CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
Spring 2015
Clevenger / Muyan

Assignment #4: Transformations

Due Date: Monday May 18th [4 weeks]

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 As always, all operations from previous assignments must continue to work as before (except where superseded by new operations).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 device coordinate systems.

The game world is defined by an independent unbounded *world coordinate system*. Objects are to be drawn in their "local" coordinates; you must determine an appropriate scale (number of units) for the local coordinate system for each of your program's different drawable objects.[1] 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 Birds and Cars 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 object transformations.

At least one of your game objects is to be defined as a dynamically transformable hierarchical object composed of a hierarchy of at least three levels of shapes, each with its own transformation which positions the shape in relation to its parent shape. In addition, at least one of the shape transformations must change dynamically during the game. You may choose which game object(s) are hierarchical and how the components change dynamically, as long as (1) at least one hierarchical object is defined and (2) the game contains features which cause the transformation(s) of one or more subcomponents in the hierarchy to change dynamically.

As an example, you could define your Cars as being made up of a "Body" plus two "Axle" subcomponents (one at the front and one at the rear); each Axle subcomponent could be made up of a "Shaft" plus two "Tire" subcomponents. Transformations would be used to position and size the axles in relation to the car body, and to orient the tires in relation to the axles. The front tires could then dynamically rotate left or right depending on the direction and amount of the current *steeringDirection* of the car. (See below for a graphical example, but note that the picture is just an

[1] It doesn't really matter what the local coordinate scale is; you just have to insure that when your program creates a drawable object it does so using local coordinate values.

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. Shock Waves

ShockWaves are new movable objects which happen to be shaped like cubic Bezier curves. The game automatically generates a new ShockWave object each time the Player's car collides with either a Bird or a Non-Player Car. A new ShockWave has an initial location equal to the Player Car location, and a randomly-generated heading 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 hierarchical object like the "Fireball" example in the notes) or into world coordinates.

For objects which do not move, the object's transformations are set once when the object is created – to values which specify the world position and orientation of the object. For moveable objects, each Timer tick is to invoke **`move()`** on the moveable 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 heading, as before.

Hierarchical components which change dynamically should also have the change applied via transformations. For example, if you choose to create a hierarchical car with dynamically-changing front wheels as in the example, then when a left- or right-arrow key is pressed (indicating a change in steering direction) a corresponding rotation should be applied to the front tires of the player's car.

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, "location" is replaced by a *translation transformation* which controls the position of the object in the *world* (not on the screen). The **`draw()`** method for each object needs to apply the "current transformations" of the object so that the object will be properly drawn. This is done utilizing the steps discussed in class: the **`draw()`** method (1) saves the current **`Graphics2D`** transform, (2) appends the local transformations of its 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.

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 net 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 "position" attribute of your GameObjects.

In addition, the **`draw()`** methods for objects which have a heading must ensure that the objects are drawn with an orientation which matches the direction they are heading (for example, Birds and Cars should face the direction they are moving). This change of orientation must be applied via a local (rotation) 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 something like (for example) **`theVTM`**. Note that the transformations contained in the VTM cause the world to be displayed "right-side up" (that is, so that north is toward the top of the screen), and also fix the "turn anomaly" from A3: when the car is headed north and the user pushes the "left-arrow" key, the car will turn *left* on the screen (which is now also left *from the point of view of the car*), and start heading *west*.

To apply the VTM during drawing, your MapView display panel's **`paintComponent()`** method should build a VTM as shown in the Lecture Notes, and concatenate the 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 moveable objects may disappear off the screen and subsequently become visible again when a "zoom-out" occurs. For example, the player might drive his/her car 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.

To make your program more user-friendly you might consider changing the cursor to a "hand" while panning (see the **`getCursor()`** and **`setCursor()`** methods in the Java **`Component`** class, along with the AWT **`Cursor`** class); or you might arrange for zoom to be relative to the current mouse location instead of relative to the center of the MapView; or you might allow the user to draw a rubber-band rectangle specifying a new area of interest for zoom-in; however, these are not requirements. If you *do* implement any such variations, be sure to document them in a *ReadMe* file (see Deliverables, below).

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, when performing selection 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 screen coordinates (producing the corresponding point in the world).

Mouse input is complicated in one additional way. Each individual 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 the **`contains()`** method for a shape must recursively determine whether the point is contained within any of that shape's subcomponents.

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 $[S]^{-1} \times [R]^{-1} \times [T]^{-1}$ (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 world object, moving in a random direction and remaining in the world until it dissipates.

In addition to drawing the curve itself (in some color of your choosing), the draw routine for ShockWaves must also draw four different-colored straight lines connecting the control points (in order) so that the control polygon (bounding region, called the *convex hull*) of the curve can easily be seen. You may add a command to hide the drawing of the control polygon lines, although it must be ON by default. 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 an iterative implementation).

Deliverables

Submit a single .zip file to SacCT containing source code, compiled (.class), and resource files (such as sound files). Your program must be in a single package named "**a4**", with a main class named "**`Starter`**". If you added any enhancements to your program, also include a separate text file named *Readme.txt* in your Zip file explaining the enhancements.

As with A1-A3, follow these steps for submitting your program:

(1) create a single ZIP file containing your source code (.java) files, your compiled code (.class) files in package (folder) "a4", *your sound files contained in a "sounds" folder as specified in A3,* and a ReadMe.txt file if needed;

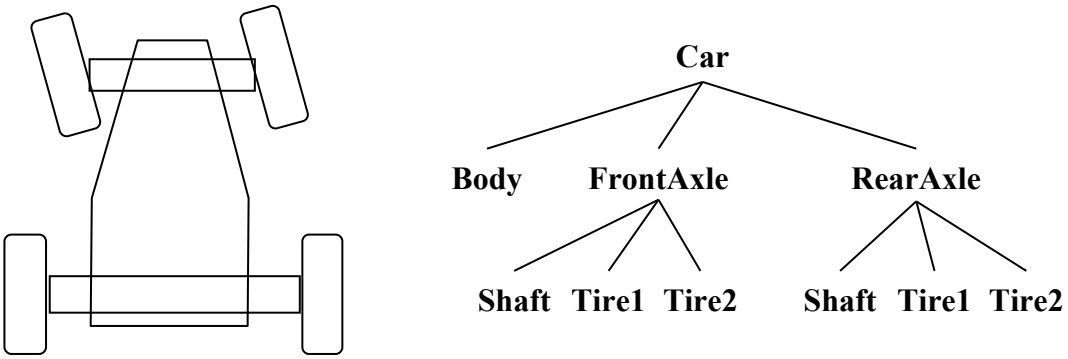
(2) *verify your submission* by unzipping your ZIP file to an empty folder and executing the program using the command **java a4.Starter** ; and then

(3) upload your ZIP file to SacCT. It is not a requirement to submit a UML diagram this time.

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

The due date for this assignment is Monday, May 18th. **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, May 18th will not be graded.



Hierarchical Car Example

Note the different shaft lengths, and the rotation of front tires in the "steering direction". Note also that the "Body" shape can be drawn with Graphics method *drawPolygon(),* and the "rounded rectangles" of the Tire shapes can be drawn with method *drawRoundRect().*

JC/PM:jc
4/13/15