CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
Spring 2021
Dr. Muyan

Assignment #4: Transformations

Due Date: Monday, May 17th, 11:59 PM

(May 17th is the final due date, no late assignments will be accepted for A4)

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 As always, all operations from previous assignments must continue to work as before (except where superseded by new operations).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

The game world is defined by an independent unbounded *world coordinate system*. All game objects are to be drawn in their "local" coordinates. The origin of the local coordinate system must coincide with the center of the object and y values must increase upward and x values must increase as going to the right. Local object transformations are to be used to map drawn objects from local to world coordinates. Appropriate local rotations must be applied to the ant, spiders, and shockwaves (see below) so that they face the direction in which they are moving in the world. If needed, appropriate local scaling factors should be applied to game objects so that they appear at their intended sizes in the world. Additionally, appropriate local translations must be applied to all game objects so that they are placed at their intended locations in the world. 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 *display coordinates,* so that the objects appear "right-side up" and proportionally at the same sizes and locations in all displays.

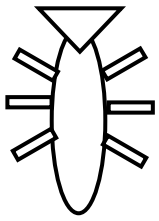
Additionally, the game is to support *zoom* and *pan* operations to allow the user to see close-up or far-away views of the world. Control over zooming and panning is to be done with pinching and pointer dragging, respectively.

2. Hierarchical object transformations

You must define the ant to be a dynamically transformable hierarchical object composed of a hierarchy of two levels of objects. Each sub-object (i.e., second-level object) of the ant (i.e., top-level object) has its own Local Transformations (LTs) which scale/rotate/translate the sub-object in relation to the origin of the local coordinate system of the ant (i.e., the center of the ant). In other words, LTs of the sub-object transform the sub-object from its local coordinate system into its top-level shape's (ant's) local coordinate system. The ant also has its own LTs which scale/rotate/translate the ant in relation to the origin of the world coordinate system. In other words, LTs of the ant transform the ant from its local coordinate system into world coordinate system. Origins of local coordinate systems of all top- and second-level-objects coincide with the origin of the world coordinate system.

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,

the ant could be constructed from eight sub-objects: “Body” (oval), “Head” (triangle), six “Legs” (rectangles), as shown in the following figure:



Note that you can use the following primitive shapes to create sub-objects: a square for legs, a circle for the body, and a triangle for the head. To add a dynamic transformation, you can make the legs go in and out of the body as the ant moves. However, be aware this is just an example; you may define the hierarchy and the dynamic transformation 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 ant collides with a spider. A new shockwave has an initial location equal to the ant location, and a randomly-generated constant heading and speed values (the speed should be slow enough that the shockwave is seen on the display 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 be equal to 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 display coordinates – the “location” of an object (center of object) was defined relative to the display origin (i.e., upper left corner of the MapView) and the **`draw...()`** method used for drawing each object’s used coordinates which are relative to the parent container origin (i.e., upper left corner of the content pane of the Game form). For instance, to determine the drawing coordinates of a food station (which is passed to its **`drawRect()`** method), we calculated the upper left corner point of the food station in the display coordinate system (in display space) based on its center location and then added **`getX()/Y()`** values of the MapView to this upper left corner point.

Now, each object should be defined in its own *local coordinate system*. We define each object by determining the local space locations of its points (i.e., local points) that will be utilized in **`draw...()`** methods. For instance, to determine the local points of a food station, which has a square shape, we determine the location of its lower left corner point in its local space (since lower left corner in the local space corresponds to upper left corner in the display space). Likewise, to

determine the local points of a flag, which has a triangle shape, we determine locations of top, bottom left, and bottom right corner points in its local space. Then, to calculate the drawing coordinates of each object (which is passed to its **`draw...()`** method), we add its local points to the **`getX()/Y()`** values of the MapView.

Each game object should have a set of CN1 **`Transform`** objects defining its *current LTs* (one **`Transform`** each for translation, rotation, and scaling). Hence, you should add **`myTranslate, myRotate,`** and **`myScale`** private fields of type **`Transform`** to GameObject. You should also add public methods that update these LTs called **`translate(), rotate(),`** and **`scale()`** to GameObject. This set of transformations specifies how the object is to be transformed from its local coordinate system into its top-level shape's coordinate system (in the case of a sub-object of the hierarchical object Ant) or into world coordinates (in the case of Ant, Spider, Flag, FoodStation, and ShockWave).

For movable objects (i.e., Ant, Spider, and ShockWave), each timer tick is to invoke **`move()`** on the object, as before. However, instead of changing the object's *location* values, the **`move()`** method will now *apply a translation to the object's translation LT*. The amount of this translation is calculated from the elapsed time, speed, and heading, as before. In addition, *rotation LTs* of the ant and spiders should be updated whenever their headings are changed (i.e., update it for the ant in turn left and right methods in Ant and update it for spiders in **`tick()`** method in GameWorld).

Since we are using their translation LTs to position objects in the world, game objects no longer need (x, y) location values. Hence, you must delete the "location" attribute of your game objects and **`getLocation()`** method in GameObject should return values obtained by reading the translation elements of the object's translation LT (use **`getTranslateX()/Y()`** methods of **`Transform`** class to read these elements).

The spiders should bounce off the virtual walls as in previous assignments. Starting in A2, we have been using the display (MapView) dimensions to set these boundaries. In A4, you are required to the use initial values of the world window defined in world space (see below) as these boundaries. You should not update the virtual wall boundaries when the world window size or location changes with the zoom and pan operations. You should check whether the center of the spider defined in the world space (returned by the **`getLocation()`** method which is updated as indicated above) is within the virtual wall boundaries, which are also defined in the world space. If the spider is out of the virtual wall boundaries, you should bounce the spider off the walls (e.g., update its rotation LT with proper values).

Previously, the **`draw()`** method for each object needed only to worry about drawing a simple shape at the "display location" defined in the object. Now it needs to apply the current LTs 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 **`Graphics`** transform, (2) performs "local origin" transformation - part two (3) appends the LTs of the object onto the **`Graphics`** transform, (4) performs "local origin" transform - part one (5) draws the object (or its sub-objects, in the case of a hierarchical object), and (6) then restores the saved **`Graphics`** transform (using **`setTransform()`**). That is, each **`draw()`** method temporarily adds its own LTs to the **`Graphics`** transformation prior to invoking drawing operations, and then restores the **`Graphics`** transformation before returning.

Please note that after VTM is applied, the text drawn on objects (using **`drawString()`** method) would look upside up, but mirrored across the vertical axis. To fix the mirroring effect, after the object is drawn (i.e., after Step#5 listed above), perform the following steps in the **`draw()`** method: (a) perform "local origin" transformation - part two (b) append scaling by (-1, 1) onto the **`Graphics`** transform, (c) perform "local origin" transform - part one, (d) draw the text, (e) resume with Step#6 listed above.

World/Display Coordinates

Your program must maintain a *Viewing Transformation Matrix (VTM)* which contains the series of transformations necessary to change world coordinates to display coordinates. This VTM is then applied to every drawing coordinate during a repaint operation. The VTM is simply an instance of the CN1 **`Transform`** class, named **`theVTM`**. Note that the transformations contained in the VTM cause the world to be displayed "right-side up" (e.g., triangles that represent flags do not look upside down any more, and now north is towards the top of the screen). Hence, for instance, in A4 when the ant is headed north and the user pushes the "left" button, the ant will turn *left* with respect to itself on the display, and start heading *northwest*. The VTM also makes the game objects appear proportionally at the same sizes and locations in all displays.

To apply the VTM during drawing, your MapView's **`paint()`** method should concatenate the current VTM into the **`Transform`** of the **`Graphics`** object used to perform the drawing (do not forget to apply "local origin" transformations before and after applying the VTM). **`paint()`** then passes this **`Graphics`** object to the **`draw()`** method of each object. As described earlier, each **`draw()`** method will then in turn temporarily adds object's LTs to the same **`Graphics`** transformation. Do not forget to call **`resetAffine()`** method on **`Graphics`** object at the end of **`paint()`** to restore its transformation to the original values.

In order to build a correct VTM, the program must keep track of the "current window" in the world – that is, left, right, top, and bottom boundaries of the window (the (x, y) coordinates of the lower left and upper corners of the window) in the world space (not display space). The world window values will be changed by the zoom and pan operations (see below).

You must initialize your world boundaries. Initially, the lower left corner of the world window, which corresponds to lower left corner of the display (MapView), must coincide with the origin (0.0, 0.0) of the world coordinate system. In addition, initially, upper right corner of the world window should be assigned to (**`MapView_Width/2, MapView_Height/2)`** which allow the world window to have the same aspect ratio as the display on the current skin (device). Please note that you can properly initialize upper right corner values only after calling **`show()`** on your form (since you can only get the correct values for the width and height of the MapView after **`show()`**). However, you still need to set upper right corner before **`show()`** is called so that you would not receive errors in **`paint()`** routine (that builds VTM which needs upper right corner value) when it is called for the first time (when **`show()`** is called). Hence, before the **`show(),`** you should set the value of upper right corner to some valid temporary value and then, call **`repaint()`** on MapView after setting the upper right corner to the correct initial value (which should be set right after **`show()`**).

Note that you should be using type *float* to represent world coordinates. Do not allow user to zoom out further, if one of the world window dimensions exceeds 1000 as indicated in the class notes. If objects move outside the current *world window*, they will no longer be visible on the screen (CN1 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 *pinching*, such that *mouse right click together with mouse movement towards the upper left corner of screen would zoom out* (on actual device this would be the pinching where two fingers come closer), and *mouse right click together with mouse movement going away from the upper left corner of screen would zoom in* (on actual device this would be the pinching where two fingers go away from each other). Pan is to be implemented by capturing pointer drag, such that *mouse left click together with mouse movement would pan*. Pointer drag would move the world window in the dragging direction. Hence, dragging towards left would move the objects on screen towards right and vise-versa. Likewise, dragging towards bottom would move the objects on screen towards top and vise-versa.

Each of zoom in/out and pan left/right/top/bottom operations applies an adjustment to the current world window boundary values and then tells the MapView to repaint itself. The MapView then computes a *new* VTM based on the updated world window and applies that VTM to the transform object of the graphics object of the MapView. Zoom and pan are supported both in "Play" and "Pause" modes.

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 make the ant go off the display, only to have it become visible after zooming out a bit. You should be sure to test that this works correctly in your program.

Pointer Input

The overriden **`pointerPressed()`** method provides (x, y) coordinate of the pointer location *in screen space*. However, since each object is defined in its local space (hence, does not know about the screen/display/world space), when selecting objects (in pause mode), we need to determine the corresponding pointer locations in the local spaces of the objects.

Hence, when selecting an object, first, the program must transform the pointer input coordinate from screen space to world space. To do this, apply the *inverse* of the VTM to the pointer coordinates in display space (first convert the pointer coordinates from screen space to display space by deducting **`getAbsoluteX()/Y()`** values of **`MapView`**) to produce the corresponding point in the world. Next, each individual selectable object's (in this game the selectable object are Flag and FoodStation) **`contains()`** method determines whether a given pointer location lies within the object. To do this, the **`contains()`** method of each object must apply the inverse of the object's *LTs* to the world coordinate of the pointer in order to determine the corresponding pointer location in its local space. Note that this means that in A4, the signature of **`contains()`** method in **`ISelectable`** interface used in A3 must be updated as indicated in the class notes.

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 transform, in the proper order, then to obtain the inverse of that transform (as indicated in the class notes). 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.

Bezier Curves

As stated above, shockwaves are shaped like cubic Bezier curves. Each new curve is to be defined by four *randomly-generated* control points in its local space (i.e., local 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.

As with other shapes, the drawing coordinates used by the drawing routine of the curve should be equal to *local points* plus **`getX()/Y()`** values of MapView. Also, the drawing routine for the curve must use the recursive implementation (not an iterative implementation).

Verification of Submission

Your program must be contained in a CN1 project called A4Prj. You must create your project following the instructions given at "2 – Introduction to Mobile App Development and CN1" lecture notes posted at Canvas (Steps for Eclipse: 1) File → New → Project → Codename One Project. 2) Give a project name "A4Prj" and uncheck "Java 8 project" 3) Hit "Next". 4) Give a main class name "Starter", package name "com.mycompany.a4", and select a "native" theme, and "Hello World(Bare Bones)" template (for manual GUI building). 5) Hit "Finish".). Further, ***you must verify that your program works properly from the command prompt*** before submitting it to Canvas: First make sure that the A4Prj.jar file is up-to-date. Then get into the A4Prj directory and type (all in one line) the following for Windows machines:

```
java -cp dist\A4Prj.jar;JavaSE.jar com.codename1.impl.javase.Simulator com.mycompany.a4.Starter
```

For the command line arguments of Mac OS/Linux machines please refer to the class notes. Penalties will be applied to submissions which do not work properly from the command prompt.

Deliverables

Submitting your program requires similar steps to A3 (as in A3, UML is not needed in A4):

1. Create a "TEXT" (i.e., not a pdf, doc etc.) file called "readme-a4.txt" that includes the lab number and the name of the specific machine you have used in that lab to build/test your program. Be sure to **verify that your program works from the command prompt on the lab machine** as explained above. For instructions on how to connect to a lab machine, please see the related page on Canvas and for hints on how to how to build/test your assignment on a lab machine, please see the syllabus. In addition, **be sure that you include the *src* folder and *jar* file generated/tested on the lab machine in the below-mentioned zip file**. You may also include additional information you want to share with the grader in this text file. You will receive the grader comments on your text file as annotations when grades are posted (click on "View Feedback" link next to readme-a4.txt on Canvas to see the comments).

2. Create a "ZIP" file containing (1) the entire **"src" directory** under your CN1 project directory (called A4Prj) which includes source code (".java") for all the classes in your program and the audio files, and (2) the **A4Prj.jar** (located under the "A4Prj/dist" directory) which is automatically generated by CN1 and includes the compiled (".class") files for your program in a zipped format.

Do **NOT** include other directories/files under the CN1 project directory. Be sure to name your ZIP file as YourLastName-YourFirstName-a4.zip.

3. Login to ***Canvas***, select "Assignment#4", and upload your ZIP file and TEXT file separately (do **NOT** place this TEXT file inside the 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 and TEXT files).

As always, all submitted work must be strictly your own!

The due date for this assignment is Monday, May 17th. **This is also the final date for submission of A4 since the 10-day late policy does NOT apply to the last assignment.**

Assignments submitted after 11:59pm Monday, May 17th will not be graded.