Chapter 4: Affine transformations and stroking

In the last chapter, we used a `Graphics2D` object merely as a drawing surface. However, a `Graphics2D` maintains an internal state, similar to PostScript's graphics state, that can be manipulated giving us more control over the way in which shapes are rendered. In this chapter, we'll explore some of these features, paying special attention to affine transformations, and see how they might be used.

1: The rendering engine's internal state

An instance of `Graphics2D` maintains seven pieces of information that are used when a `Shape` is rendered. We saw a few of these in the last chapter. Most of these can be accessed, in the usual way, through `get` and `set` methods.

1. Paint: This determines what `Paint` is applied when a figure is rendered. At this point, you may think of a `Paint` as simply specifying a `Color`. There is more we can do with this, however, such as specifying a gradient fill.

2. Font: There is, at all times, a `Font` available for displaying any text. If may be modified through the `setFont` method of `Graphics2D`.

3. Transformation: We define a `Shape` by specifying its coordinates. The details of how these coordinates are converted into pixel coordinates we see on the screen are handled through an affine transformation that may be modified. In fact, Java2D provides a class `AffineTransform` that allows us to work with 2-dimensional affine transformations easily.

4. Stroke: When `Shapes` are drawn, we can control the thickness of the curve and other attributes through a `Stroke` object. This allows us, for instance, to draw dashed lines.

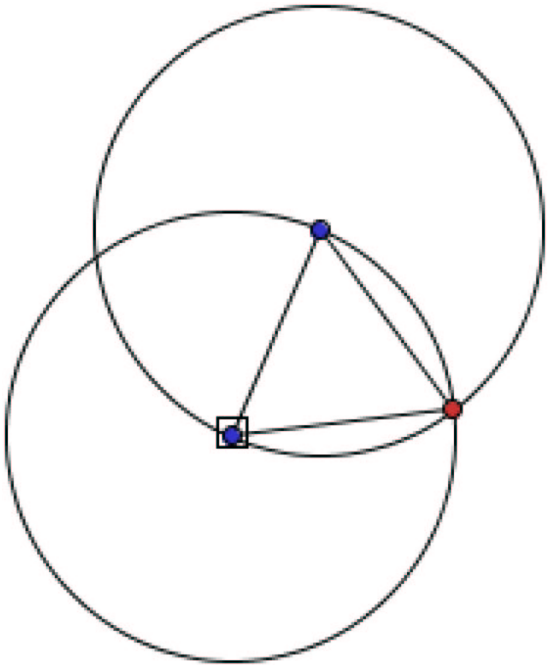
5. Clipping shape: We may specify a `Shape` to restrict the region in which we actually draw other `Shapes`. For instance, if we set the clipping shape to a rectangle and then draw a line, any portion of that line outside the rectangle is not displayed.

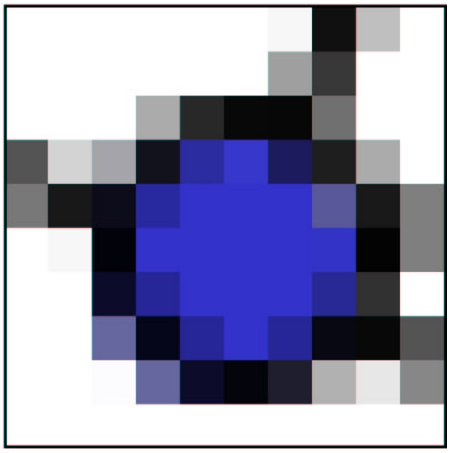
6. Compositing rule: This specifies a recipe by which colors are blended as they are added to the drawing surface. This may be used to achieve some interesting effects.

7: Rendering hints: Java2D allows us to control how some rendering operations are performed. Typically, we can use this to ask for either high quality or better performance. One particularly useful hint is anti-aliasing. When we display a black and white image on, say, a computer screen where the image is displayed in pixels, the image may appear jagged since a pixel is either black or white. Anti-aliasing is a technique that decides to color nearby pixels in shades of gray to smooth the discontinuity in the image. This can have a dramatic effect on the quality of the image. To turn this hint on, use this

```
g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                   RenderingHints.VALUE_ANTIALIAS_ON);
```

This will generally produce more attractive illustrations at the cost of more time for the rendering process.





Other `RenderingHints` may be found by studying the documentation.

2: Affine Transformations

The coordinate system that Java provides for our drawing surface is not well suited for general mathematical illustrations. First, it is not oriented in the conventional way as the measure on the vertical axis increases as we move down. Second, pixels are not usually a convenient unit to us. Fortunately, Java2D provides us with a convenient way to change the coordinate system to one more favorable for our purposes.

Affine transformations of the plane can be written in the form

$$x' = ax + by + c$$
$$y' = dx + ey + d.$$

Transformations of this kind have the pleasant properties that lines are carried into lines and a set of parallel lines into another set of parallel lines. Indeed, an affine transformation may be thought of a linear transformation composed with a translation. As such, they include familiar transformations such as translations, rotations and reflections.

Affine transformations are important in Java2D and a class, `java.awt.geom.AffineTransform`, facilitates their use. In a moment, we will see the worth of `AffineTransforms`. But first, let's see how to create one. There are many ways to do this and you are advised to consult the documentation. However, if we use the empty constructor

```
AffineTransform transform = new AffineTransform();
```

then the transformation represented is the identity. From here, there are methods that make simple modifications to the transformation. For instance, `transform.rotate(-Math.PI/2)` creates an affine

transformation describing a rotation about the origin of $-\pi/2$ radians. (Notice that rotations are measured in radians.) We can also perform translations, scaling and shears using

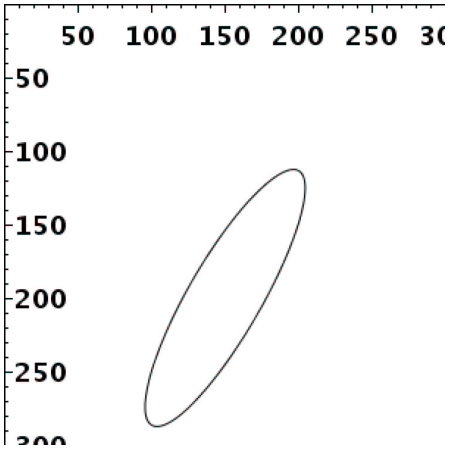
```
transform.translate(100, 200);
transform.scale(2, -1);
transform.shear(1, 0);
```

In each case, the transformation, originally representing a function f, is composed with a new function, say a translation represented by g, to produce the new transformation $f \circ g$. More generally, if `f1` and `f2` are `AffineTransforms`, then `f1.concatenate(f2)` replaces `f1` by an `AffineTransform` representing $f_1 \circ f_2$.

`AffineTransforms` may be used in two different ways. First, if we have a `Shape`, such as a `Line2D` or `Ellipse2D`, we may transform the `Shape` to obtain a new one. Here is an example:

```
AffineTransform at = new AffineTransform();
at.translate(150, 200);
at.rotate(-Math.PI/3);
at.scale(2, 0.5);
Ellipse2D.Float circle = new Ellipse2D.Float(-50, -50, 100, 100);
Shape shape = at.createTransformedShape(circle);
g.draw(shape);
```

The result looks like this:



While the `AffineTransform` is created by first translating, then rotating and scaling, the effects on the circle occur in the opposite order: first it is scaled, then rotated and finally translated. This is natural given the order in which the transformations are composed.

As mentioned above, a `Graphics2D` object maintains an `AffineTransform` as part of its internal state and this transform may be manipulated. This gives the second way in which `AffineTransforms` may be used. To see how this works, let us introduce two important coordinate systems.

When we are displaying our illustration on a computer screen, Java must determine how each pixel will be colored. It addresses these pixels through a coordinate system known as `device space`. We generally don't need to worry about this coordinate system since its use is taken care of for us by the

`Graphics2D`. However, you should realize that instead of drawing onto a computer screen, we may be printing the illustration on a printer. In this case, device space gives a coordinate system for addressing different regions on the piece of paper. Either way, these details are handled for us by the instance of `Graphics2D`.

When we draw, say, a line:

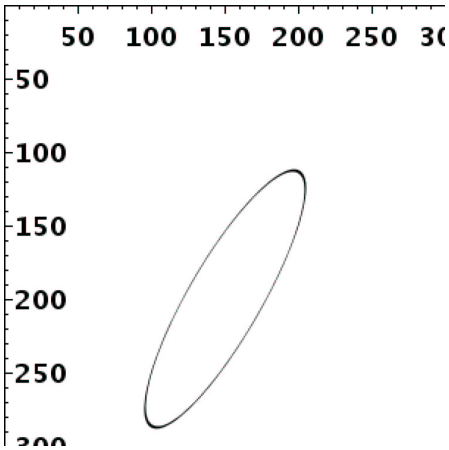
```
g.draw(new Line2D.Float(100, 100, 200, 200));
```

the endpoints of the `Line` we construct are given in what is called *user space*. You should think of this as a convenient, but abstract, coordinate system that might not coincide with `device space`. However, a `Graphics2D` object maintains an `AffineTransform` that converts the definition of `Shapes`, given in user space, into the objects to be rendered in device space.

Initially, this transform is defined so that the origin of user space corresponds with the upper left corner of device space and one unit represents one pixel if we displaying the illustration on a computer screen or one point if we are printing onto a piece of paper.

However, `Graphics2D` has methods that allows us to modify this transformation. For instance, if we define an `AffineTransform` called `at` and then say `g.transform(at)`, the `Graphics2D`'s `AffineTransform` is composed with `at`. For instance, we could draw the same, well almost the same, picture as above as follows:

```
AffineTransform at = new AffineTransform();
at.translate(150, 200);
at.rotate(-Math.PI/3);
at.scale(2, 0.5);
g.transform(at);
Ellipse2D.Float circle = new Ellipse2D.Float(-50, -50, 100, 100);
g.draw(circle);
```



Notice that there is an important difference in the figures: when we modify the `Graphics2D`'s internal transform, the ellipse is not draw with a uniform thickness. In this situation, *everything* is transformed including the curve that is drawn to represent the ellipse. While this can be a desirable effect at times, it is usually not what we want. In addition, it is often useful, say when we want to place a label next to a point, to be able to work directly with pixels. Therefore, we will typically favor the

first method and transform individual `Shapes`. Later in this chapter, we will explain why modifying the `Graphic2D` has this effect.

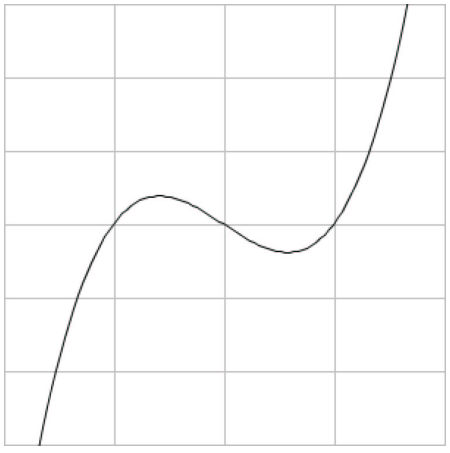
The inverse of an `AffineTransform` may be obtained with the `createInverse` method. However, we need to use a bit of care when doing this. Not every affine transformation is invertible so it is possible that the inverse does not exist. If this is the case, Java will raise an `Exception` to notify us and we must *catch* the exception. We haven't seen how to do this yet, but here is a block of code that works:

```
AffineTransform inverse;
try {
    inverse = transform.createInverse();
} catch(NoninvertibleTransformException ex) {
    do something here if transform is not invertible
}
```

One final note: it is possible to set the `Graphics2D`'s internal transform through its `setTransform` method. There is some danger in doing this, however, since device space may be represented differently on different systems. Generally speaking, `setTransform` should only be used after first retrieving the transform using `getTransform`. In this way, we may change and then restore the transformation.

3: Graphing a function

Let us now construct a simple mathematical illustration that illustrates how `AffineTransforms` can be used. In a frame 300 pixels wide and 300 pixels hight, we will graph the function $y = x^3 - x$ in a window where $-2 \leq x \leq 2$ and $-3 \leq y \leq 3$. We will also include a 1×1 grid in the background drawn in a light gray.



First, we construct an `AffineTransform` to convert the mathematical coordinate system into the `JPanel`'s coordinate system. We wish for the origin of our new coordinate system to be at the center of the frame. For convenience, we will scale the y coordinate by -1 so that the coordinate increases as we

move up. Finally, we scale both coordinates so that one horizontal unit corresponds to 75 pixels while one vertical unit is 50 pixels.

```
AffineTransform transform = new AffineTransform();
transform.translate(150, 150);
transform.scale(1, -1);
transform.scale(75, 50);
```

Next, we construct the grid, transform it and draw it.

```
g.setPaint(Color.lightGray);
GeneralPath path = new GeneralPath();
for (int i = -2; i <= 2; i++) {
    path.moveTo(i, -3);
    path.lineTo(i, 3);
}
for (int i = -3; i <= 3; i++) {
    path.moveTo(-2, i);
    path.lineTo(2, i);
}
g.draw(transform.createTransformedShape(path));
```

Finally, we construct the graph of the function, transform it and draw it.

```
int steps = 50;
float dx = 4.0f/steps;
g.setPaint(Color.black);
path = new GeneralPath();
path.moveTo(-2, valueAt(-2));
for (float x = -2+dx;  x <= 2; x += dx)
    path.lineTo(x, valueAt(x));
g.draw(transform.createTransformedShape(path));
```

The method `valueAt` returns the value $x^3 - x$ when given x. You will see that we are approximating the graph by a series of short line segments.

This is an important example, and we will return to it later to make it more generally usable.

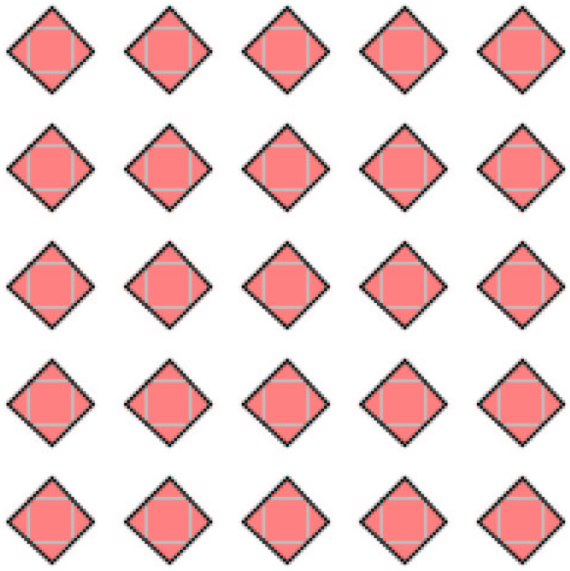
Exercise 1: Using `AffineTransforms`, draw a regular pentagon with sides of length 50 pixels, one side horizontal and the opposite vertex at $(100, 50)$.

Exercise 2: Redraw the figure from Chapter 3 explaining the arithmetic series and the sum of cubes using `AffineTransforms`.

Exercise 3: Fill a 300×300 `JPanel` with a tiling by congruent equilateral triangles whose sides are 50 pixels wide.

Exercise 4: Modifying the rendering engine's affine transformation is a technique familiar to PostScript programmers. In fact, PostScript can store and recall a sequence of transformations through the `gsave` and `grestore` commands. Since Java2D does not give this ability explicitly, it is tempting to mimick it using a `Stack`. However, my experience is that this is not always the most natural way to proceed. Draw the following figure in both PostScript and Java2D trying to find a comfortable way to manipulate

the necessary transformations. Notice that the pink area is filled first, then the gray square drawn and finally the black outline.



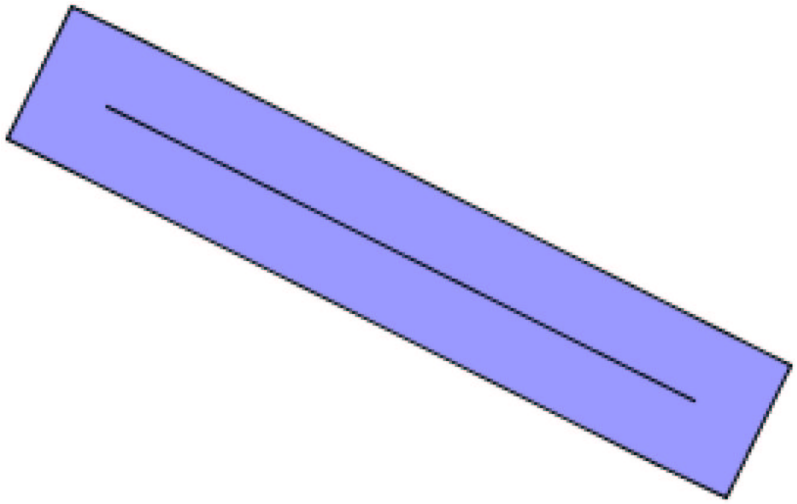
4: Stroking paths

Let us now investigate some options we have when we `draw` a `Shape`. For instance, if we `draw` a `Line2D`, we can specify how thick we want the line to appear by specifying the `Stroke` used. More generally, the `Graphics2D` maintains a current `Stroke` object from which it seeks information when drawing shapes. We can easily modify the `Stroke` through the `setStroke` method of `Graphics2D`.

For instance, if we want to draw a line 10 units wide, we can say

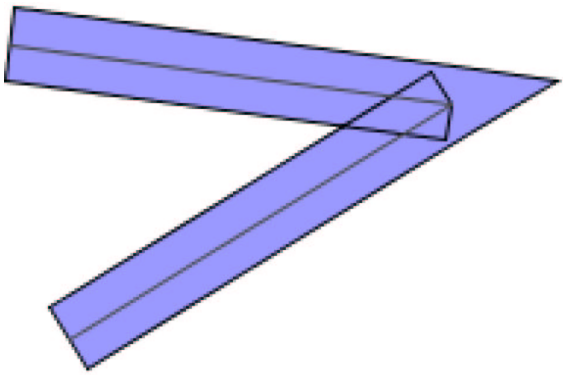
```
BasicStroke stroke = new BasicStroke(50);
g.setStroke(stroke);
g.draw(new Line2D.Float(50, 50, 250, 150));
```

The units in which the line thickness are given are interpreted in *user* space just as are the coordinates of the line's endpoints. The rendering engine then lays out the line with its appropriate thickness in user space. It may look something like this:

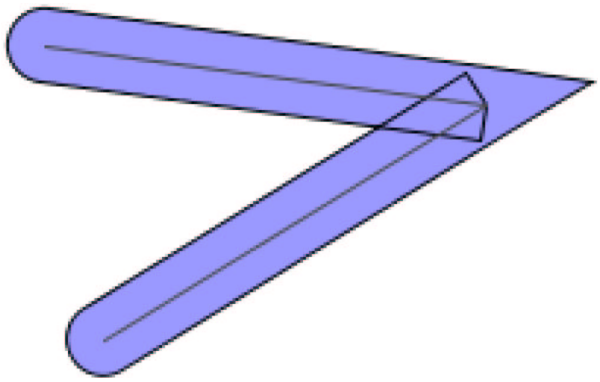


in user space. If the `Graphics2D`'s transformation has been modified, it is applied to this figure and if the transformation results from a non-uniform scaling, then the thickness of a line will depend on the angle at which it is drawn. The interior of the region is then filled. In other words, `draw` is really just a special case of `fill`.

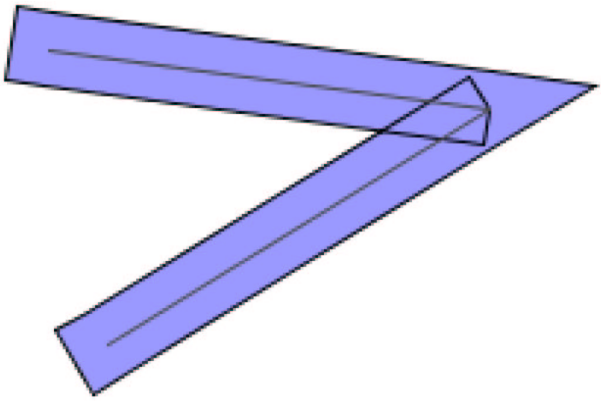
There are a few other options we can specify in `BasicStroke`: how the ends of segments are displayed, the sharpness at which corners are drawn and whether the line should be dashed or not. Here are some possibilities:



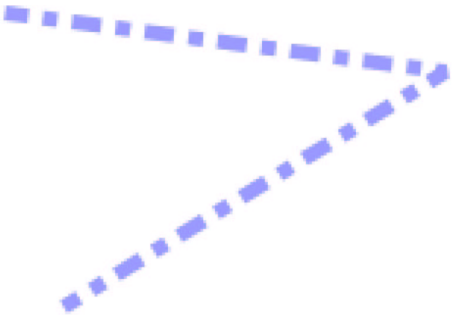
```
BasicStroke stroke = new BasicStroke(25f, BasicStroke.CAP_BUTT,
                                     BasicStroke.JOIN_MITER);
```



```
BasicStroke stroke = new BasicStroke(25f, BasicStroke.CAP_ROUND,
                                     BasicStroke.JOIN_MITER);
```



```
BasicStroke stroke = new BasicStroke(25f, BasicStroke.CAP_SQUARE,
                                     BasicStroke.JOIN_MITER);
```



```
BasicStroke stroke = new BasicStroke(5f, BasicStroke.CAP_BUTT,
                                     BasicStroke.JOIN_MITER, 1f,
                                     new float[] {10, 5, 5, 5}, 2);
```

The array of `floats` in this example give alternately the length of a dash and the space between this and the next dash. The final argument, known as the *offset*, tells where in the cycle to begin.

Exercise 5: Draw the graph of $x^3 - x$ stroked with a line of thickness 2 pixels and add the derivative stroked with a dashed line of thickness 1 pixel.