CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
Spring 2019 - Dr. Muyan

Assignment #3: Interactive Graphics and Animation

Due Date: Monday, April 15th 11:59 PM

1. Introduction

The purpose of this assignment is to help you gain experience with interactive graphics and animation techniques such as repainting, timer-driven animation, collision detection, and object selection. Specifically, you are to make the following modifications to your game:

(1) the game world map is to display in the GUI (in addition to the text form on the console),
(2) the movement (animation) of game objects is to be driven by a timer,
(3) the game is to support dynamic collision detection and response,
(4) the game is to support simple interactive editing of some of the objects in the world, and
(5) the game is to include sounds appropriate to collisions and other events.

2. Game World Map

If you did Assignment#2 (A#2) properly, your program included an instance of a **`MapView`** class which is an observer that displayed the game elements *on the console*. **`MapView`** also extended **`Container`** and that panel was placed in the middle of the game frame, although it was empty.

For this assignment, **`MapView`** will display the contents of the game *graphically* in the container in the middle of the game screen (in addition to displaying it in the text form on the console). When the **`MapView update()`** is invoked, it should now also should call **`repaint()`** on itself. As described in the course notes, **`MapView`** should override **`paint()`**, which will be invoked as a result of calling **`repaint()`**. It is then the duty of **`paint()`** to iterate through the game objects invoking **`draw()`** in each object – thus redrawing all the objects in the world in the container. Note that **`paint()`** must have access to the **`GameWorld`**. That means that the reference to the **`GameWorld`** must be saved when **`MapView`** is constructed, or alternatively the **`update()`** method must save it prior to calling **`repaint()`**. Note that the modified **`MapView`** class communicates with the rest of the program *exactly* as it did previously (e.g., it is an observer of **`GameWorld`**).

As indicated in A#1, each type of game object has a different shape which can be bounded by a square. The size attribute provides the length of this bounding square. The different graphical representation of each game object is as follows: Bases are filled isosceles triangles; energy stations are filled circle; the player robot is a filled square while non-player robots are unfilled square; and drones are unfilled isosceles triangles. Hence, the size attribute of an energy station indicates the diameter of the circle, size of a base or drone indicates the length of the unequal side and height of the isosceles triangle, and size of a robot indicates the length of equal sides of the square.

Bases should include a text showing their number; energy stations should include a text showing their capacity. Use the **`Graphics`** method **`drawString()`** to draw the text on bases and energy stations.

The appropriate place to put the responsibility for drawing each shape is within each type of game object (that is, to use a *polymorphic* drawing capability). The program should define a new interface named **`IDrawable`** specifying a method **`draw(Graphics g, Point pCmpRelPrnt)`**. **`GameObject`** class should implement this interface and each concrete game object class should then provide code for drawing that particular object using the received **`Graphics`** object **`g`** (which belong to **`MapView`**) and **`Point`** object **`pCmpRelPrnt`**, which is the component location (**`MapView`**'s origin location which is located at its the upper left corner) relative to its parent container's origin (parent of **`MapView`** is the content pane of the **`Game`** form and origin of the parent is also located at its upper left corner). Remember that calling **`getX()`** and **`getY()`** methods on **`MapView`** would return the **`MapView`** component's location relative to its parent container's origin.

Each object's **`draw()`** method draws the object in its current color and size, at its current location. Recall that current location of the object is defined relative to the origin of the game world (which corresponds to the origin of the **`MapView`** in A#2 and A#3). Hence, do not forget to add **`MapView`**'s origin location (relative to its parent container's origin) to the current location while drawing your game objects since **`draw...()`** methods (e.g., **`drawRect()`**) of **`Graphics`** expects coordinates which are relative to the parent container's origin. Also, recall that the *location* of each object is the position of the *center* of that object. Each **`draw()`** method must take this definition into account when drawing an object. Remember that the **`draw...()`** method of the **`Graphics`** class expects to be given the X,Y coordinates of the *upper left corner* of the shape to be drawn. Thus, a **`draw()`** method would need to use the *location* and *size* attributes of the object to determine where to draw the object so its center coincides with its *location* (i.e., the X,Y coordinate of the upper left corner of a game object would be at **`center_location.x – size/2, center_location.y – size/2`** relative to the origin of the **`MapView,`** which is the origin of the game world).

3. Animation Control

The **`Game`** class is to include a timer (you should use the **`UITimer`**, a build-in CN1 class) to drive the animation (movement of movable objects). **`Game`** should also implement **`Runnable`** (a build-in CN1 interface). Each tick generated by the timer should call the **`run()`** method in **`Game`**. **`run()`** in turn can then invoke the "Tick" method in **`GameWorld`** from the previous assignment, causing all moveable objects to move. This replaces the "Tick" button, which is no longer needed and should be eliminated.

There are some changes in the way the Tick method works for this assignment. In order for the animation to look smooth, the timer itself will have to tick at a fairly fast rate (about every 20 *msec* or so). In order for each movable object to know how far it should move, each timer tick should pass an "elapsed time" value to the **`move()`** method. The **`move()`** method should use this elapsed time value when it computes a new location. For simplicity, you can simply pass the value of the timer event rate (e.g., 20 msec), rather than computing a true elapsed time. However, *it is a requirement that each* **`move()`** *computes movement based on the value of the elapsed time parameter passed in*, not by assuming a hard-coded time value within the **`move()`** method itself. You should experiment to determine appropriate movement values (e.g., in A#1, we have specified the initial speed of the drone to be a random value

between 5 and 10, you may need to adjust this range to make your drones have reasonable speed which is not to too fast or too slow). In addition, be aware that methods of the built-in CN1 **`Math`** class that you will use in **`move()`** method (e.g., **`Math.cos()`**, **`Math.sin()`**) expects the angles to be provided in radians not degrees. You can use **`Math.toRadians()`** to convert degrees to radians. Likewise, the built-in **`MathUtil.atan()`** method that you might have used in the strategy classes also produces an angle in radians. You can use **`Math.toDegrees()`** to convert degrees to radians.

Remember that a **`UITimer`** starts as soon as its **`schedule()`** method is called. To stop a **`UITimer`** call its **`cancel()`** method. To re-start it call the **`schedule()`** method again.

4. Collision Detection and Response

There is another important thing that needs to happen on Tick method in **`GameWorld`**. After invoking **`move()`** for all movable objects, your Tick method must determine if there are any collisions between objects, and if so to perform the appropriate "collision response". You must handle collision detection/response by having **`GameObject`** class implement a new interface called "**`ICollider`**" which declares two methods
`boolean collidesWith(GameObject otherObject)` and
`void handleCollision(GameObject otherObject)` which are intended for performing collision detection and response, respectively.

In the previous assignment, collisions were caused by pressing one of the "pretend collision buttons" (i.e., "Collide With NPR", "Collide With Base", "Collide With Drone", "Collide with Energy Station") and the objects involved in the collision were chosen arbitrarily. Now, the type of collision will be detected automatically during collision detection, so the pretend collision buttons are no longer needed and should be removed. Collision detection will require objects to check to see if they have collided with other objects, so the actual collisions will no longer be arbitrary, but will correspond to actual collisions in the game world. There are more hints regarding collision detection in the notes below.

Collision response (that is, the specific action taken by an object when it collides with another object) will be similar as before. Hence, **`handleCollision()`** method of a game object should call the appropriate collision handling method in **`GameWorld`** from the previous assignment. Collisions also generate a *sound* (see below) and thus, collision handling methods in **`GameWorld`** should be updated accordingly.

In addition to the player robot, NPRs can also collide with other NPRs, bases, drones, and energy stations. The effects of these collisions on an NPR would be the same as the effects caused by these collisions on the player robot (i.e., collision with other NPR would increase damage level and fade color of both NPRs, collision with base would increase the *lastBaseReached* value of the NPR given that the base number is one more than the current *lastBaseReached* value, collision with a drone would increase the damage level of the NPR, collision with an energy station would increase NPR's energy level, reduce capacity of the energy station to zero, fade the color of the energy station, and add a new energy station).

Finally, since we now automatically detect collisions, we do not need to (and thus, we should not) assign the next *lastBaseReached* value to NPRs each time "Change Strategies" button is hit. This value of the NPR will now be updated as a result of collisions with the bases. In addition, although we still assume that NPRs never run out of energy, collision between a NPR and an energy station would increase the energy level of the NPR.

5. Sound

You may add as many sounds into your game as you wish. However, you must implement particular, clearly different sounds for *at least* the following situations:

(1) when robots collides, e.g., a player robot-NPR or NPR-NPR collision happens (such as a crash sound),
(2) when a robot (NPR/player robot) collides with an energy station (such as an electric charge sound),
(3) any collision which results in the player robot losing a life (such as an explosion or dying sound),
(4) some sort of appropriate background sound that loops continuously during animation.

Sounds should only be played if the "Sound" attribute is "On". Note that except for the "background" sound, sounds are played as a result of executing a collision. Hence, you must play these sounds in collision methods in **`GameWorld`**.

You may use any sounds you like, as long as I can show the game to the Dean and your mother (in other words, they are not disgusting or obscene). Short, unique sounds tend to improve game playability by avoiding too much confusing sound overlap. Do not use copyrighted sounds. You may search the web to find these non-copyrighted sounds (e.g., **`www.findsounds.com`**).

You must copy the sound files directly under the *src* directory of your project for CN1 to locate them. You should add **`Sound`** and **`BGSound`** classes to your project to add a capability for playing regular and looping (e.g., background) sounds, respectively, as discussed in the lecture notes. These classes encapsulate given sound files by making use of **`InputStream`**, **`MediaManager, and Media`** build-in CN1 classes. In addition to these built-in classes, **`BGSound`** also utilizes **`Runnable`** build-in CN1 interface. You should create a single sound object in **`GameWorld`** for each audio file and when you need to play the same sound file, you should use this single instance (e.g., all robot and energy station collisions should use the same sound object).

6. Object Selection and Game Modes

In order for us to explore the Command design pattern more thoroughly, and to gain experience with graphical object selection, we are going to add an additional capability to the game. The game is to have two modes: "*play*" and "*pause*". The normal game play with animation as implemented above is "play" mode. In "pause" mode, animation stops – the game objects don't move, the clock stops, and the background looped sound also stops. Also, when in pause mode, the user can use the pointer to select some of the game objects as explained below.

Ability to select the game mode should be implemented via a new GUI command button that switches between "play" and "pause" modes (you should create an additional command class to handle action events generated by this button and set its target as **`Game`**). When the game first starts it should be in play mode, with the mode control button displaying the label "Pause" (indicating that pushing the button switches to pause mode). Pressing the Pause button switches the game to pause mode and changes the label on the button to "Play", indicating that pressing the button again resumes play and puts the game back into play mode (also restarting the background sound, if sound is enabled).

- Object Selection

When in pause mode, the game must support the ability to interactively *select* objects. To identify "*selectable"* objects, you should have those objects implement an interface called **`ISelectable`** which specifies the methods required to implement selection, as discussed in the lecture notes. Selecting an object allows the user to perform certain actions on the selected object. For this assignment, all **`Fixed`** objects (*Energy Stations and Bases)* are selectable. The selected energy station/base must be highlighted by drawing it as an un-filled circle/triangle (a normal energy station/base is drawn as a filled circle/triangle). Selection is only allowed in *pause* mode, and switching to *play* mode automatically "unselects" the object.

An individual object is selected by pressing the pointer on it. Pressing on an object selects that object and "unselects" all other objects. Clicking in a location where there are no objects causes the selected object to become unselected. Remember that pointer (x,y) location received by overriding the **`pointerPressed()`** method of **`MapView`** is relative to screen origin. You can make this location relative to **`MapView`**'s parent's origin by calling the following lines inside the **`pointerPressed()`** method:

```
x = x - getParent().getAbsoluteX()
y = y - getParent().getAbsoluteY()
```

A new ***Position*** command (which should extend from **`Command`** as the other commands of the game) is to be added to the game, invocable from a new "Position" button. When the position command is invoked, the selected energy station or base is to be moved to a new location. This movement should be done by following the below steps in the given order: 1) Select the object 2) Hit Position button 3) Click on **`MapView`** to select a new position for the selected object 4) Selected object would appear in the new location. The position action should only be available while in pause mode, and should have no effect on unselected objects. Note that the new position command gives the user the ability to create an arbitrary track configuration; the game is no longer constrained to use the hard-coded initial layout provided when the game starts or life is lost.

To execute the position command properly in A#3, we remove the requirement that was introduced in A#1 that restricted all fixed game objects from changing location once they are created. Also, note that regardless of their capacity all energy stations are allowed to be moved.

- Command Enabling/Disabling

Commands should be enabled only when their functionality is appropriate. For example, the Position command should be disabled while in play mode; likewise, commands that involve playing the game (e.g., changing the player robot direction) should be disabled while in pause mode. Note that disabling a command should disable it for all invokers (buttons, keys, *and* menu items). Note also that a disabled button or menu item should still be *visible*; it just cannot be active (enabled). This is indicated by changing the appearance of the button or menu item. To disable a button or a menu item which is added to the form by **`addComponentToSideMenu()`** use **`setEnabled()`** method of **`Button`** (remember that **`Checkbox`** is-a **`Button`**). To disable a menu item which is added to the form by **`addCommandToSideMenu()`** use **`setEnabled()`** method of **`Command`**. To disable a key, use **`removeKeyListener()`** method of **`Form`** (remember to re-add the key listener when the

command is enabled). You can set disabled style of a button using **`getDisabledStyle().set...()`** methods on the **`Button`**.

Additional Notes

- Please make sure that in A#2 you have added a **`MapView`** object directly to the center of the form. Adding a **`Container`** object to the center and then adding a **`MapView`** object onto it would cause incorrect results when a default layout is used for this center container (e.g., you cannot see any of the game objects being drawn in the center of the form).
- To draw a un-filled and filled triangle you can use **`drawPolygon()/fillPolygon()`** methods of **`Graphics.`** These methods expect the following parameters: **`xPoints`** which is an array of integers that has x coordinates of the corners, **`yPoints`** which is an array of integers that has y coordinates of the corners, **`nPoints`** which is the integer that indicates the number of corners of the polygon. For instance, to draw a triangle with corner coordinates (1,1) (3,1) (2,2), you should assign **`xPoints =`** {1,3,2}, **`yPoints =`** {1,1,2}, **`nPoints`** = 3.
- To draw a filled circle with radius **`r`** at location **`(x,y)`** use **`fillArc(x, y, 2*r, 2*r, 0, 360)`**.
- As before, the origin of the game world (which corresponds to the origin of the **`MapView`** for now) is considered to be in its *lower left* corner. Hence, the Y coordinate of the game world grows *upward* (Y values increase *upward*). However, origin of the **`MapView`** container is at its upper left corner and Y coordinate of the **`MapView`** grows *downward*. So when a game object moves north (e.g., it's heading is 0 and hence, its Y values is increasing) in the game world, they would move up in the game world. However, due to the coordinate systems mismatch mentioned above, heading up in the game world will result in moving down on the **`MapView`** (screen). Hence your game will be displayed as upside down on the screen. This also means that when we robots turn left they will go west in game world, but they will go east on the **`MapView`** (and turning right means going west on the **`MapView`**). In addition, triangles (e.g. bases and drones) will be drawn upside down in **`MapView`**. Leave it like this – we will fix it in A#4.
- The simple shape representations for game objects will produce some slightly weird visual effects in **`MapView`**. For example, squares or triangles will always be aligned with the X/Y axes even if the object being represented is moving at an angle relative to the axes. This is acceptable for now; we will see how to fix it in A#4.
- You may adjust the size of game objects for playability if you wish; just don't make them so large (or small) that the game becomes unwieldy.
- As indicated in A#2, boundaries of the game world are equal to dimensions of **`MapView`**. Hence, drones should consider the width and the height of **`MapView`** when they move, not to be out of boundaries.
- Because the sound can be turned on and off by the user (using the menu), and also turns on/off automatically with the pause/play button, you will need to test the various combinations. For example, turning off sound, then pressing pause, then pressing play, should result in the sound ***not*** coming back on. There are several sequences to test.

- When two objects collide handling the collision should be done only once. For instance, when two robots collide (i.e., robot1 and robot2) the robots damage level should be increased only once, not twice. In the two nested loops of collision detection, **`robot1.collidesWith(robot2)`** and **`robot2.collidesWith(robot1)`** will both return true. However, if we handle the collision with **`robot1.handleCollision(robot2)`** we should not handle it again by calling **`robot2.handleCollision(robot1)`**. This problem is complicated by the fact that in most cases the same collision will be detected repeatedly as one object passes through the other object. Another complication is that more than two objects can collide simultaneously. One straight-forward way of solving this complicated problem, is to have each collidable object (that may involve in such a problem) keep a list of the objects that it is already colliding with. An object can then skip the collision handling for objects it is already colliding with. Of course, you'll also have to remove objects from the list when they are no longer colliding.

  Implementation of this solution would require you to have a **`Vector`** (or **`Arraylist`**) for each collidable object (i.e., all game objects in Robo-Track game) which we will call as "collision vector". When collidable object obj1 collides with obj2, right after handling the collision, you need to add obj2 to collision vector of obj1. If obj2 is also a collidable object, you also need to add obj1 to collision vector of obj2. Each time you check for possible collisions (in each clock tick, after the moving the objects) you need to update the collision vectors. If the obj1 and obj2 are no longer colliding, you need to remove them from each other's collision vectors. You can use **`remove()`** method of **`Vector`** to remove an object from a collision vector. If two objects are still colliding (passes through each other), you should not add them again to the collision vectors. You can use **`contains()`** method of **`Vector`** to check if the object is already in the collision vector or not. The **`contains()`** method is also useful for deciding whether to handle collision or not. For instance, if the collision vector of obj2 already contains obj1 (or collision vector of obj1 already contains obj2), it means that collision between obj1 and obj2 is already handled and should not be handled again.

- You should tweak the parameters of your program for playability after you get the animation working. Things that impact playability include object sizes and speeds, etc. Your game is expected to operate in a reasonably-playable manner.
- As before, you may not use a "GUI Builder" for this assignment.
- All functionality from the previous assignment must be retained unless it is explicitly changed or deleted by the requirements for this assignment.

- Your program must be contained in a CN1 project called A3Prj. 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 "A3Prj" and uncheck "Java 8 project" 3) Hit "Next". 4) Give a main class name "Starter", package name "com.mycompany.a3", 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 A3Prj directory and type "java -cp dist\A3Prj.jar;JavaSE.jar com.codename1.impl.javase.Simulator com.mycompany.a3.Starter" for Windows machines. For the command line arguments of Mac OS/Linux machines please refer to the class notes.). Substantial penalties will be applied to submissions which do not work properly from the command prompt.

Deliverables

Submitting your program requires the same three steps as for A#1 and A#2 except that you **do not** need to submit a UML for A#3:

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 A3Prj) which includes source code (".java") for all the classes in your program and the audio files, and (2) the A3Prj.jar (located under the "A3Prj/dist" directory) which is automatically generated by CN1 and includes the compiled (".class") files for your program in a zipped format. Be sure to name your ZIP file as YourLastName-YourFirstName-a3.zip.

3. Create a "TEXT" (i.e., not a pdf, doc etc.) file called "readme-a3.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#3", 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).

All submitted work must be strictly your own!

Appendix 1: Sample GUI

The following shows an example of how a completed A#3 game might look. It has a control panel on the left, right, and bottom with the required command buttons. Notice that the bottom control container does not have the Collide With NPR, Collide With Base, Collide With Drone, Collide With Energy Station buttons as in A#2 GUI, instead it has the Pause/Play and Position buttons. Position button is disabled since the game here is in "Play" mode as indicated by the fact that the Pause/Play button shows "Pause". **MapView** in the middle contains *4 bases* (blue filled triangles), *2 energy stations* (green filled circles), 1 player robot (red filled square), 3 NPRs (red unfilled squares), and 2 drones (black unfilled triangles). Bases and energy stations include a text showing their number and capacity, respectively. As indicated in A#1, all bases and all robots have the same size whereas sizes of the rest of the game objects are chosen randomly when created. The initial capacity of an energy station is proportional to its size. Note also that the world is "upside down" (e.g., the triangles are facing down). We will fix this in A#4.