CS 355 Lab #3: Viewing Transformations

Overview

This lab builds on the previous ones by introducing scrolling and zooming. It also has you do the transformations you did last time by actually building matrices for affine transformations and applying them.

User Interface

All functionality from the first three labs should still work for this lab. Here is the external functionality you should add to the program:

- You should add functionality to support the horizontal and vertical scroll bars in your window. This will include updating the viewing transform and initiating a refresh of the drawing area.
- You should also add functionality for the "Zoom In" and "Zoom Out" buttons, which should enlarge or reduce the size of the displayed objects by a factor of two up to a maximum of 400% and a minimum of 25% zoom. The maximum drawing area (2048×2048) is what would be visible when zoomed out to 25% (i.e., the maximum viewable area). This will also include updating the view transform and initiating a refresh of the drawing area.
- All of the user's drawing in the viewport should be at the current scroll position and level of zooming, then converted to the world-space drawing area. That is, if you draw a shape at 50% zoom and then go back to 100%, the object should be twice as large as what you drew it at. Similarly, if you draw it in the center of the viewport but the viewport is scrolled over, the object should stay at that position in the world-space drawing area and move accordingly on the screen.
- When zooming in or out, the center of the viewport should stay at the current position in the world space unless doing so would take the viewed area outside the maximum drawable area.
- The interface elements for selected objects (bounding shapes or boxes, handles, etc.) should be independent of the level of zoom. That is, they should be drawn consistently at the same size regardless of how large or small the drawn shapes are scaled. That means the clickable area for a handle should also stay constant. (Remember that these interface elements are part of the view/controller, not the model, so you have access to the current zoom level when drawing or clicking on them.)
- Similarly, the tolerance for the user to select lines should stay constant in screen space, not in world or object space. As discussed in class when discussing drawing lines and allowing such tolerance, the user shouldn't have to be more precise when zoomed out.

Model

The model should stay unchanged from the previous assignment, and all viewing transformations should be handled in the controller and viewer.

Remember that all interactions in the model should already be in *world coordinate space* (i.e., in terms of a global drawing area), so no changes should be required. If you did mix some notions of what's happening on the screen into the model, you may make changes to correct that.

One exception: if you implemented the point-in-shape or point-near-line tests in the model in terms of world coordinates, the threshold for "near enough" to a line should scale with the viewing so that the on-screen tolerance stays constant regardless of the zoom level. (Suggestion: have the controller pass the tolerance as a parameter to the "near enough to the line" test and scale it according to the zoom level.)

View / Controller

In general, everything related to the display of the data (scrolling and zooming) rather than the data itself should reside in the view / controller combination, never in the model.

For this lab, the viewer should draw with appropriate translation and scaling based on the current scrolling and zoom settings. Correspondingly, the controller should appropriately convert the on-screen positions of mouse events to account for these settings.

Implementation Notes

This lab uses the same program shell (helper classes) as Labs #1 and #2.

Transformations

For this lab, you must implement all transformations as individual `AffineTransformation` objects, and *you must calculate the matrix entries yourself*. Each `AffineTransformation` should be constructed by passing it the six entries of the matrix, not by making individual calls to apply rotation, scaling, or translation to the transformations. *You may only build transformations by passing in the entries for the matrix or by composition of transformations you've already built.*

Once built, you should again use them to transform drawing onto the `Graphics2D` for the displayed drawing area or to `transform` a point or set of points.

Drawing

You should draw with an affine transformation that converts from object coordinates to viewport coordinates. This per-object drawing transformation $\mathbf{M}_i$ is simply a compositing of the respective object-to-world transformation and the world-to-viewport transformation:

$$\mathbf{M}_i = \mathbf{V}\ \mathbf{O}_i \tag{1}$$

You can compose matrix transformations using the `AffineTransformation` class's `concatenate` method.

Object to World

The object-to-world transformation is the same as what you did when drawing in the previous lab: rotate first, then translate to the object's position in the world:

$$\mathbf{O}_i = \mathbf{T}(\mathbf{c}_i)\ \mathbf{R}(\theta_i) \tag{2}$$

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

You can use this transformation to convert from world to viewport coordinates:

$$\mathbf{p}_{\text{world}} = \mathbf{O}_i\ \mathbf{p}_{\text{object}} \tag{3}$$

World to View

You should similarly construct a viewing transformation $\mathbf{V}$ that converts between world space and the viewport space. It should scale the coordinates accordingly as the zoom level changes and translate the coordinates according to the scrolling bars.

$$\mathbf{V} = \mathbf{S}(f)\ \mathbf{T}(-\mathbf{p}) \tag{4}$$

where $\mathbf{S}(f)$ denotes scaling by the current scaling factor f, and $\mathbf{T}(-\mathbf{p})$ is a translation from the position $\mathbf{p}$ (in world coordinates) of the origin of the viewport's coordinate system to the origin of the world coordinate system.

You can use this transformation to convert from world to viewport coordinates using V:

$$\mathbf{p}_{\text{view}} = \mathbf{V}\ \mathbf{p}_{\text{world}} \tag{5}$$

Doing the Drawing

For drawing, you should again pass this to the `setTransformation` method of the `Graphics2D` object you're drawing to. For example, if the `Graphics2D` object passed to your viewer is `g`, and the composite transformation $\mathbf{M}$ you have computed (Eq. 1) is `objectToScreen`, this would look like this:

```
g.setTransformation(objectToScreen);
```

As before, you then draw in object space (other than for lines). This transformation should handle *both* the object-to-world and world-to-view transformations.

Selecting

To select shapes, you will need to convert from screen coordinates to world coordinates, and then from world coordinates to each shape's object coordinates. You can, of course, do this with a single transformation that is the inverse of the matrix $\mathbf{M}_i$ you already calculated. Rather than inverting this matrix, you can compute it directly by reversing the steps and undoing each step:

$$\mathbf{M}_i^{-1} = \mathbf{O}_i^{-1}\ \mathbf{V}^{-1} \tag{6}$$

From Eq. 2,

$$\begin{aligned} \mathbf{O}_i^{-1} &= \mathbf{R}^{-1}(\theta_i)\ \mathbf{T}^{-1}(\mathbf{c}_i) \\ &= \mathbf{R}(-\theta_i)\ \mathbf{T}(-\mathbf{c}_i) \end{aligned}$$

and from Eq. 4,

$$\begin{aligned}\mathbf{V}^{-1} &= \mathbf{T}^{-1}(-\mathbf{p})\,\mathbf{S}^{-1}(f) \\ &= \mathbf{T}(\mathbf{p})\,\mathbf{S}(1/f)\end{aligned}$$

To transform points using the matrix $\mathbf{M}_i^{-1}$, again use the `AffineTransformation` class's `transform` method. For example, if the transformation $\mathbf{M}_i^{-1}$ is stored in an object called `screenToObject`, and you want to transform the point `screenPoint` (in viewing coordinates) to the point `objectPoint` (in object coordinates), you would do this:

```
screenToObject.transform(screenPoint,objectPoint);
```

Tip: For testing, you might want to try first transforming the screen-space point to world coordinates and then look at the values to make make sure that part is working. Then transform that from world to object coordinates and check it again. Once you're confident everything is working, concatenate the transformations.

Managing the Scroll Bars

Your controller will receive notifications telling you that the user has dragged a scroll bar. If you set the size of the scroll bar to the size of the drawing area and the size of the "knob" to the size of the viewable area of the drawing (depending on the level of zooming), the position of the scroll bar should be the same as the position of the upper left corner of the viewed area, which you can use to build the viewing (world-to-view) and inverse viewing (view-to-world) transformations.

As the user zooms in or out, you will need to change the position of the scroll bars and the size of the knobs accordingly, which you can do using the functions provided in `GUIFunctions`. Be aware that changing the position of the scroll bar will cause Java to notify all of the scroll bar's listeners that it changed. This includes you, so you may want to use a flag to ignore these "events" while making multiple changes so that your screen doesn't make multiple refreshes while you're making these changes.

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.

- Correctly drawing while zoomed in or out (10 points)
- Correctly drawing while scrolled (10 points)
- Correctly selecting while scaled (10 points)

- Correctly selecting while scrolled (10 points)
- Correctly implementing all transformations ($\mathbf{M}_i$, $\mathbf{V}$, $\mathbf{O}_i$, $\mathbf{T}(\mathbf{c}_i)$, $\mathbf{R}(\theta_i)$, $\mathbf{S}(f)$, $\mathbf{T}(\mathbf{p})$, and their inverses) as `AffineTransformation` objects created either by *you* giving it the matrix elements or by composition (40 points)
- Otherwise generally correct behavior, including correct scroll bar behavior, zooming while staying centered at the same position in the drawing area, etc. (20 points)

TOTAL: 100 points

Note: The rubric is constructed so that you can get up to 60% credit for implementing all of the external functionality correctly by using Java's built-in rotation, translation, and scaling transformations. To get full credit, you must implement these yourself by providing the elements for the respective matrices.

Change Log

- May 13: initial version for Spring 2015