CS 355 Lab #2: Interacting with Drawing

September 30, 2013

Overview

This lab builds on the previous one by introducing selection, moving, rotating, and reshaping.

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.
- 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.)
- 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.
- 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.

Rotating Shapes

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

Changing Shapes

- Clicking on an endpoint or corner handle and dragging it should cause the shape to change accordingly. The object itself should be drawn continuously as it is reshaped, with the shapes behind it redrawn accordingly as needed.
  - For lines, dragging an endpoint's handle should simply move that endpoint.
  - For triangles, dragging a corner's handle should similarly move that corner.
  - For the other shapes, all of which use bounding box corners for their handles, moving a corner should keep the opposite corner in place while changing the bounding box to reflect that and the moved corner.

Model

The first thing you will 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 translation offset and a rotation angle. For each shape, you will need to compute its center, and when that shape moves or changes shape 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 of the square. (Alternatively, you may store the half-size if you prefer.) 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

You may choose to 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. 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 notation used in the rest of this section draws on material we won't 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 affine transformations. (If you use this, make sure to reset the transformation to the identity prior to setting the transformations for subsequent shapes; otherwise, the transformations will build up.) Since we're only using translation and rotation, these easily fit within the class of affine transformations.

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

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

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.

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'll either reverse the process in Eq. 1 or use the inverse of the object-to-world transformation $\mathbf{O}_i$ in Eq. 3. You may either compute and apply the transformation(s) or use Java's `AffineTransformation` class to store the transformation and apply it to points. But either way, *you will need to do the point-in-shape test yourself in object coordinates*.

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

Once in object space, you'll find most of the tests to be pretty straightforward.

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 $\mathbf{O}_i$.

Changing Shapes

Moving a shape's handle should change the shape's object-space representation (and possibly its translated position), then refresh the drawing area. You'll find that this is the trickiest part of the assignment, but if you keep the coordinate spaces straight you'll do fine.

First, test to see if a mouse-down event occurs at one of the selected object's handles. As before, use Eq. 4 or Eq.5 to convert from the world coordinates of the mouse-down to the object's coordinates. Once in this space, testing against the position of the handles is relatively straightforward. *Make sure to remember which handle is being moved.*

As you drag the handle, you'll get a new position of the endpoint / corner *in world coordinates*. If you convert this from world to object coordinates, you'll now have the new corner in that space. Then you update the representation and the object-to-world transformation accordingly.

Lines

For lines, just update the moved endpoint in world coordinates.

Triangles

Only one of the corners moves at a time *in world space*, but moving one of the corners means that you have to update the center, which means you have to update the relative offsets from the center for all points accordingly. This will take some thinking through, so be careful with it.

Java's drawing routines don't care about the order, so don't worry about turning the shape inside-out, so to speak. For other drawing packages, or for your own geometric tests (depending on which method you use), you might have to put the points back into clockwise or counterclockwise order.

Other Shapes

First compute the location of the corner opposite the one being moved. After getting the new location of the moved corner, use its position and the position of the (unmoved) opposite corner to rebuild the shape (height/width, size, or radius) accordingly. Remember to update the center as well.

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 (30 points total)
  - Lines (8 points)
  - Rectangles (4 points)
  - Squares (2 points)
  - Ellipses (4 points)
  - Circles (2 points)
  - Triangles (10 points)
- Changing color once selected (2 points)
- Moving by clicking and dragging (5 points)
- Reshaping while not rotated (10 points)
- Rotating and drawing rotated (5 points)
- Selecting while rotated (5 points)
- Reshaping while rotated (8 points)
- Generally correct behavior otherwise (5 points)

TOTAL: 80 points

Roughly breaking this down, you can get up to 47 points (a bit over half credit) for a base implementation that includes revising the model, implementing all the selection tests, recoloring shapes, and otherwise having correct behavior. Adding clicking and dragging of shapes adds another 5 points (the hard part here is the selection test), and reshaping using the handles while the shape is not rotated adds another 10 points. This means you can get up to 62 points (just over 3/4 credit) without implementing rotation. Supporting rotation for drawing, selecting, and reshaping adds the final 18 points for full credit.

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 20: initial version
- September 24:
  - revised to allow selection tests in either the model (generically) or the controller
  - greater clarification in the user notes of either building your own transformations or using Java's
  - minor clean-up
- September 30:
  - added clarification about not needing rotation handles for lines
  - added clarification about rotation handles for circles being optional
  - added clarification on needing to reset Java's drawing transformations between each shape
  - improved the description of the mathematical steps for drawing and selection (with appropriate references to these equations added or changed elsewhere)
  - added rubric