CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
Fall 2013
John Clevenger

Assignment #4: Transformations

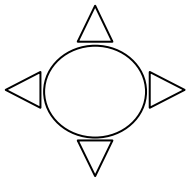
Due Date: Tuesday, December 17th [3½ weeks]

Overview

For this assignment you are to extend your program from Assignment #3 (A3) to include several uses of 2D transformations plus additional graphics. As always, all operations from previous assignments must continue to work as before (except where superseded by new operations).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. Objects are 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 unbounded *world coordinate system*. Initially, the origin (0,0) of the world coincides with the screen's lower left corner (MapView area). A screen transformation is used so that the game world appears "right-side up". 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/panning is to be done with the mouse wheel and mouse movement. Additionally, an appropriate rotation must be applied to movable objects so that they "face" the direction they move.

2. Hierarchical object transformations. The research-and-development department has created a new kind of missile which player tanks can fire: a *Spiked Grenade*. These are to be defined as a dynamically transformable hierarchical object instead of as a single geometric shape or image. That is, each Spiked Grenade 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. In addition, at least one of the shape transformations must change dynamically as a function of time during the game, and the grenade must also "tumble" (rotate around its top-level origin) as it flies toward its target. As an example, a Spiked Grenade could be constructed as shown in the figure to the right, with the triangles moving in and out while the entire grenade tumbles as it moves. Use the "G" key to fire a Grenade. You may define the effect in the game of a Spiked Grenade hitting an enemy tank as you choose. (You may also add the ability to fire Spiked Grenades to enemy tanks, but it is not a requirement.) Note that a Spiked Grenade is defined as "a kind of Missile", meaning you should implement it in your code by *extending the Missile class* and defining grenade-specific operations (such as the way it gets drawn) in the subclass. The majority of a grenade's behavior will be inherited from the parent (Missile) class – including that it has a limited lifetime after which it gets removed from the world.



3. Bezier Curves. Player tanks are to be enhanced with the ability to throw *plasma waves*. Plasma waves are movable objects which happen to be shaped like Bezier curves. When the player presses the 'P' key it causes the player's tank to throw a plasma wave. (You may also add the ability to throw plasma waves to enemy tanks, but it is not a requirement.) Plasma waves move in the direction the tank is heading but at a slightly higher speed. When a plasma wave collides with another tank, the other tank is immediately destroyed (removed from the game) and

the user scores some points. Plasma waves go harmlessly through all objects except tanks. They have a maximum lifetime (you may choose the value) after which they dissipate and are removed from the world.

Additional Details

Local Coordinates and Object Transformations

Previously, each object was defined and drawn directly in terms of its screen coordinates – 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 components of hierarchical objects) or into world coordinates.

For objects which do not move, the current transformation is set once – to the world position and 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. 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. The "position" field(s) in objects are no longer needed and should be removed.

Previously, the **`draw()`** method for each object needed only to worry about drawing a simple object 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 needs to apply the "current transformation" 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 its object's own transformations 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 then (4) 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 methods specify the object's appearance in "local object coordinate space". That is, all drawing operations 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)`**, now it is **`g.drawRect(0,0,width,height)`**, drawing in "local space" at (0,0) – which is then translated by the AT to the proper location.) The net effect is that *the output of **`draw()`** operations will be coordinates in "world space"*.

Hierarchical components, and components which change dynamically (such as the rotation of a Spiked Grenade as it flies) also should have those changes applied via transformations.

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 concatenate the current VTM onto 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 window position values will be changed by the zoom and pan operations (see below).

The program may assume any initial (default) 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., from (0.0, 0.0) to (1023.0, 1023.0)). 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. Also, since the game world is now infinite, you should remove any code which forces objects to remain inside a fixed boundary (for example, tanks no longer "block" at the screen edges; they can move off-screen).

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 only need to operate in *play* mode.

To make your program more user-friendly you might consider changing the cursor to a "hand" while panning, or arranging for zoom to be relative to the current mouse location instead of relative to the center of the MapView, but these are not requirements.

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.

Bezier Curves

Each time the 'P' key is pressed the player's tank should generate a *different* plasma wave (curve). 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 five times the size of the tank (but no more; otherwise a single plasma shot becomes a "doomsday" weapon). 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 the direction the tank was heading, and remains in the world until it collides with a tank or dissipates.

In addition to drawing the curve itself, the draw routine for plasma waves must also draw four 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 choose the color scheme for drawing the curve and the control polygon lines. You may also 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`**". As always, *all submitted work must be strictly your own.*

The due date for this assignment is Tuesday, December 17th. **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 Tuesday, December 17th will not be graded.

JC:jc
11/21/13