CSUS
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
Fall 2012
John Clevenger

Assignment #4: Transformations, Viewing, & Curves

Due Date: Friday, December 7th [2 weeks]

Overview

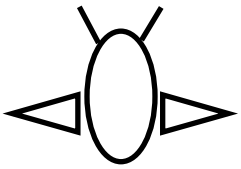
The objective of this assignment is to extend your program from Assignment #3 (A3) to include 2D modeling and viewing transformations, along with an extra-credit topic: Bezier curves. As always, all operations from previous assignments must continue to work as before (except where superseded by new operations). In this assignment, you are to add the following three things:

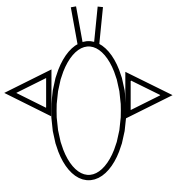
1. Local, world, and device coordinate systems. Objects are to be drawn in their "local" coordinates; transformations are then used to map drawn objects to "world" and then to "screen" coordinates. The game world is defined by an independent *world coordinate system*. Initially, the origin (0,0) position of the world coincides with the lower left corner of the screen (MapView area). A screen transformation is used so that the game world appears "right-side up". The initial (default) "world window" should match the screen size in your previous assignments, so that initially the entire game world is visible on the screen; the user has the ability to *ZOOM* and *PAN* to change what is displayed in the MapView area. Control over zooming and panning is done with mouse operations (see below).

2. Hierarchical and dynamic object transformations. Each *bird* is to be defined as a dynamically transformable hierarchical object (instead of as a single geometric shape or image). That is, a bird is to be composed of a hierarchy of at least two levels of shapes, each with its own transformation which positions the shape in relation to its parent shape. At least one of the shape transformations must change dynamically as a function of the timer during the game as it moves (for example, the "wings" could flap back and forth and get larger/smaller as seen from above).

As an example, the bird could be constructed as shown in the following figure, with the triangles moving back and forth along the body while also being scaled larger and smaller over time. The pictures also show "legs" which move as the bird flies. (This is just an example; you may use another form for the bird as long as it is hierarchical and meets the above requirements.)

Bird at time *t*





Bird at time *t+n*

3. Bezier Curves (Extra Credit). Vehicles (cars and trucks) should display a "decoration" in the form of a unique cubic Bezier curve. The beginning and ending points (control points P0 and P3) for the curve should be two randomly-chosen vehicle corners. The other two (interior) control points should be randomly chosen points lying within the vehicle boundary. You may use the *Decorator* design pattern to accomplish the decoration, but this is not a requirement (that is, you may instead simply hard-code the decoration into the **`draw()`** method of the vehicle).

The control points for decorated vehicles should be set when the vehicle is created, so that the curve is drawn the same way (for a given vehicle) on each repaint. In addition to drawing the curve itself, the draw routine must also draw three straight lines connecting the control points (in order) so

that the convex hull (bounding region) of the curve can easily be seen (the fourth line in the convex hull will be a vehicle side). You may choose the color scheme for drawing the curve and the convex hull lines. As with other output, the drawing routines for the curve should use *local coordinates* to draw the curve and its outline. Also, the drawing routine for the curve must use the recursive implementation (not the iterative implementation.)

Local Coordinates and Object Transformations

Previously, each object was defined and drawn directly in terms of its screen coordinates – that is, each object's **`draw()`** method used coordinates which were screen values. Now, each object should be drawn in its own *local coordinate system*, and should have a set of **`AffineTransform`** (**`AT`**) objects defining its *current transformation* (one **`AT`** each for translation, rotation, and scaling). This set of transformations specifies how the object is to be transformed from its local coordinate system into the world coordinate system (or into its *parent's* coordinate system in the case of hierarchical objects; see the "Fireball" example in the Lecture Notes).

For objects which do not move, the current transformation is set once – to the world position/orientation of the object – when the object is created. For moveable objects, each Timer tick is to invoke **`move()`** on the moveable object, as before. Instead of changing the object's position values, the **`move()`** method will now *apply a translation to the object's AT*. The amount of this translation is calculated from the elapsed time, speed, and heading, as before.

Previously, the **`draw()`** method for each moving object needed only to worry about drawing a simple object shape at the proper location. Now it also needs to apply the "current transformation" of the object so that the object will be properly drawn. This is done utilizing the mechanism discussed in class: the **`draw()`** method saves the current **`Graphics2D`** transform, appends the transformation of the object onto the **`Graphics2D`** transform, draws the object (and its sub-objects, in the case of a hierarchical object) using *local coordinate system* draw operations, and then restores the saved **`Graphics2D`** transform. That is, each **`draw()`** method temporarily adds its own object's local transformations to the **`Graphics2D`** transformation prior to invoking drawing operations, and then restores the **`Graphics2D`** transformation before returning.

All object drawing commands specify the object appearance in "local object coordinate space". That is, all drawing commands must be relative to the "local origin" (0,0). This is different from the previous assignment, where your **`draw()`** commands were relative to the "location" of the object. Now, the "location" is being set in the translation transformation added to the **`Graphics2D`** object prior to doing the actual drawing. For example, if you previously had a draw command like **`g.drawRect(xLoc,yLoc,width,height)`**, it becomes **`g.drawRect(0,0,width,height)`**, drawing in "local space" at (0,0) – which is then translated by the AT to the proper location.

In addition, hierarchical components which change dynamically should have the change applied via transformations. For example, movement of the hierarchical components of the bird (described earlier) should be applied via an AT.

World/Screen Coordinates

Your program must maintain a *Viewing Transformation Matrix (VTM)* which contains the series of transformations necessary to change world coordinates to screen coordinates. This VTM is then applied to every coordinate output during a repaint operation. The VTM is simply an instance of the Java **`AffineTransform`** class, named (for example) **`theVTM`**.

To apply the VTM during drawing, your MapView display panel's **`paintComponent()`** method should build an updated VTM and concatenate it into the **`AffineTransform`** of the **`Graphics2D`** object used to perform the drawing. **`paintComponent()`** then passes this **`Graphics2D`** object to the **`draw()`** method of each shape. As described above, each **`draw()`** method will then in turn temporarily add its own object's local transformations to that same **`Graphics2D`** transformation.

In order to build a correct VTM, the program uses the appropriate translate and scale operations to map the world coordinates onto a (hypothetical) Normalized Device (ND) and then onto the screen (**`JPanel`**), utilizing the current "world window" and "JPanel size" values to build the VTM as described in the Lecture Notes.

Zoom & Pan

Implementing zoom and pan operations is done by providing a way for the user to change the world window boundaries. Zoom is to be implemented by using the mouse *wheel*, such that *moving the mouse wheel forward zooms in* (decreases the world window size), and *moving the mouse wheel backward zooms out* (increases the world window size). Note that rotating the mouse wheel generates a **`MouseWheelEvent`** which is handled by listeners implementing the **`MouseWheelListener`** interface, analogous to how other mouse events are handled.

Pan is to be implemented using mouse *motion:* moving the mouse left/right *while holding down the SHIFT key* should cause horizontal panning and moving the mouse up/down should cause vertical panning. Note that *mouse motion* refers to moving the mouse *without holding down a mouse button (*mouse movement *with a mouse button down* is called a *drag).*

Mouse Input

A mouse event contains a **`Point`** giving the current mouse location *in screen coordinates*. However, when selecting objects (in pause mode), mouse input needs to determine *world* locations. Therefore, during selection the program must transform mouse input coordinates from screen units to world units. To do this, apply the *inverse* of the current VTM to the mouse coordinates (producing the corresponding point in the world). This world point can then be passed to the **`contains()`** method of each object, which converts the world point into local coordinates (by applying the inverse of the object's local transforms) and uses the result to determine whether the object contains the specified point.

Deliverables

Submit a single .zip file to SacCT containing source code, compiled (.class), and resource files (e.g. sounds and image files). Your program must be in a single package named "**`a4`**", with a main class named "**`Starter`**".

The due date for this assignment is Friday, December 7th at midnight. This is also the final date for submission of late assignments. This deadline applies to ***all*** assignments (not just A4). Assignments submitted after Friday, December 7th will not be graded. However, partial credit will be given for incomplete work submitted by the deadline.

JC:jc
11/22/12