CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
Spring 2015 – Clevenger / Muyan

Assignment #3: Interactive Graphics and Animation

Due Date: Thursday, April 16th [3 Weeks + Spring Break]

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 program is to make use of the proxy and factory method design patterns,
(2) the game world map is to display in the GUI, rather than in text form on the console,
(3) the movement (animation) of game objects is to be driven by a timer,
(4) the game is to support dynamic collision detection and response,
(5) the game is to support simple interactive editing of some of the objects in the world, and
(6) the game is to include sounds appropriate to collisions and other events.

2. Proxy Design Pattern

To prevent Views from being able to modify the **`GameWorld`** object received by their **`update()`** methods, the model (**`GameWorld`**) should pass a GameWorld *proxy* to the views; this proxy should allow each view to obtain all the required data from the model while prohibiting any attempt by the view to *change* the model. The simplest way to do this is to define an interface **`IGameWorld`** listing all the methods provided by a **`GameWorld`**, and to provide a new class **`GameWorldProxy`** which implements this same interface. The **`GameWorld`** model then constructs and passes to each observer's **`update()`** method a **`GameWorldProxy`** object instead of the actual **`GameWorld`** object. See the course notes and the attachment at the end of Assignment #2 for additional details.

Recall that there are two approaches which could have been used to implement the **`Observable`** pattern: defining your own **`Observable`** interface, or extending the Java **`Observable`** class. If you chose to use Java's "**`Observable`**" class (e.g. declaring that your **`GameWorld`** class extends **`java.util.Observable`** instead of implementing your own **`Observable`** interface as discussed in class), you may find it is more complicated to deal with the proxy requirement: you will have to override not only the Java **`notifyObservers()`** method so that you can pass a proxy, but also the **`addObserver()`** method so that you can keep track of what observers to call back with a proxy. This means it may almost be less work to change your program to use an **`Observable`** *interface*.

3. Factory Method Design Pattern

If you implemented A2 correctly, all game object creation takes place in a method named **`initLayout()`**. Currently this method most likely makes direct calls to the constructors for each type of object to be added to the game world, which as discussed in class and in the

Lecture Notes tends to lead to code which is difficult to maintain and modify. For this assignment you are to replace each direct call to a game object constructor with a call instead to a *factory method* which returns an object of the corresponding type. For example, your program should now contain methods such as *makePylon(), makeCar()*, etc.; these methods should be invoked by **`initLayout()`** to obtain new game world objects.

4. Game World Map

If you did assignment #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 **`JPanel`** and that panel was placed in the middle of the game frame, although it was empty.

For this assignment, you are to change **`MapView`** so that it displays the contents of the game *graphically* in the **`JPanel`** 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) **`paintComponent()`**, which will therefore be invoked as a result of calling **`repaint()`**. It is then the duty of **`MapView`**'s **`paintComponent()`** 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 panel.

Note that **`paintComponent()`** must have access to the **`GameWorld`** (or more correctly, to a **`GameWorldProxy`**). This means that a reference to the **`GameWorldProxy`** must be saved when **`MapView`** is constructed, or alternatively the **`update()`** method must receive and 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.).

Each game object has its own different graphical representation (shape). Pylons are filled circles; oil slicks are filled ovals; fuel cans are filled squares; the player's car is a filled rectangle while non-player cars are unfilled rectangles; and birds are unfilled circles. Note that the attributes associated with each concrete game object can be used to specify the drawing characteristics of the object (for example, cars have a *length* and a *width* which can be used to draw the corresponding rectangle). Pylons should include a text String showing their number; fuel cans should include text showing their fuel quantity (size). (You may choose to also use the *size* attribute of a fuel can to determine the actual drawing size, but the game might look better if fuel cans were all the same size and used the text String to show quantity.) Use the **`Graphics`** method **`drawString()`** to draw the text on pylons and fuel cans.

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)`**. Each game object should then implement the **`IDrawable`** interface with code that knows how to draw that particular object using the received "**`Graphics`**" object (this then replaces the **`toString()`** polymorphic operation from the previous assignment.)

Each object's **`draw()`** method draws the object in its current color, at its current location. Recall that the *location* of each object is the location of the *center* of that object (this is for compatibility with things we will do later). Each **`draw()`** method must take this definition into account when drawing an object. For example, the **`drawRect()`** 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 a **`draw()`** method for a rectangular object would need to use the *location*, *width*, and *height* 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)`**. Similar adjustments apply to drawing circles.

5. Animation Control

The program is to include a timer to drive the animation (movement of movable objects). Each event generated by the timer should be caught by an **`actionPerformed()`** method. **`actionPerformed()`**can then in turn 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 (generate **`ActionEvents`**) at a fairly fast rate (about every 20 *milliseconds* or so). In order for each object to know how far it should move, each timer tick should pass an "elapsed *msec*" value to the **`move()`** method of each movable object. 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.

Another change has to do with the way the time value in your **`ScoreView`** JPanel is computed: it must take into account that it is being called at a much higher rate, so it must keep track of how many calls have been made and only decrement the on-screen counter when enough calls to equal one second of elapsed time have been made.

6. Collision Detection and Response

There is another important thing that needs to happen each time the timer ticks. In addition to 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 "collision buttons" ("Collide With NPC" for example), and the objects involved in the collision were chosen arbitrarily. Now, the type of collision will be detected automatically during collision detection, so those 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.

Collision response (that is, the specific action taken by an object when it collides with another object) will be similar in most cases to previous assignments (when the player's car collides with a fuel can, for example, the car's fuel level is increased, the fuel can is removed from the game, and another fuel can is added to the world). One additional constraint on collisions is that when the player's car collides with an NPC, there is a 20% probability that this

will cause an engine leak and create a new oil slick. Your player/NPC collision code should generate a random number and use it to determine whether a new oil slick is to be generated. If so, position the new oil slick at the location of the player's car at the time of the collision (as opposed to a random location as in previous assignments). Some collisions also generate a *sound* (see below). There are more hints regarding collision detection in the notes below.

7. 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 the player's car collides with an NPC (such as a crunching sound),
(2) when the player's car collides with a fuel can (such as a slurping sound),
(3) any collision which results in the player 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 command". This means that the collision command objects will need access to the "Sound attribute" contained in the GameWorld. This in turn means that these command objects must be given the GameWorld as a "target" (refer to the Course Notes section on the Command Design Pattern).

You may use any sounds you like in your program, as long as we can show the game to the Dean and your mother (in other words, as long as 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.

8. Object Selection and Game Modes

In order to gain experience with *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 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 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. 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 (or group of objects) allows the user to perform certain actions on the selected object(s). 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). Selection is only allowed in *pause* mode, and switching to *play* mode automatically "unselects" all objects. For this assignment, only *Fuel Cans* and *Pylons* are selectable.

An individual object is selected by *clicking* on it with the mouse. Clicking on an object normally selects that object and "unselects" all other objects. You must also implement one of the following: either, (a) clicking on several objects in succession *with the control (**CTRL**) key down* causes *each* of the clicked objects to become "selected" (as long as they are "selectable"), or (b) rubber-band selection (all the objects in a rubber-band rectangle become selected while all other objects become unselected). Regardless of which you choose, clicking in a location where there are no objects should cause all objects to become unselected.

- New Commands

The game is to support three new commands in Pause mode. The first new command is "Delete", which removes all currently selected objects from the game world. The Delete command should be bound to a new GUI button (see the picture at the end) and also to the Delete (DEL) key using a Java KeyBinding. The Delete action should only be available while in pause mode, and should have no effect on unselected items.

The additional two new commands are "Add Pylon" and "Add Fuel Can". These commands should be bound to new GUI buttons, and should add instances of the respective object to the world. The location of the added object is determined by clicking the mouse on the map view after having invoked the command. When the mouse is clicked the program should prompt (using a **`JOptionPane`**) for an input number; for adding a Fuel Can the number becomes the capacity (size) of the fuel can; for Pylons it becomes the sequence number of the new pylon.

Note that the combination of the three new commands gives the user the ability to create an arbitrary racetrack configuration; the game is no longer constrained to use the hard-coded initial layout. (The program could also be extended to support ability to add additional Non-Player Cars, but this is not a requirement.) Note also that with the addition of the factory method pattern for creating pylons and fuel cans, the program can easily be extended to support dynamic creation of new, more complex types of track objects.

- Command Enabling/Disabling

Commands should be enabled only when their functionality is appropriate. For example, the Delete 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, keystrokes, *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.

The Command design pattern supports enabling/disabling of commands. That is, enabling/disabling a command, if implemented correctly, enables/disables the GUI components which invoke it. If you used the Java "**`Abstract Action`**" implementation of the Command pattern, your commands already support this: calling **`setEnabled(false)`** on an **`Action`** attached to a button and a menu item automatically "greys-out" both the button and menu item.

9. Additional Notes

- Requirements in previous assignments related to organization and style apply to this assignment as well. Except for those functions that have been explicitly superseded by new operations in this assignment, all functions must work as before.
- Now that you have successfully implemented the Strategy pattern (in A2), you should eliminate the "Switch Strategy" GUI button. Instead, you should add code to Non-Player Cars to cause them to switch to a different strategy automatically every, say, 10 seconds. This can be done by having the NPC's *move()* method keep track of how many times it has been called, executing the switch when 10 seconds have elapsed.
- The **`draw()`** method in an object is responsible for checking whether the object is currently "selected". If unselected, the object is drawn normally; if selected, the object is drawn in "highlighted" form. A simple method of doing "highlighting" is to draw the object in "reverse video color" by fetching each of the RGB components of the object's color, subtracting each of those from 255, and using the resulting values as the highlighted RGB drawing values. You are welcome to use a fancier highlighting scheme if you wish.
- You may if you want use the **`Graphics`** method **`drawImage()`** to render your game objects. This is not a requirement (but it will make your game look LOTS more interesting). Note that if you do this, you still must have some way of distinguishing *selected* from *unselected* objects. A common technique for this is to simply have TWO images for each object – one the "unselected" version and a second which is the "selected" version. As with sounds, you may use any images you want as long as we can show the game to the Dean and your mother (except that you may not use copyrighted images).
- As before, the origin of the "game world" is considered to be in the *lower left.* However, since the Y coordinate of a **`JPanel`** grows *downward*, "up" (north) in your game will be "down" on the screen (that is, a car that is moving north (heading = 0) moves *down* the screen). The game will therefore be "upside down". Leave it like this – we will fix it in A4.
- The simple shape representations for game objects will produce some slightly weird visual effects in the map view. For example, rectangles 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 subsequent assignment.
- 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.
- You may add additional features to your game as long as they do not conflict with the specifications. For example, fuel cans could have an "expiration date" after which they start to leak and eventually disappear. Another example would be to add more complex strategies to NPCs (as long as they are implemented using the Strategy pattern). All such enhancements are optional, and you shouldn't spend time on them until you have the basic program working – but such things can produce a more playable game if you have the time to work on them.
- **`MouseEvents`** contain a method **`isControlDown()`** which returns a **`boolean`** indicating whether the **`CTRL`** key was down when the mouse event occurred. See the **`MouseEvent`** class in the Java API JavaDoc for further information.

- You may "hard code" knowledge of frame and panel sizes into your program. Hard-coding such sizes make determining things like location boundaries easier to accomplish, although it makes the program a bit less flexible in the long run.
- Your sound files must be included in your SacCT submission. File names in programs should always be referenced in *relative-path* and *platform-independent* form. DO NOT hard code some path like "C:\MyFiles\A3\sounds" in your program; this path ***will not exist*** on the machine used for grading. A good way to organize your sounds is to put them all in a single directory named "**sounds**" in the same directory as your a3 package folder (not *inside* the "a3" folder; at the *same level* as the "a3" folder), and reference them using "relative path" notation, starting with "**.**" Also, each "slash" in the file name path should be independent of whether the program is running under Windows, Linux, or MacOS (so don't put a hard-coded "\" or "/" in your file names; instead, use the Java constant "**File.separator**"). Thus, to specify a file "**crash.wav**" in a directory "**sounds**" immediately below the current directory, use:

```
String slash = File.separator ;       // predefined Java constant
String crashFileName = "." + slash + "sounds" + slash + "crash.wav" ;
```

- 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.
- Another problem you will need to solve is that when two objects collide, the collision may be detected repeatedly as one object passes through the other. This is complicated by the fact that more than two objects can collide simultaneously. One straight-forward way of solving this is to have each collidable object keep a list of the objects that it is colliding with. An object can then skip the collision handling for objects it is already colliding with. Of course, you'll have to remove objects from that list when they are no longer colliding.
- Another problem that can occur with collision handling is that nested iterators will throw exceptions if you attempt to remove an object from the collection while the iterator is active (this generates a *"concurrent modification exception"* error). To solve this, **mark** each object which needs removal (or add it to a list), and then after all collisions have been handled, *then* go back and remove them.
- 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.
- As before, your code must be contained in a package (folder) named (in this case) "**a3**", and it must be runnable with the command "**java a3.Starter**".

Deliverables

Submitting your program requires similar steps as for A1/A2:

(1) create a single ZIP file containing your source code (.java) files, your compiled code (.class) files in package (folder) "a3", *and your sound files contained in a folder named "sounds" which in turn is contained in the same folder as the one containing the "a3" package folder*;

(2) *verify your submission* by unzipping your ZIP file to an empty folder and executing the program using the command **java a3.Starter** ; and then

(3) upload your ZIP file to SacCT. It is not a requirement to submit a UML diagram this time.

Note that it is important to include your sound files *in the proper folder* in your zip file, and that step (2) – *verifying your submission* – is especially critical for this assignment since it will verify that you have your sound files properly placed. Failure to test (verify) your zip file will incur a substantial penalty.

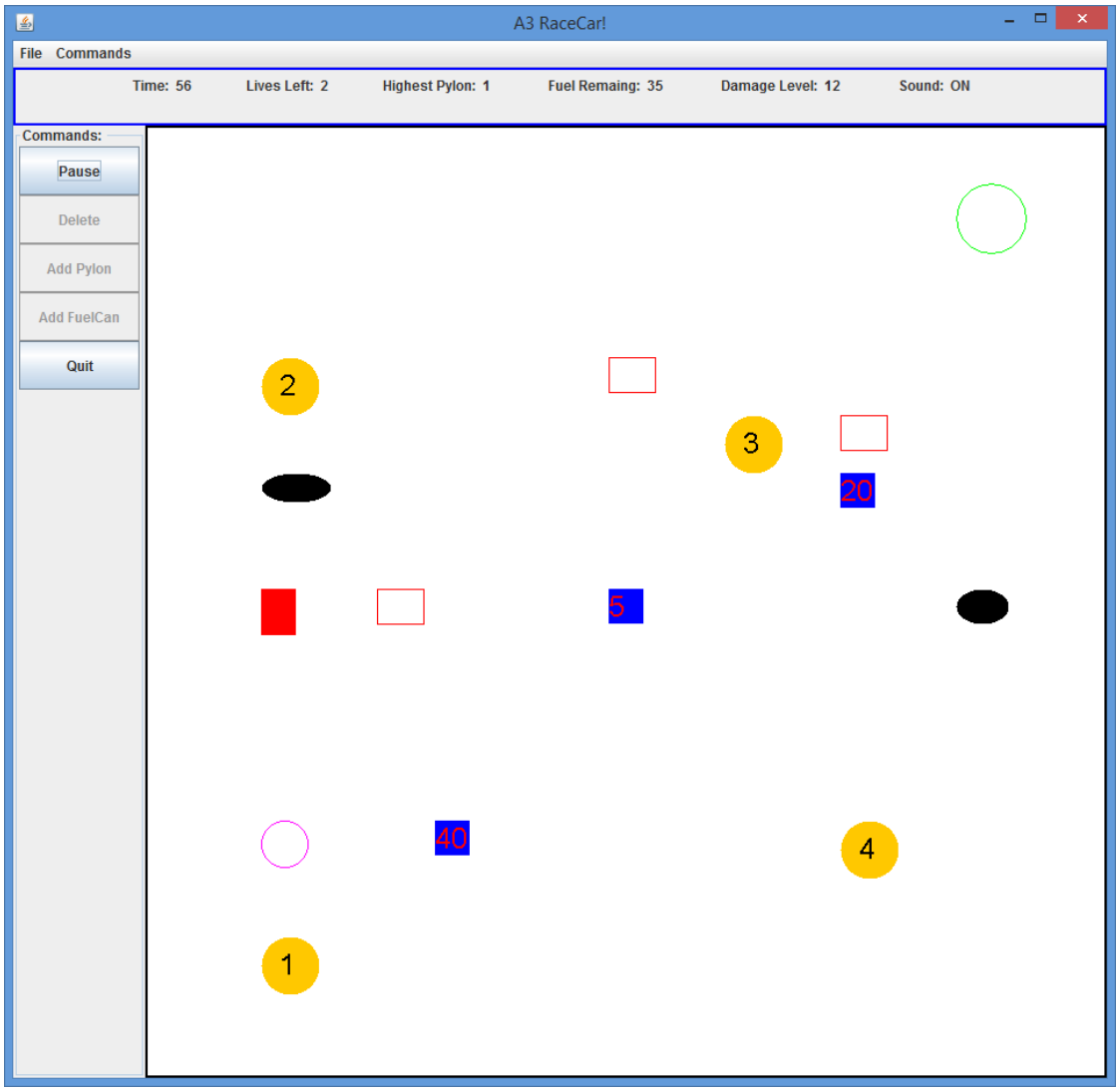
Also include in your ZIP a text file called "readme" in which you describe any changes or additions you made to the assignment specifications (for instance, if you bind any additional keys or add extra buttons, explain those in your readme file). Also include a description of any additional strategies or other enhancements in your readme file.

Be sure to keep a *backup copy* of all submitted work. All submitted work must be *strictly* your own.

Appendix 1: Sample GUI

The following shows an example of how a completed A3 game might look. It has a control panel on the left with the required command buttons (including several *disabled* buttons 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 score view panel showing the current game state (as in A2), and a new map view panel in the middle containing four orange labeled pylons (filled circles), three blue labeled fuel cans (filled squares), one player car (filled red rectangle) and three non-player cars (unfilled red rectangles), two oil slicks (black ovals), and two birds (unfilled circles).

Although you can't tell from the static image, the left bird is moving *south* (toward the *top* of the screen and the player's car), while the right bird is moving *north-west* (toward the *lower left* of the upside-down view). The cars are moving in various directions, but again with respect to north being at the *bottom* of the screen. We will fix the upside-down orientation in Assignment 4.



JC/PM:jc
3/21/15