CS 355 Lab #2: Interacting with Drawing

Overview

This lab builds on the previous one by introducing selection, moving, and rotating. To do it you will need to use selection tests and both object-to-world and world-to-object transformations.

User Interface

Selection and Operations on Selected Shapes

- Clicking within four pixels of a line should select the line. Clicking inside any other shape should select the shape.
- Once a shape is selected you should highlight it by drawing the border of the shape (other than lines). You should also draw round or small square handles at the ends of the line, at the corners of the triangle, or at the corners of the bounding boxes for the other shapes. (These will be used for resizing the shape in the next lab, but for this lab they're just for indicating a selection. You may omit the corner handles for now as long as you visually indicate in some other manner that the shape is selected.)
- For shapes other than lines, you should also draw an additional handle for rotation. (If you wish, you may omit the rotation handle for circles for obvious reasons.)
- While a shape is selected, you should change the current color indicator to the color of the object. Clicking on that indicator and selecting a different color should change the color of the currently selected shape as well as the indicator.
- Selecting another shape should deselect the current one, as should clicking outside of any existing shape.
- Clicking on the currently selected object's handle should take precedence over selecting a new object. That is, if a handle of the currently selected object appears inside another shape, clicking on it should manipulate the current shape through its handle rather than selecting the other shape.

Moving Shapes

- Clicking within a shape and dragging the mouse should move the shape accordingly. The object itself should be drawn continuously as it is moved, with the shapes behind it redrawn as needed. The amount the object moves should be the same as the mouse moves. *The shape should not "jump" when initially clicked and dragged.*)

Rotating Shapes

- Clicking on and dragging the rotation handle of the current shape should rotate the shape accordingly. The object should be drawn continuously as it is rotated, with the shapes behind it redrawn as needed.

Model

The first thing you need to do to separate the object-to-world transformation parameters (location, orientation) from the object-space representation stored in each subclass. (The exception to this will be for lines, as explained shortly.)

The Shape Class

You should change your model so that the parent Shape class now includes not only the color of the shape but a center position and a rotation angle. For each shape, you will need to compute its center, and when that shape moves you will need to update that accordingly. As before, these should also have accessor (get and set) methods.

All of the shape subclasses except for lines should now be represented in a native object space as follows.

Lines

You should store the two endpoints of the line in world space. You may, if you choose, represent it in an object-space (which would just be the length) and store the position and angle in the transformation, but this is not required because of the diminutive nature of simple lines.

Rectangles

You should store the height and width of the rectangle. (Alternatively, you may store the half-height and half-width if you prefer.) The rotation angle should be initialized to 0 radians.

Squares

You should store the size (or half-size) of the square. The rotation angle should be initialized to 0 radians.

Ellipses

You should store the height and width of the bounding rectangle. (Alternatively, you may store the half-height and half-width if you prefer.) The rotation angle should be initialized to 0 radians.

Circles

You should store the radius of the circle. The rotation angle should be initialized to 0 radians.

Triangles

The Triangle class should store the location of each of the three corner points *relative to the center of the shape*. For a triangle, the center is the average of the coordinates for the three corners in world space. The rotation angle should be initialized to 0 radians.

Notes

As before, the model should be independent of the user interface (the view / controller). Specifically, it should not know at all about handles, mouse events, etc. — those are all part of the interface, not the underlying geometric model.

Geometry Tests

Add a method to your model class that takes a point in world coordinates (which for this assignment are the same as screen coordinates) and a selection tolerance. When this method is called the model should then test each shape in the model, from front to back, to see if that point falls within that shape. For lines, it should test to see if distance from the point to the line is within the desired tolerance. This approach puts all the geometry testing in the model itself. Try to keep the tests generic in the spirit of model independence: a "point in shape" test, find the first hit, etc.

Implementation Notes

This lab uses the same program shell (helper classes) as Lab #1. The first thing you should do is to rework your code from the previous lab to work with the new model, particularly the separation between the object-to-world transformation parameters and the object-space representation. You should proceed to the other portions only after you have made sure the functionality of the previous lab works.
Note: some of the terms and notation used in the rest of this section draws on material we may not have covered in class by the time the lab is assigned. Go ahead and get started, and we'll cover the rest of what you need soon.

Drawing the Shapes

Java's 2D Graphics API supports drawing shapes with what are called *affine transformations*, which include translation, rotation, scaling, and skewing. In particular, Java has its own `AffineTransform` class.

For each object you'll need to either build an object-to-world transformation $\mathbf{O}_i$ that convert from the object's coordinate space to the world (drawing) coordinate space as follows:

$$\begin{aligned} \mathbf{p}_w &= \mathbf{O}_i \, \mathbf{p}_o && (1) \\ &= \mathbf{R}_{\theta_i} \mathbf{p}_o + \mathbf{c}_i && (2) \end{aligned}$$

where

$$\mathbf{O}_i = \mathbf{T}_{\mathbf{c}_i} \mathbf{R}_{\theta_i} \qquad (3)$$

where $\mathbf{T}_{\mathbf{c}_i}$ denotes translation by offset $\mathbf{c}_i$, and $\mathbf{R}_{\theta_i}$ denotes rotation by angle θ_i.

Suppose that a particular rectangle was centered at $(100, 50)$ and rotated by an angle of $\pi/4$, with height `height` and width `width`. You can draw this shape onto the `Graphics2D` object `g` with code like this:

```
// create a new transformation (defaults to identity)
AffineTransform objToWorld = new AffineTransform();

// translate to its position in the world (last transformation)
objToWorld.translate(100,50);

// rotate to its orientation (first transformation)
objToWorld.rotate(Math.PI / 4);

// set the drawing transformation
g.setTransform(objToWorld);

// and finally draw
g.fillRect(-width/2,-height/2,width,height);
```

Important: Java applies the transformations in the reverse order in which you specify them (call the respective methods). The above example will rotate *then* translate even though the method calls are made in the other order. We'll talk later in the class about why many graphics systems do this.

Selection

For lines, you may code the distance-to-line test in world coordinates either whatever geometric approach you choose.

For all other shapes, the first thing you need to do in order to test to see if a point is within a particular shape is to convert that point's coordinates in world space $\mathbf{p}_w$ to its coordinates $\mathbf{p}_o$ in the object's space. To do this, you can again build an `AffineTransform` object that maps points in world space to points in object space. Rather than drawing with it, for selection you'll use its `transform` method, which maps points in one space to points in another.

$$\mathbf{p}_o = \mathbf{R}_{\theta_i}^{-1}(\mathbf{p}_w - \mathbf{c}_i) \quad (4)$$

$$= \mathbf{O}_i^{-1}\,\mathbf{p}_w \quad (5)$$

For the example rectangle used earlier for drawing, you can build the world-to-object transformation and then apply it like this:

```
// create a new transformation (defaults to identity)
AffineTransform worldToObj = new AffineTransform();

// rotate back from its orientation (last transformation)
worldToObj.rotate(- Math.PI / 4);

// translate back from its position in the world (first transformation)
worldToObj.translate(-100,-50);
```

```
// and transform point from world to object
worldToObj.transform(worldCoord, objCoord);
```

Again, Java will apply the transformations in the reverse order in which you specify them.

Note that this example assumes that the `Point2D` object `worldCoord` already contains the point in world coordinates, and the `Point2D` object `objCoord` has already been allocated. After this runs, the result will be stored in `objCoord`.

Once in object space, you'll find most of the tests to be pretty straightforward. *You must do the point-in-shape tests yourself in object coordinates. You may not use any point-in-shape tests built into Java.*

Moving Shapes

Other than lines, moving a shape should change its location (center), not its object-space representation. This, of course, also affects its object-to-world transformation $\mathbf{O}_i$.

Rotating Shapes

Other than lines, rotating a shape should change only its rotation angle, not its object-space representation. This, of course, also affects its object-to-world transformation $\mathbf{O}_i$.

Other Notes

You'll find that it's best to work in double-precision arithmetic as much as possible. Java's GUI API uses the integer-based `Point` class. You should convert to and store double-precision `Point2D` objects wherever possible.

Submitting Your Lab

To submit this lab, again zip up your src directory to create a single new `src.zip` file, then submit that through Learning Suite. If you need to add any special instructions, you can add them there in the notes when you submit it.

Rubric

This is tentative and may be adjusted up to or during grading, but it should give you a rough breakdown for partial credit.

- Redoing the functionality from Lab #1 using the new model that separates the object's position and orientation from its specific shape properties, including correct model-view-controller separation (10 points)
- Selecting shapes, including indicating selection and drawing handles (36 points total)
  - Lines (10 points)
  - Rectangles (5 points)

  - Squares (3 points)
  - Ellipses (5 points)
  - Circles (3 points)
  - Triangles (10 points)

- Changing color once selected (4 points)
- Moving by clicking and dragging (10 points)
- Rotating and drawing rotated (10 points)
- Selecting while rotated (10 points)
- Generally correct behavior otherwise (10 points)
- Correct use of world-to-object and object-to-world transformations (10 points)

TOTAL: 100 points

Note that this point distribution relates less to the amount of code you have to write for each of these and more to the level of difficulty and demonstration of mastering the concepts we've been learning in class.

Change Log

- September 19: initial version posted