CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
Spring 2018 (Credits to Prof. Clevenger)

Assignment #3: Interactive Graphics and Animation

Due Date: Friday, April 20th [3 Weeks]

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, rather than in text form on the console,
(2) 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 include sounds appropriate to collisions and other events, and
(5) the game is to support simple interactive editing of some of the objects in the world.

1. Game World Map

If you did assignment #2 properly, your program included an instance of a **MapView** class that is an observer that displayed the game elements on the console. **MapView** also extended **Container** and that container 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. When the **MapView update()** is invoked, it should now call **repaint()** on itself. As described in the course notes, **MapView** should also implement (override) **paint()**, which will therefore be invoked as a result of calling **repaint()**. It is then the duty of **paint()** to iterate through the **GameWorld** objects (via an iterator, as before) invoking **draw(Graphics g)** in each **GameWorld** object – thus redrawing all the objects in the world in the container. Note that **paint()** must have access to the **GameWorld**. This means that a reference to the **GameWorld** must be saved when **MapView** is constructed, or else 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 its observable; it gets invoked via a call to its **update()** method as before; etc.).

- Map View Graphics

As specified in assignment #1, each game object has its own different graphical representation (shape). 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 (for example) **IDrawable** specifying a method **draw(Graphics g,** Point pCmpRelPrnt). Each game object should then implement the **IDrawable** interface with code that knows how to draw that particular object using the received "**Graphics**" object (which belong to MapView) and 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 drawXXX() methods of Graphics expects coordinates which are relative to the parent container's origin.

Each concrete game object is to have a unique shape – for example, a ship could be represented as an unfilled (open) triangle; a space station could be represented as a filled (solid) circle when it is "on" and an unfilled circle when it is "off"; a missile could be represented as a narrow filled rectangle while an asteroid could be represented as an unfilled square. (The preceding examples are not explicit requirements; you may choose other representations as long as each kind of object is unique on the screen.) Note in particular the requirement for a space station to have different representations depending on its "blink state".

Recall that the location of each object is the location of the center of that object. Each draw() method must take this definition into account when drawing an object. Remember that the drawXXX() method of the Graphics class expects to be given the X,Y coordinates of the upper left corner of the rectangle to be drawn, so for instance, a draw() method for a rectangular object would need to use the location, and size attributes of the object to determine where to draw the rectangle so its center coincides with its location. For example, the X coordinate of the upper left corner of a rectangle is (center.x – width/2) relative to the origin of the game world. Similar adjustments apply to drawing circles.

2. 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 generated by the timer should call the run() method in Game. run() in turn can then invoke the "Tick" command 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 command 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 elapsedTime 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 an object to be a random value between 5 and 10, you may need to adjust this range to make your objects have reasonable page 3 of 8 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.

3. Collision Detection and Response

There is another important thing that needs to happen each time the timer ticks. In addition to updating the Elapsed Time as necessary and invoking **move()** for all movable objects, your code must tell the game world to determine if there are any collisions between objects, and if so to perform the appropriate "collision response". The appropriate way to handle collision detection/response is to have each kind of object which can be involved in collisions implement a new interface like "**ICollider**" as discussed in class and described in the course notes. That way, colliding objects can be treated polymorphically.

In the previous assignment, collisions were caused by pressing one of the buttons ("kill asteroid", "kill flying saucer", "crash ship", etc.), and the objects involved in the collision were chosen arbitrarily. Now, the type of collision will be detected automatically during collision detection, so the "load missiles", "kill asteroid", "kill flying saucer", "crash ship", and "exterminate" buttons are no longer needed; they should be removed and replaced with collision detection which checks for the corresponding collisions. 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.

Collision response (that is, the specific action taken by an object when it collides with another object) will be similar as before: when two asteroids collide, they are removed; when a ship collides with an asteroid the ship is removed, a life is lost, and if there are lives remaining a new ship should be created[1];

If a missile struck an asteroid with a size of 25-30 (Large size), this asteroid will be broken into three new Asteroids. And the user scores a third top score (3rd highest) for taking out this object. The sum of the sizes (might not be the same) of these new asteroids is equal to the size of the original parent asteroid. The speed and direction of the new asteroids will be reset to new values. You can use a random function to generate these values. You decided the third top score.

When a missile collides with a flying saucer, the missile and flying saucer are removed and the player scores some points.

When the ship collides with a space station the ship receives a new missile supply. Some collisions also generate a *sound* (see below). There are more hints regarding collision detection in the notes below.

4. 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 a ship fires a missile,

(2) when a missile collides with an asteroid or flying saucer

[1] If your previous assignment did not add a new ship when there are remaining lives, you must modify your program for this assignment to do so.

(3) when the game ends due to no more lives, and

(4) some sort of appropriate background sound that loops continuously during animation.

You may also add sounds for other events if you like. Sounds should only be played if the "SOUND" attribute is "ON". You may use any sounds you like, as long as I can show the game to the Dean/Chair (in other words, the sounds 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 for each audio file and when you need to play the same sound file you should use this single instance.

5. 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. Specifically, 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 and the background looped sound also stops. Also, when in pause mode, the user can use the mouse to select some of the game objects.

Ability to select the game mode should be implemented via a new GUI command button that switches between "play" and "pause" modes. When the game first starts it should be in the 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).

- Object Selection

When in pause mode, the game must support the ability to interactively *select* objects. The appropriate mechanism for identifying "*selectable"* objects is to have those objects implement an interface such as **ISelectable** which specifies the methods required to implement selection, as discussed in class and described in the course notes. Selecting an object allows the user to perform certain actions on the selected object. Each selected object must be highlighted in some way (you may choose the form of highlighting, as long as there is some visible change to the appearance of each selected object). For this assignment, only *asteroids* and *missiles* are selectable. Selection is impossible in play mode.

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 ***Refuel*** command (Action) is to be added to the game, invocable from a new "Refuel" GUI button. When the Refuel command is invoked, all *selected* missiles are to have their fuel level set to the maximum. The Refuel action should only be available while in pause mode, and should have no effect on unselected items.

- Command Enabling/Disabling

Commands should be enabled only when their functionality is appropriate. For example, the Print (**P**) command should be disabled while in play mode; likewise, commands that involve playing the game (e.g., changing the player's car 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 and menu item (which is added to the form by addCommandToSideMenu()), use setEnabled() methods of Button and Command classes, respectively. 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().setXXX() methods on the Button.

Additional Notes

- Please make sure that 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). page 6 of 8

- To draw a un-filled and filled triangle you can use drawPolygon()/fillPolygon() methods of Graphics. Note that drawArc(x, y, 2*r, 2*r, 0, 360) draws a circle with radius r at location (x,y).

- 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. We will fix it in A#4.

- The simple shape representations for game objects will produce some slightly weird visual effects in the map view. 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, moveable object should consider the width and the height of the MapView in their move() method 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, pressing pause, then turning off sound, 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 objects collided (i.e., missile and asteroid) the score should be increased only once, not twice. In the two nested loops of collision detection, missile.collidesWith(asteroid) and asteroid.collidesWith(missile) will both return true. However, if we handle the collision with missile.handleCollision(asteroid) we should not handle it again by calling asteroid.handleCollision(missile). 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., car object) 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, page 7 of 8 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 the screen size, object sizes and speeds, moving speed of objects, collision distance, 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.

- When the game is over, animation should stop, and a dialog box should display showing the player's final score.

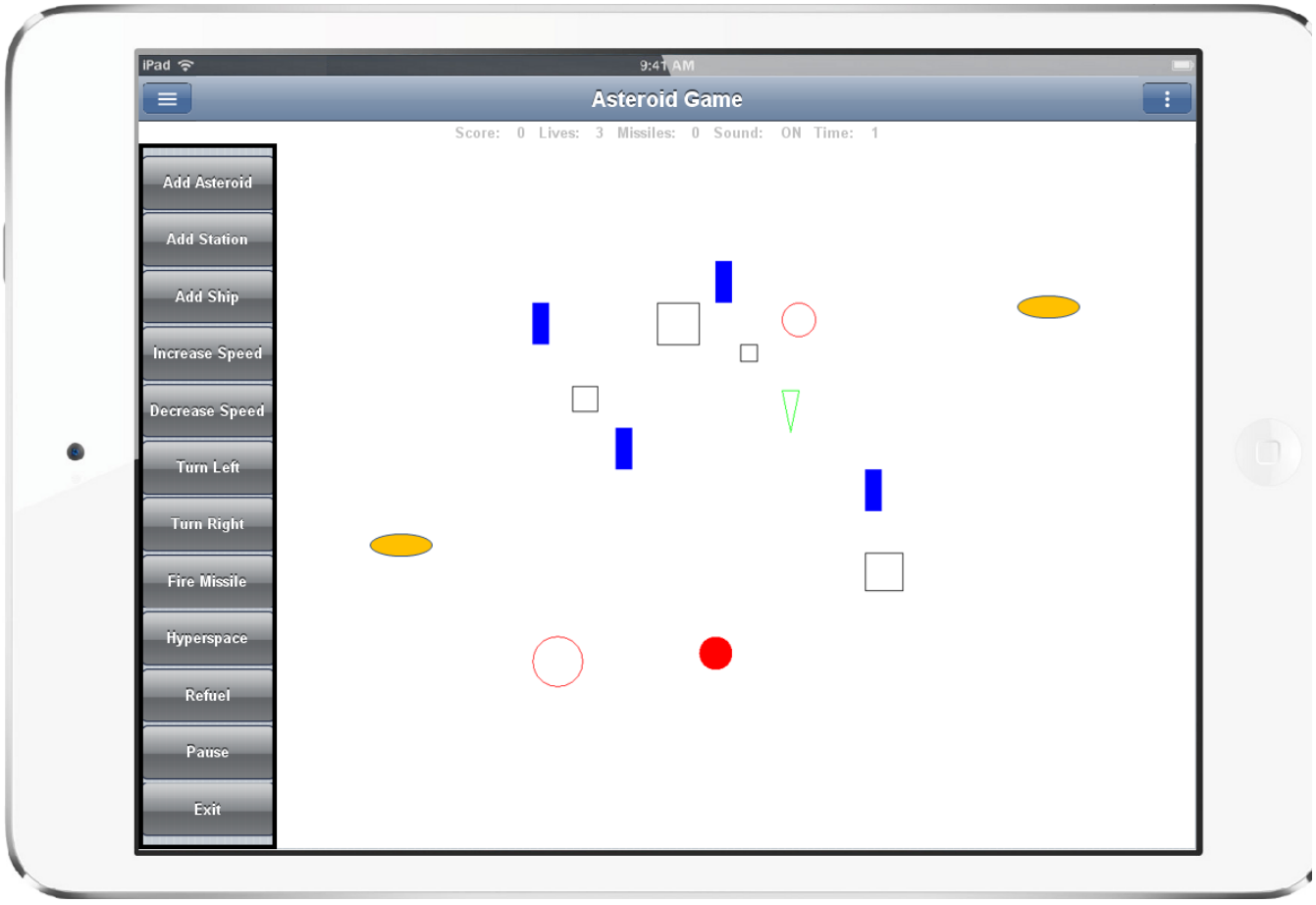
Deliverables

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 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-a3.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. Login to Canvas, select "Assignment 3", and upload your verified 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 file).

All submitted work must be strictly your own!

Sample GUI

The following shows one example of how a completed A3 game **MIGHT** look. Notice that it has a control panel on the left with the required command buttons (including the *disabled Refuel* command since the game here is in "Play" mode as indicated by the fact that the Pause/Play button shows "Pause"). It also has "File" and "Commands" menus, a points view panel showing the current game state, and a map view panel containing 4 asteroids (black squares), 4 missiles (blue rectangles), 3 space stations (red circles; two "off" and one "on"), 2 Flying Saucers (yellow ovals, and one ship (green triangle). Note that the moving objects are not necessarily oriented (rotated) in the direction in which they are moving; this is acceptable for this assignment (we'll see how to take care of that later). Note also that the world is "upside down" – the ship is currently heading "north" (0 degrees) but is facing the *bottom*. We will fix this in a subsequent assignment as well.



Note: This is a mockup screen for example only. Add UFO (Flying saucer) is not shown in control panel.