CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

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

Assignment #4: Transformations

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 As always, all operations from previous assignments must continue to work as before (except where superseded by new operations).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

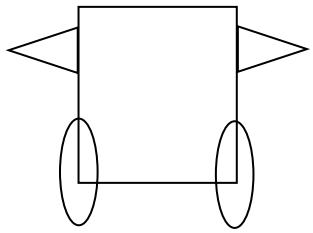
The game world is defined by an independent unbounded *world coordinate system*. All game objects are to be drawn in their "local" coordinates. The origin of the local coordinate system must coincide with the center of the object and y values must increase upward and x values must increase as going to 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 Robots (i.e., NPRs and the player robot), Drones, 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 player robot 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 player robot (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 player robot (i.e., the center of the player robot). In other words, LTs of the sub-object transform the sub-object from its local coordinate system into its top-level shape's (player robot's) local coordinate system. Remember that the player robot also has its own LTs which scale/rotate/translate the player robot in relation to the origin of the world coordinate system. In other words, LTs of the player robot transform the robot 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 or player actions and this change should be visible to the player. For example, the player robot could be constructed from five sub-objects: “Body” (square), two “Arms” (triangles), and two “Tires” (ellipses), as shown in the below figure:



To add a dynamic transformation, you can make the arms go up and down as the player robot moves. Also note that, to construct the above example, you can use the isosceles triangle, square, and perfect circle to create the arms, body, and tires, respectively.

Remember that the picture above is just an example; you may define the player robot and its dynamic transformation in any way you like as long as the player robot is hierarchical and its dynamic transformation changes as a function of the timer or player actions 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 a robot (i.e., player robot or NPR) collides with a drone or another robot (i.e., player robot or NPR). A new ShockWave has an initial location equal to the robot 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 ShockWave.

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 **`drawXXX()`** 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 NPR (which is passed to its **`drawRect()`** method), we calculated the upper left corner point of the NPR 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

drawXXX() methods. For instance, to determine the local points of a NPR, 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 drone, 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 **drawXXX()** 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. O*ne **Transform** for each 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, i.e., the player robot) or into world coordinates (in the case of Robot, Drone, Base, Energy Station, and ShockWave).

For movable objects (i.e., Robot, Drone, 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 local translation transformation*. The amount of this translation is calculated from the elapsed time, speed, and heading, as before. In addition, *rotation LTs* of robots and drones should be updated whenever their headings are changed (i.e., update it for robots in turn left and right methods in GameWorld and update it for drones in **tick()** method).

Since we are using their local translations 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 local translation transformation (use **getTranslateX()/Y()** methods of **Transform** class to read these elements).

The drones 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 drone 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 drone is out of the virtual wall boundaries, you should bounce the drone off the walls (e.g., update its local rotation transformation 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) perform "local origin" transformation - part two (3) appends the LTs of the object onto the **Graphics** transform, (4) perform "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.

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 bases do not look upside down any more, and now north is towards the top of the screen). 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. 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 drags would move the world window in the dragging direction. Hence, dragging towards left would move the objects on screen towards right and vice versa. Likewise, dragging towards bottom would move the objects on screen towards top and vice 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 steer the player robot 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 overridden **`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 Base and Energy Station) **`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. Further, for a *hierarchical* object (in this game the player robot is the hierarchical object) the **`contains()`** method for the object must recursively determine whether the point is contained within any of that shape's sub-objects as indicated in the class notes. 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 LTs 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 LT (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 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 (Get into the A4Prj directory and type "java --module-path "%PATH_TO_JAVAFX_LIB%" --add-modules=ALL-MODULE-PATH -cp dist\A3Prj.jar;JavaSE.jar com.codename1.impl.javase.Simulator com.mycompany.a3.Starter". For Mac OS/Linux machines, use the following command line: "java --module-path $PATH_TO_JAVAFX_LIB --add-modules=ALL-MODULE-PATH -cp dist/A3Prj.jar:JavaSE.jar com.codename1.impl.javase.Simulator com.mycompany.a3.Starter". Substantial 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. Be sure to verify that your program works from the command prompt as explained above.

2. Create a *single* file in "ZIP" format 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 and audio files in a zipped format. Be sure to name your ZIP file as YourLastName-YourFirstName-a4.zip. Also include in your ZIP a text file called "readme" in which you describe any changes or additions you made to the assignment specifications.

3. Create a "TEXT" (i.e., not a pdf, doc etc.) file called "readme-a4.txt" that includes your name and whether you have tested your submission on a lab machine. You are **strongly encouraged** to run and test your program on a lab machine (Hydra terminal server is recommended) before submitting your project. If you have done this test, also specify the lab number and the name of the specific machine you have used in that lab (or if you have used Hyrda as recommended, just say it was tested on Hydra) in the text file. In addition, if you have done this test, be sure that you include the

src folder and *jar* file generated/tested on the lab machine in the above-mentioned zip file. This information helps the grader in case your submission does not work on the machine (s)he uses. 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 when grades are posted.

4. 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!