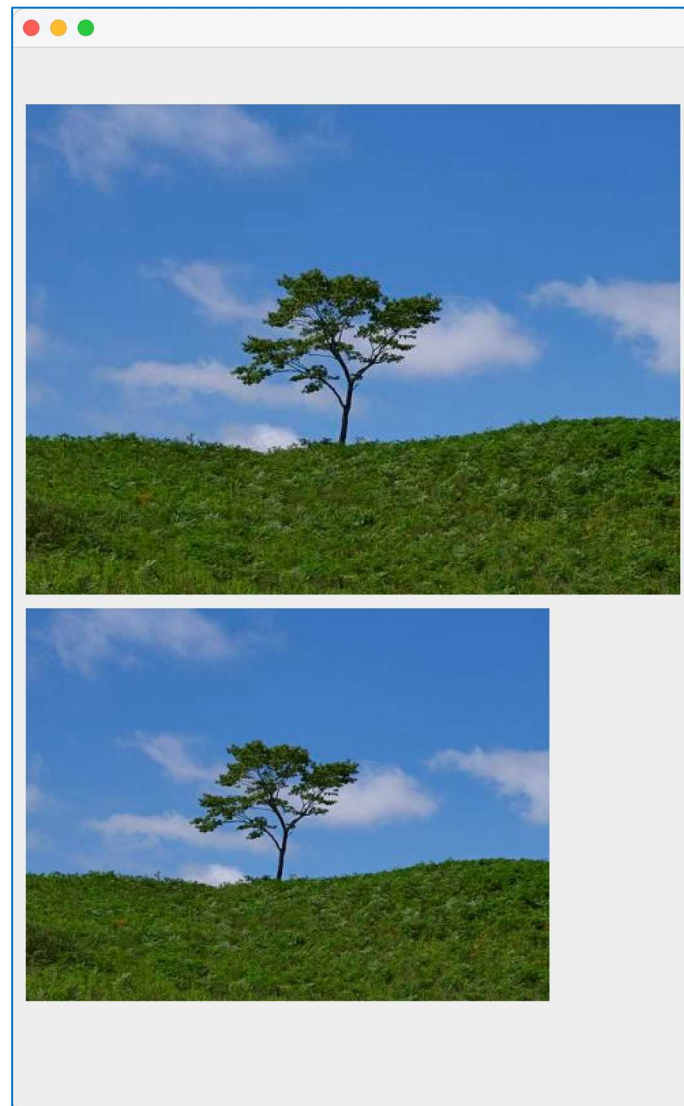


Figure 3. The execution result of the scale method.

2. 2 Translation

As an example of moving an image, the “translate()” method in Figure 4 is used instead of “affineTransExample()”. The 22nd line of “AffineTransformExample.java” is also rewritten to “translate()”.

The variable `bimage2` of the same size as `bimage1` is prepared (lines 46 and 47 in Figure 4). The conversion is executed by “`translate(50, 20)`” (line 49 in Figure 4), and the image is moved 50 pixels in the x axis direction and 20 pixels in the y axis direction.

Figure 5 shows the execution result of the scale method.

```
45  void translate() {  
46      bimage2 = new BufferedImage(bimage1.getWidth(),  
47                                  bimage1.getHeight(), bimage1.getType());  
48      AffineTransform affine = new AffineTransform();  
49      affine.translate(50, 20);  
50      AffineTransformOp operator =  
51          new AffineTransformOp(affine, AffineTransformOp.TYPE_BICUBIC);  
52      operator.filter(bimage1, bimage2);  
53  }
```

Figure 4. The translate method that moves the image 50 pixels in the x axis direction and 20 pixels in the y axis direction.



Figure 5. The execution result of the scale method.

2.3 Rotation

As an example of rotating an image, the “rotate()” method in Figure 6 or 7 is used instead of “affineTransExample()”. The 22nd line of “AffineMap0.java” is also rewritten to “ rotate()”.

To use the rotate method of AffineTransform, radians must be used. To convert degrees to radians, “toRadians” method of Math class is used (See Section 5.4 on page 13 of “AWT Event Handling” (The sixth day)). When rotating the image 30 degrees clockwise, use

Math.toRadians(30).

The rotate method on line 49 in Figure 6 has one argument, so it rotates around the origin (0, 0). The rotate method on line 49 in Figure 7 has three arguments which indicates the rotation angle and the center of rotation. In this example, the center of the image (240, 180) is specified.

The execution results of the rotate methods in Figure 6 and 7 are shown in Figure 8 and 9, respectively.

```
45  void rotate() {  
46      bimage2 = new BufferedImage(bimage1.getWidth(),  
47                                  bimage1.getHeight(), bimage1.getType());  
48      AffineTransform affine = new AffineTransform();  
49      affine.rotate(Math.toRadians(30));  
50      AffineTransformOp operator =  
51          new AffineTransformOp(affine, AffineTransformOp.TYPE_BICUBIC);  
52      operator.filter(bimage1, bimage2);  
53  }
```

Figure 6. The rotate method with one argument.

```
44  
45  void rotate() {  
46      bimage2 = new BufferedImage(bimage1.getWidth(),  
47                                  bimage1.getHeight(), bimage1.getType());  
48      AffineTransform affine = new AffineTransform();  
49      affine.rotate(Math.toRadians(30), 240, 180);  
50      AffineTransformOp operator =  
51          new AffineTransformOp(affine, AffineTransformOp.TYPE_BICUBIC);  
52      operator.filter(bimage1, bimage2);  
53  }
```

Figure 7. The rotate method with three arguments.

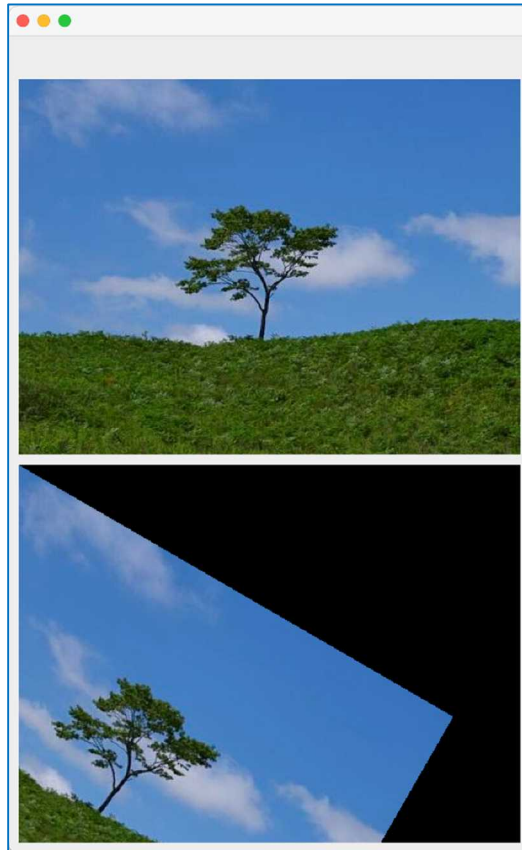


Figure 8. The execution result of the rotate method with one argument.

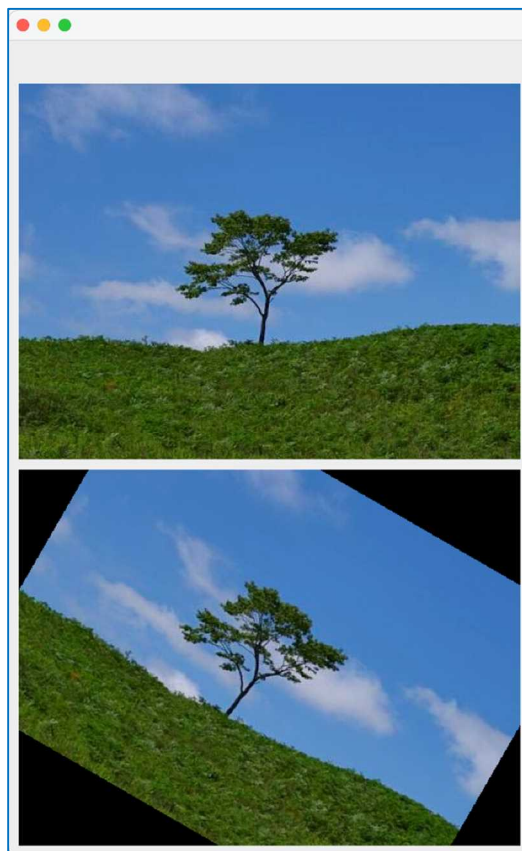


Figure 9. The execution result of the rotate method with three arguments.

2. 4 Shear

As an example of shearing an image, the “shear()” method in Figure 10 or 11 is used instead of “affineTransExample()”. The 22nd line of “AffineMap0.java” is also rewritten to “shear()”.

To use the method shear of AffineTransform, radians must be used.

If the image is sheared at the same size as the original, the result is difficult to understand, so it is reduced to 80%.

In the shear method of Figure 10, the image is sheared 15 degrees in the x direction. In the shear method of Figure 11, the image is sheared 10 degrees in the y direction.

The execution results of the shear methods in Figure 10 and 11 are shown in Figure 12 and 13, respectively.

```
45  void shear() {  
46      bimage2 = new BufferedImage(bimage1.getWidth(),  
47          bimage1.getHeight(), bimage1.getType());  
48      AffineTransform affine = new AffineTransform();  
49      affine.scale(0.8, 0.8);  
50      affine.shear(Math.toRadians(15), 0);  
51      AffineTransformOp operator =  
52          new AffineTransformOp(affine, AffineTransformOp.TYPE_BICUBIC);  
53      operator.filter(bimage1, bimage2);  
54  }
```

Figure 10. The shear method to shear 15 degrees in the x direction.

```
45  void shear() {  
46      bimage2 = new BufferedImage(bimage1.getWidth(),  
47          bimage1.getHeight(), bimage1.getType());  
48      AffineTransform affine = new AffineTransform();  
49      affine.scale(0.8, 0.8);  
50      affine.shear(0, Math.toRadians(10));  
51      AffineTransformOp operator =  
52          new AffineTransformOp(affine, AffineTransformOp.TYPE_BICUBIC);  
53      operator.filter(bimage1, bimage2);  
54  }
```

Figure 11. The shear method to shear 10 degrees in the y direction.

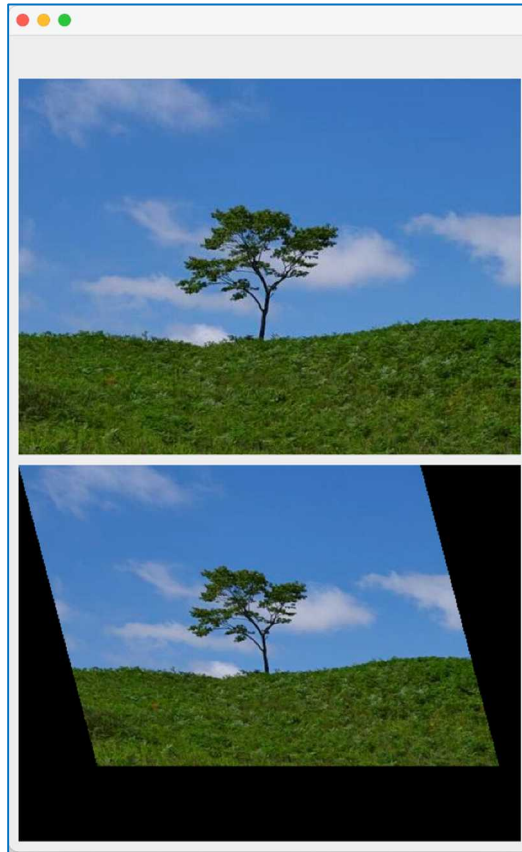


Figure 12. The execution result of the shear method in Figure 10.

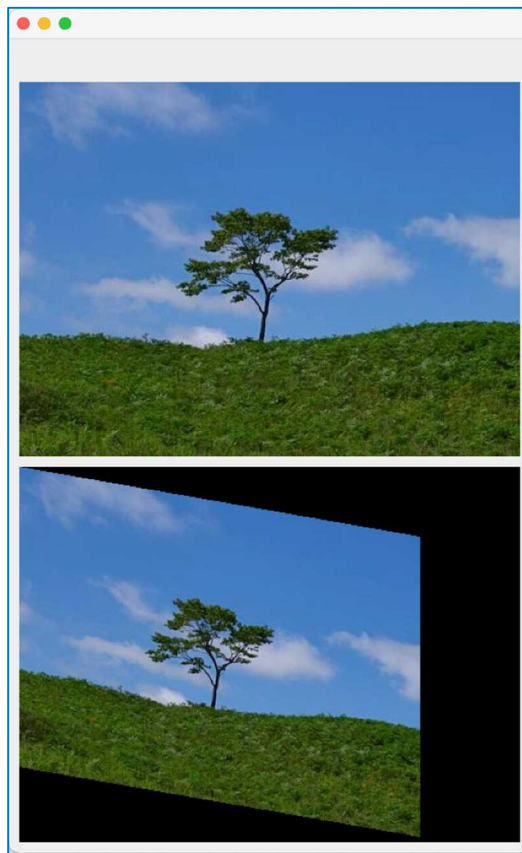


Figure 13. The execution result of the shear method in Figure 11.

3. Combining Scale and Rotate

In AffineTransform, the image is enlarged / reduced based on the origin (0, 0). In the example program in Figure 6 of Section 2.3, the rotation is also based on the origin. AffineTransform prepares method

`rotate(double theta, double anchorx, double anchory)`

that can specify its center (anchorx, anchory) in the case of rotation. However, in order to use it after scale, it is necessary to calculate the center of rotation.

This chapter introduces a method of enlarging / reducing and rotating around arbitrary place relatively easily. The procedure for enlarging / reducing or moving acertain coordinates (cx, cy) is as follows.

- (1) Determine coordinate (cx, cy) to be the center of processing (Figure 14 (a))
- (2) Move the coordinates (cx, cy) to the origin (0, 0) (Figure 14 (b)).
- (3) Execute enlargement / reduction and rotation around the origin (0, 0) (Figure 14 (c)).
- (4) Move from the origin (0, 0) to the coordinates (cx, cy) (Figure 14 (d)).

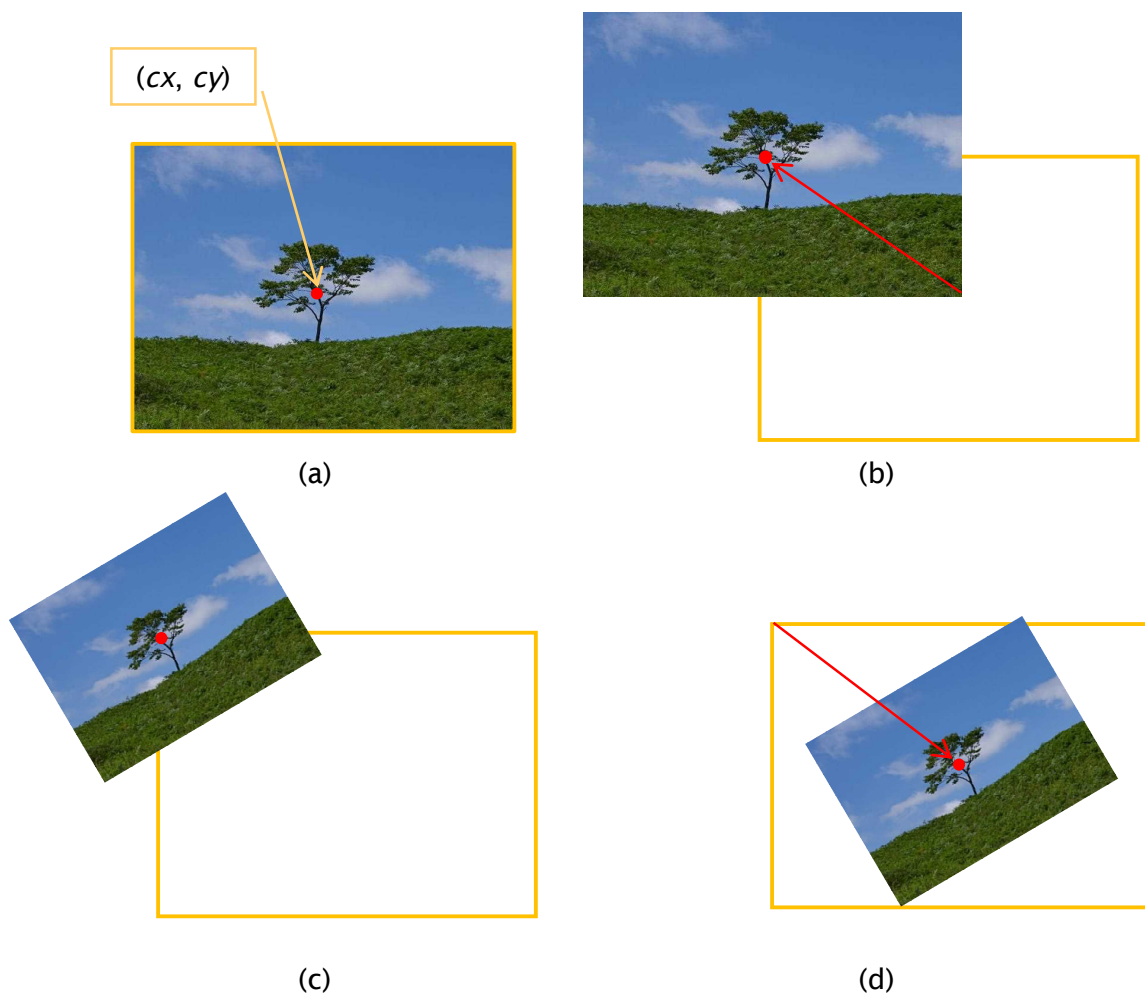


Figure 14 The procedure for affine transform from (a) to (d).

(A) AffineMap5

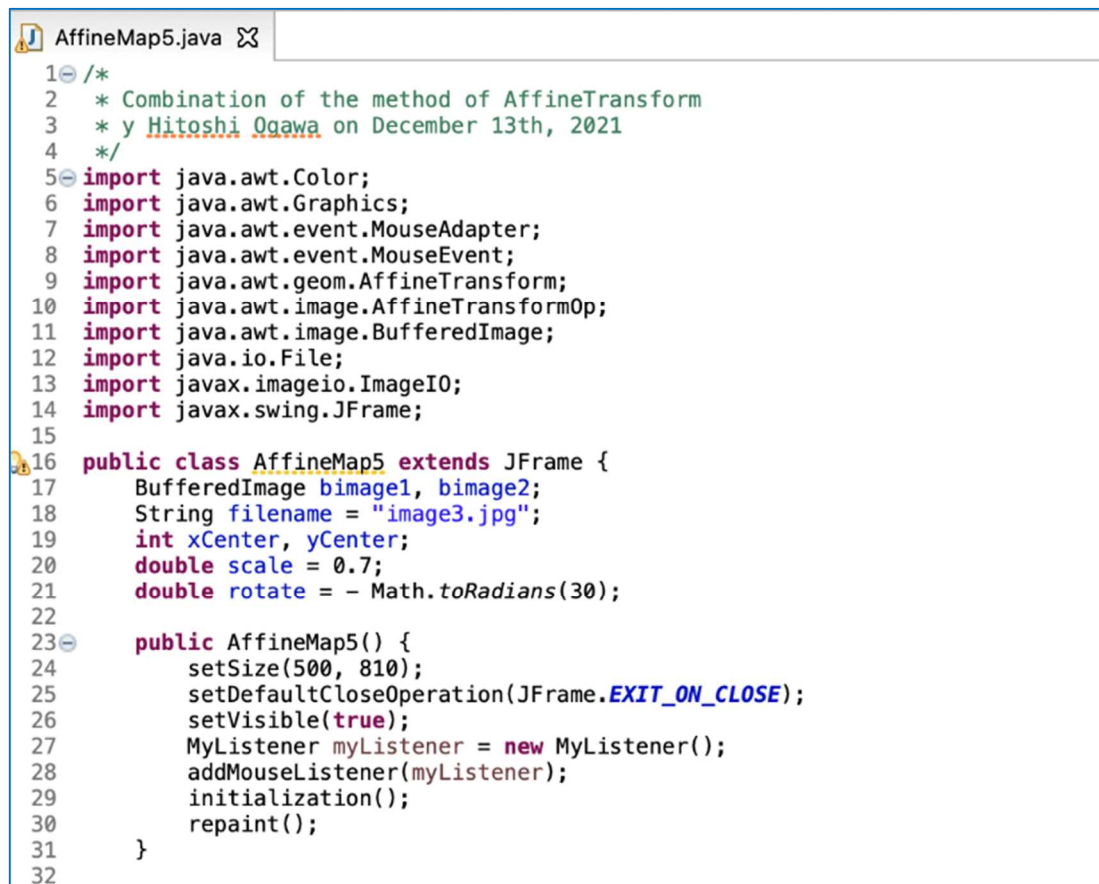
AffineMap5 is an example of a program that reduces and rotates around the place pressed on the image.

Figure 15 shows the program source of AffineMap5.

In the field, the important variables are declared. BufferedImage variable “bimage1” is prepared for the original image. BufferedImage variable “bimage2” is prepared for the image obtained by transformation. String variable “filename” is initialized by “image3.jpg”. The int type variable “xCenter” and “yCenter” which record the pressed place are prepared. The initial value of the double variable “scale” is set to 0.7, and the initial value of the double variable “rotate” is set to the value obtained by converting -30 degrees (30 degrees counterclockwise) to radians.

In the paint method, the original image and the result are displayed without scale. A red circle of size 5 is drawn in the pressed place.

In the inner class MyListener, considering where the image is drawn, a value obtained by decreasing 10 pixels from the x coordinate when pressed is assigned to “xCenter” and a value obtained by decreasing 30 pixels from the y coordinate is assigned to “yCenter”. Finally, the transform method is called.

The image shows a screenshot of a Java IDE with a single file named 'AffineMap5.java' open. The code is a Java Swing application that demonstrates affine transformations. It includes imports for AWT and Swing classes, a main class 'AffineMap5' extending 'JFrame', and an inner class 'MyListener' for handling mouse events. The code is color-coded with syntax highlighting. Line numbers 1 through 32 are visible on the left side of the editor.

```
1  /*
2   * Combination of the method of AffineTransform
3   * y Hitoshi Ogawa on December 13th, 2021
4   */
5  import java.awt.Color;
6  import java.awt.Graphics;
7  import java.awt.event.MouseAdapter;
8  import java.awt.event.MouseEvent;
9  import java.awt.geom.AffineTransform;
10 import java.awt.image.AffineTransformOp;
11 import java.awt.image.BufferedImage;
12 import java.io.File;
13 import javax.imageio.ImageIO;
14 import javax.swing.JFrame;
15
16 public class AffineMap5 extends JFrame {
17     BufferedImage bimage1, bimage2;
18     String filename = "image3.jpg";
19     int xCenter, yCenter;
20     double scale = 0.7;
21     double rotate = - Math.toRadians(30);
22
23     public AffineMap5() {
24         setSize(500, 810);
25         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26         setVisible(true);
27         MyListener myListener = new MyListener();
28         addMouseListener(myListener);
29         initialization();
30         repaint();
31     }
32 }
```

Figure 15. An example of a program combining the transfer method.


```

33 public void paint(Graphics g){
34     super.paint(g);
35     int hposition = 440;
36     int size = 5;
37
38     if(bimage1 != null){
39         g.drawImage(bimage1, 10, 70, this);
40     }
41     if(bimage2 != null){
42         g.drawImage(bimage2, 10, hposition, this);
43     }
44     if(xCenter != 0) {
45         g.setColor(Color.red);
46         g.fillRect(xCenter - size + 10, yCenter - size + 70, size, size);
47     }
48 }
49
50 void initialization() {
51     try {
52         bimage1 = ImageIO.read(new File(filename));
53     } catch (Exception e) {
54         System.out.println("error " + e);
55     }
56 }
57
58 void transform() {
59     bimage2 = new BufferedImage(bimage1.getWidth(),
60                                bimage1.getHeight(), bimage1.getType());
61     AffineTransform affine = new AffineTransform();
62     affine.translate(xCenter, yCenter);
63     affine.scale(scale, scale);
64     affine.rotate(rotate);
65     affine.translate(-xCenter, -yCenter);
66     AffineTransformOp operator =
67         new AffineTransformOp(affine, AffineTransformOp.TYPE_BICUBIC);
68     operator.filter(bimage1, bimage2);
69 }
70
71 public static void main(String[] args) {
72     new AffineMap5();
73 }
74
75 class MyListener extends MouseAdapter {
76
77     public void mousePressed(MouseEvent e) {
78         xCenter = e.getX() - 10;
79         yCenter = e.getY() - 70;
80         System.out.println(xCenter + ", " + yCenter);
81         transform();
82         repaint();
83     }
84 }
85 }

```

Figure 15. An example of a program combining the transfer method (continue).

In the transform method, four methods are registered in the variable “affine”. Note that the last registered process is intended to be executed first compared to the procedure in Figure 14.

Figure 16 and 17 show the execution result of the program in Figure 15.

Figure 16 shows the result of processing near the center of the image. Figure 17 shows the result of pressing near the upper left of the image.

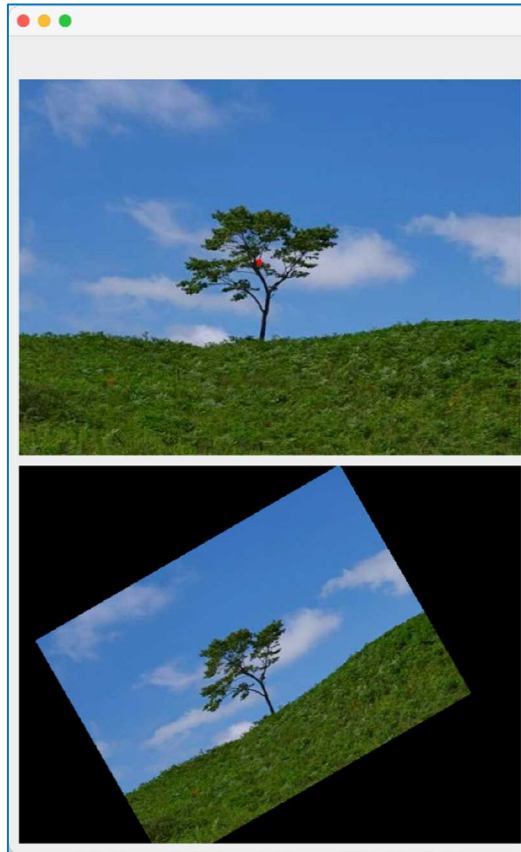


Figure 16. The result of processing near the center of the image.

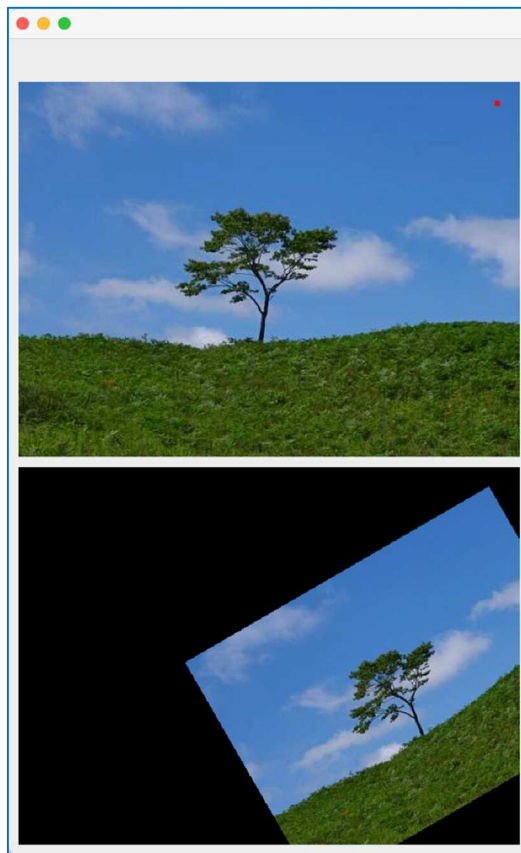


Figure 17. The result of pressing near the upper right of the image.