CS 355 Lab #2: Interacting with Drawing

Overview

This lab builds on the previous one by introducing selection, reording, 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

This lab adds new commands; it doesn't remove any. The shell still has all of the commands from the previous lab, and you should still implement them.

Important: In this spec, "clicking" refers to *pressing* the left mouse button. All of your interactions will key off of the mouse being pressed rather than released. The Java function that deals with this, and that you need to implement, is the `mousePressed` method found in class `MouseListener`.

The items below are things you need to implement.

Selection

- 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.
- 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.
- 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.

Operations on Selected Shapes

- Selecting a different color should change the color of the currently selected shape as well as the indicator.
- Choosing "`Delete Shape`" from the "`Edit`" menu should delete the currently selected shape. No other shape should be selected.

- You should also implement the following four menu items in the "Object" menu: "`Move Forward`", "`Move Backward`", "`Send to Front`", and "`Send to Back`". The actions for these menu commands should be self-explanatory, but for all four of them, the currently selected shape should remain selected.

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.

Changing Line Endpoints

- Clicking on a line endpoint handle and dragging it should move the endpoint accordingly. The line itself should be redrawn continuously as it is changed, with other shapes behind it redrawn accordingly as needed.

Model

The first thing you need to do in the model is to separate the object-to-world transformation parameters (location, orientation) from the object-space representation stored in each subclass. You can download the `drawing.zip` file to get the new versions of the `Shape` class and its subclasses.

Note: because the model is different, drawings saved with Lab #1 are incompatible with those for Lab #2.

Shape

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. (This version of the `Shape` superclass is provided.) For each shape, you will need to compute its center, and when that shape moves you will need to update the center accordingly. As before, you should provide accessor methods (get and set) for the center point of the shapes.

All of the shape subclasses should now be represented in a native object space as defined in the following sections.

Lines

Lines are a bit different, and so to keep consistency with the other shapes, we will be storing them in a slightly unintuitive manner. You should store one endpoint of the line in the `center` member variable of the `Shape` class, and the other endpoint as a member variable of the `line` subclass that you define.

Rectangles

You should store the height and width of the rectangle. The rotation angle should be initialized to 0 radians.

Squares

You should store the 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. 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.

If you do this approach, try to keep the tests generic in the spirit of model independence: a "point in shape" test, find the first hit, etc.

Alternatively, you can choose to let the model contain only geometric data and then have the controller do the geometric selection tests.

Either of these approaches works for this assignment. For a larger system you might build both the geometry representation and computation on this geometry into the model while letting the controller focus on the conversion between the user interaction and this geometry engine. For a simpler system like this one, you might let the model be a data store only and then put the selection tests in the controller.

Implementation Notes

This lab uses the same program shell (helper classes) as Lab #1. However, you can get the new version of the `Shape` class and its subclasses by downloading the `drawing.zip` file from Learning Suite.

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 build an object-to-world transformation $\mathbf{O}_i$ that converts 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

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

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 using 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 world space coordinate ($\mathbf{p}_w$) to the corresponding point in the shape's object space ($\mathbf{p}_o$). 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.

$$\begin{aligned} \mathbf{p}_o &= \mathbf{R}_{\theta_i}^{-1}(\mathbf{p}_w - \mathbf{c}_i) && (4) \\ &= \mathbf{O}_i^{-1}\,\mathbf{p}_w && (5) \end{aligned}$$

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

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

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

Changing Line Endpoints

Moving a line's endpoint handle should change the corresponding endpoint, then refresh the drawing area. First, test to see if a mouse-down event occurs at one of the selected line's handles. *Make sure to remember which handle is being moved.* Then update the moved endpoint.

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.

Restrictions

The same restrictions apply to the new `Shape` class and its subclasses as applied in Lab 1.

Submitting Your Lab

To submit this lab, zip up your src directory to create a single new `src.zip` file, then submit that through Learning Suite. We will then drop that into a NetBeans project, copy in the originals for any files you are not allowed to modify (see the Restrictions section of the Lab #1 specification), do a clean build, and run it there.

If there are compile errors in your project because you modified files that you were not allowed to modify, we will grade your lab accordingly.

If you need to add any special instructions, please do so in the notes when you submit it. If you use any external source files, please make sure to include those as well.

Rubric

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

- Correct separation of the object's position and orientation from its specific shape properties, and use of world-to-object and object-to-world transformations (10 points for doing this while implementing only the functionality from Lab #1)
- Selecting shapes, including indicating selection and drawing handles (26 points total)
  - Lines (5 points)
  - Rectangles (5 points)
  - Squares (3 points)
  - Ellipses (5 points)
  - Circles (3 points)

  - Triangles (5 points)
- Changing color once selected (4 points)
- Reordering and Deleting Shapes (10 points)
- Moving by clicking and dragging (10 points)
- Rotating and drawing rotated (10 points)
- Moving line endpoints (10 points)
- Selecting shapes while rotated (10 points)
- Generally correct behavior otherwise (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.
