CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
Fall 2015 – Dr. Muyan

Assignment #4: Transformations

Due Date: Monday Dec 14th

Overview

For this assignment you are to extend your program from Assignment #3 (A3) to include several uses of 2D transformations plus additional graphics and interactive operations. As always, all operations from previous assignments must continue to work as before (except where superseded by new operations). Specifically, you are to add the following things to your program:

1. Local, world, and screen coordinate systems

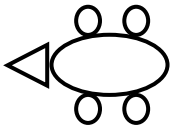
The game world is defined by an independent unbounded *world coordinate system*. Objects are to be drawn in their "local" coordinates. Local object transformations are to be used to map drawn objects from local to world coordinates, and then a Viewing Transformation Matrix [VTM] is to be used to map the contents of the *world window* to *normalized device* coordinates and then to *screen coordinates,* so that the world appears "right-side up" on the screen. Initially, the origin (0.0, 0.0) of the world coincides with the screen (MapView area) lower left corner.

Additionally, an appropriate rotation must be applied to Dogs, Cats, and ShockWaves (see below) so that they face the direction in which they are moving. The game also is to support *zoom* and *pan* operations to allow the user to see close-up or far-away views of the world. Control over zooming/panning is to be done with the mouse wheel and mouse drag.

2. Hierarchical and dynamic object transformations

You must define the Dog to be a dynamically transformable hierarchical object composed of a hierarchy of two levels of shapes. Each sub-object (i.e., second-level shape) of the Dog (i.e., top-level or parent shape) has its own transformation which positions the sub-object in relation to the origin (i.e., the center) of the Dog. The Dog also has its own transformations which positions itself in the world. In addition, at least one of the transformations of one or more sub-objects in the hierarchy must change dynamically as a function of the timer and this change should be visible to the player.

For example, a dog could be constructed from six sub-objects, "Body" (oval), "Head" (triangle), and four "Feet" (circles), as shown in the following figure:



To add a dynamic transformation, you can make the feet move in and out as the dog walks. But note that this is just an example; you may define the hierarchy and the movement in any way you like as long as it is hierarchical and changes dynamically as described above.

3. ShockWaves

ShockWaves are new moving objects which happen to be shaped like cubic Bezier curves. The game automatically generates a new ShockWave object each time the player "scoops" the animals in the net. A new ShockWave has an initial location equal to the net location, and a randomly-generated direction and speed (the speed should be slow enough that the ShockWave remains on the screen at least for a short time).

Collisions between ShockWaves and other objects have no effect; the ShockWave continues moving in the world. However, ShockWaves have a maximum lifetime after which they dissipate and are removed from the world. The maximum lifetime of a ShockWave should roughly the amount of time it takes the ShockWave to move all the way across the world window which exists at the time it is created (so that, for example, if the window is subsequently made significantly larger the disappearance of the ShockWave can be observed). See below for further constraints on ShockWaves.

Additional Details

Local Coordinates and Object Transformations

Previously, each object was defined and drawn using screen coordinates – the "location" of an object was a screen coordinate and the **`draw()`** method in each object 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 its parent's coordinate system (in the case of a sub-objects of the hierarchical Dog object) or into world coordinates (in the case of Dog, Cat, Net, and ShockWave).

For moving objects (Animals and ShockWaves), each Timer tick is to invoke **`move()`** on the object, as before. However, instead of changing the object's *position* values, the **`move()`** method will now *apply a translation to the object's "translation AT"*. The amount of this translation is calculated from the elapsed time, speed, and direction, as before. In addition, for the animals, the **`move()`** method will now apply a rotation to the *object's "rotation AT"* to change their direction (as indicated in A1 you should add or subtract small random values to animal directions so that they do not run in a straight line). For the net object (which is a guided object), the object's related transformation ATs (i.e., scale and translation ATs) should also be updated when the net is resized or moved.

Previously, the **`draw()`** method for each object needed only to worry about drawing a simple shape at the "screen location" defined in the object. Now it needs to apply the "current transformations" of the object so that the object will be properly drawn. This is done utilizing the mechanism discussed in class: the **`draw()`** method (1) saves the current **`Graphics2D`** transform, (2) appends the own transformation of the object onto the **`Graphics2D`** transform, (3) draws the object (and its sub-objects, in the case of a hierarchical object) using *local coordinate system* draw operations, and (4) then restores the saved **`Graphics2D`** transform. That is, each **`draw()`** method

temporarily adds its own object's local transformation to the **Graphics2D** transformation prior to invoking drawing operations, and then restores the **Graphics2D** transformation before returning.

All object drawing methods are to specify the object's appearance in "local object coordinate space". That is, all drawing operations must be relative to the object's "local origin" (0.0, 0.0). This is different from the previous assignment, where your **draw()** commands specified the "screen 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)**, this command should be changed to be **g.drawRect(0,0,width,height)**, drawing the rectangle in "local space" at (0,0) – which is then translated by the AT to the proper location in the world.) The major effect is that the output of **draw()** operations will be coordinates in "world space". Note that an additional side effect of this approach is that objects no longer need X,Y location values, and that their **getLocation()** methods should return values obtained by reading the translation elements of the object's translation transformation; you should delete the "location" attribute of your game objects.

In addition, the **draw()** methods for objects which have a direction must ensure that the objects are drawn with an orientation which matches the direction they are heading (for example, Animals and ShowWaves should face the direction they are moving). This change of orientation must be applied via a local rotation transformation. Likewise, the **draw()** methods for objects which can be resized (e.g., the net) must ensure that the objects are drawn with a scale that is applied via a local scale transformation.

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**. Note that the transformations contained in the VTM cause the world to be displayed "right-side up".

To apply the VTM during drawing, your MapView display panel's **paintComponent()** method should concatenate the current VTM 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 earlier, each **draw()** method will then in turn temporarily add its own object's local transformations to the same **Graphics2D** transformation.

In order to build a correct VTM, the program must keep track of the "current window" in the world – that is, the X/Y coordinates of the left, right, top, and bottom of the window in the world. The world window position values will be changed by the zoom and pan operations (see below).

The program may assume any initial (default) world window boundary locations you choose, including world coordinate values which happen to be the same as the *screen* values you have been using (e.g., (0.0, 0.0) to (1000.0, 800.0)). Note however that these are now *world* coordinate values, not screen coordinates. Note also that world coordinates are always real numbers; you should be using Java type *float* or *double* to represent them. If objects move outside the *world window*, they will no longer be visible on the screen (Java will apply *clipping*; you don't have to) – but the user can cause them to become visible again by "zooming out".

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*, and *moving the mouse wheel backward zooms out*. Pan is to be implemented by capturing *mouse movement while a button is down* (i.e., mouse *drag*). Each of these operations (zoom in or out and pan left or right) applies an adjustment to the current world window boundary values and then tells the MapView panel to repaint itself. The MapView panel then computes a *new* VTM based on the updated world window and applies that VTM to the drawing operations. Zoom and pan are only supported in "Play" mode, not in "Pause".

Note that a side-effect of combining a world coordinate system with zoom and pan operations is that objects may disappear off the screen and subsequently become visible again when a "zoom-out" occurs. For example, the player might guide the net off the screen, only to have it become visible after zooming out a bit. You should be sure to test that this works correctly in your program.

Mouse Input

A mouse event contains a **`Point`** giving the current mouse location *in screen coordinates*. However, when selecting dogs (in pause mode), mouse input needs to determine *world* locations. Therefore, when selecting dogs (in pause mode), the program must transform mouse input coordinates from screen units to world units. To do this, apply the *inverse* of the VTM to the mouse coordinates (producing the corresponding point in the world).

Mouse input is complicated in one additional way. Each individual selectable shape's **`contains()`** method determines whether a given location (e.g. from the mouse) lies within the shape. However, since mouse locations (after applying the inverse VTM as described above) will be *world* locations but shapes are defined in their own *local* coordinate system, the mouse world location must be further transformed into the coordinate system of the shape. To do this, the **`contains()`** method of each shape must apply the inverse of the shape's *local transformations* to the mouse world coordinate in order to determine whether the world coordinate lies within the shape. Further, for *hierarchical* objects (i.e., Dogs) the **`contains()`** method for a shape must recursively determine whether the point is contained within any of that shape's sub-objects.

Be sure to compute the "inverse local transform" properly, including taking into account the order of application of these transforms. One method is to concatenate all local transforms into a single combined matrix, in the proper order, then obtain the inverse of that matrix. Another method is to apply the inverse of each local transform (translate, rotate, and scale) in sequence separately. However, note that it is a theorem of linear algebra that the inverse of a *sequence* of affine transforms is the product of the inverses of each individual transform, *in the opposite order*. For example, given a sequence [T] x [R] x [S], the inverse of this sequence is $[S]^{-1}$ x $[R]^{-1}$ x $[T]^{-1}$ (note the reversed order). See the Lecture Note Appendix on Matrix Algebra for further details.

ShockWaves (Bezier Curves)

As stated above, ShockWaves are shaped like Bezier curves. Each new curve is to be defined by 4 *randomly-generated* control points. The range of the (x,y) values of the control points should be constrained such that the curve can be as much as about three to four times the size of the other game objects (but no more; otherwise a single ShockWave becomes overwhelming). Note that since the control points are random (although constrained), some curves will be small

while others will be large, and each will be a unique shape. Note also that the curve becomes a new object, moving in a random direction and remaining in the world until it dissipates.

As with other shapes, the drawing routines for the curve should use *local coordinates* to draw the curve. Also, the drawing routine for the curve must use the recursive implementation (not an iterative implementation).

Deliverables

Submitting your program requires similar steps as for A1-A3:

1. Create a *single* file in "ZIP" format containing (1) the *Java source code (".java")* for all the classes in your program, and (2) the compiled (".class") files for your program *(3) your sound files contained in the proper folder as specified in A3* (UML is not needed). Your program must be in a single package named "**a4"**, with a main class named "**`Starter`**". Be sure your ZIP file contains the proper folder hierarchy (i.e., "a4" and "sound" folders reside at the same level of the hierarchy). Also include in your ZIP a text file called "readme.txt" in which you describe any changes or additions you made to the assignment specifications.

2. *Verify your submission file.* To do this, copy your ZIP file to an empty folder on your machine, unzip it, open a command prompt window, change to the folder where you unzipped the file, and type the command "**java a4.Starter"**. If this does not properly execute your program, your ZIP file is not properly structured; go back to step (1) and fix it. Substantial penalties will be applied to submissions which have not been properly verified according to the above steps.

3. Login to ***SacCT***, select "Assignment 4", and upload your *verified* ZIP file. Also, be sure to take note of the requirement stated in the course syllabus for keeping a *backup copy* of all submitted work (save a copy of your ZIP file).

As always, *all submitted work must be strictly your own.*

The due date for this assignment is Monday, Dec 14th. **This is also the final date for submission of late assignments. This deadline applies to *all* assignments (not just Assignment #4).**

Assignments submitted after 11:59pm Monday, Dec 14th will not be graded.