CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
Fall 2012
John Clevenger

Assignment #3: Interactive Graphics and Animation

Due Date: Wednesday, November 21st [2 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) a new kind of game object, a ***Bird***, is to be added to the game,
(3) 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 include sounds appropriate to collisions and other events, and
(6) 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 which 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, **`MapView`** will display 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 **`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`**. 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

Each game object is to have 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)`**. 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 assignments).

Each object's **draw()** method draws the object in its current color and size, at its current location. Each concrete game object is to have a unique shape. 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.

The different types of game objects must be distinguishable when they are drawn. The drawing methods of the **Graphics** class determine what kinds of graphical representations can be used for the various game objects. **Graphics** provides routines for drawing four basic shapes: rectangles, squares (which are just rectangles with equal width and height), ovals, and circles (ovals with equal width and height); these shapes can be drawn in either *filled* or *unfilled* form. The following table shows a suggestion for using different shapes to differentiate between game objects:

Graphics shape	Filled	Unfilled
Rectangle	Log	Truck
Square	Rock	Car
Oval	Turtle	Lily Pad
Circle	Frog	Bird

Note that the above is not a requirement; you may use any shapes you like as long as they are unique for each class of **GameObject**. For example, you might want to investigate the use of **Graphics** methods **drawPolygon()** and **drawImage()**. (Note: we will be changing the representation of the **Bird** class later, so don't spend any time figuring out a fancy representation for birds.)

2. Animation Control

The **Game** class 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()** 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 required 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 *msec* or so). This means that the Tick command must be modified to account for this faster rate when it updates the displayed value for "Elapsed Time" in the **ScoreView** (basically, it must count the number of *ticks* which have occurred and only increment the Elapsed Time when a full second's worth of ticks have been seen). Also, in order for each object to know how far it should move, each timer tick should pass an "elapsed time" 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 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 time values.

In addition to all the game objects from previous assignments, this assignment is to include a new game object class: **Bird**. Birds are *moveable* and *drawable* game objects whose *direction* is a random compass heading in integer degrees and always in the range 90-270 (meaning Birds always fly in a more-or-less southerly direction). Note that this is different from the constraints previously applied to other game objects, which limited direction to one of North, South, East, or West. Birds also have an attribute *range* which specifies the diameter of a circle on the ground which the bird can see (with the center of the circle being directly below the bird; any prey (such as a Frog) which comes within this circle will be captured and eaten by the bird.

In the previous assignments the user had to add various objects (Rocks, Logs, Turtles, etc.) to the world manually. For this assignment the program should add those game objects automatically. When the game starts the initialization code should automatically add several rocks, cars, and trucks. Thereafter, the *Timer* routine should periodically add new logs and turtles, as well as adding additional cars and trucks, so that the game plays smoothly from the start. This means the various "Add" buttons are no longer needed and should be eliminated.

The Timer routine should also periodically add a new Bird to the game. Birds should be added frequently enough that no more than about 10 seconds elapses with no Bird in the game, but not so frequently that there are too many Birds (this is because Birds represent a danger to the Frog; too many of them will make the game unbeatable). A good thing to do would be to use a random number generator to vary the time between adding birds, but this is not a requirement; you may for example add a Bird every, say, 10 seconds. Birds should always be added to game at a random location along the *top (northern) edge* of the game (meaning their initial X coordinate is chosen randomly within the width of the MapView panel and their initial Y coordinate is always the value of the height of the MapView panel). Thereafter the bird flies more-or-less southerly, capturing any Frog that falls within its range (except that Frogs on Lily Pads are protected from Birds). (See below regarding the orientation of "up" on the panel.)

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 numbered buttons ("1-Car Collision", "4-Hop Onto Log" , etc.), and the objects involved in the collision were chosen arbitrarily. Now, the type of collision will be determined automatically during collision detection, so the numbered collision 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 instead 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) is to be implemented as implied by previous assignments:

- Frog collision with Car, Truck, Rock, Bird, or occupied Lily Pad: the frog dies.
- Frog collision with Log or Turtle: the frog is marked "safe" and has his speed, direction, and location attributes updated from the Log or Turtle, as in the previous assignment.
- Frog collision with empty Lily Pad: frog is marked "safe" and has his attributes updated to match the Lily Pad, as in the previous assignment.

Each time the frog "hops" the corresponding command should mark the frog as "in danger" if he is located anywhere in one of the river channels or is out of bounds, and "safe" if he is anywhere else (he may not survive the "safety" of the road or dirt, but he's safe when he first hops there – until collision detection is applied). At the end of the collision detection loop if the frog is not marked "safe" then the frog has died. This solves the problem that some deadly actions (for example, hopping into the water or out of bounds) do not result in an identifiable "collision" with anything.

Some collisions also generate a *sound* (see below). There are more hints regarding collision detection in the notes below.

4. Sound

The game must implement short, clearly different sounds for *at least* the following situations:

(1) when the frog hops,
(2) when the frog dies,
(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 and your mother (in other words, disgusting or obscene sounds are not allowed). Short, unique sounds tend to improve game playability by avoiding too much confusing sound overlap. Do not use copyrighted sounds.

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 (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 impossible in play mode.

An object is selected by *clicking* on it with the mouse (recall that *click* has a specific meaning: mouse press and release with no motion in between). Clicking on an object normally selects that object and “unselects” all other objects. However, clicking on several objects in succession *with the control (**`CTRL`**) key down* should cause *each* of the clicked objects to become “selected” (*as long as they are “selectable”*). Clicking in a location where there are no objects causes all objects to become unselected. You may implement rubber-band selection if you wish (for example, dragging a rectangle around a group of objects to select them), but it is not required.

For this assignment only floaters (logs and turtles) and vehicles (cars and trucks) are selectable; the selection process should be ignored by any other kind of object. The GUI should display a new button “DELETE” that invokes a new command object which deletes any currently selected objects. You may also want to add a key binding for the “Delete” key which invokes the same command object, but this is not a requirement.

- Command Enabling/Disabling

Commands should be enabled only when their functionality is available to be used according to the specifications of the assignment. For example, the Delete command should be disabled while in play mode; likewise, commands that involve playing the game (e.g. hopping the frog) 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 the menu item.

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.

- 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. You may choose the nature of the highlighting.
- **`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.
- The "Add Frog" and "Hop" commands are all now handled by keyboard input and are no longer needed; they can be eliminated from the GUI.
- Your sound files must be included in your 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`**" within the same directory as your Starter class, and reference them using "relative path" notation, starting with "**`.`**" Further, 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" ;
```

- As before, the origin of the "game world" is considered to be in the *lower left.* Note however that since the Y coordinate of a **`JPanel`** grows *downward*, "up" in your game will be "down" on the screen (that is, the lily pads will be at the bottom of the screen, as shown in the attached image). Your game will be upside down. Leave it like this -- we'll fix it in assignment #4.
- 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.
- Because the sound can be turned on and off by the user (using the menu option), and also turns off and on 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 also other sequences to test.
- When the game is over, animation should stop, and a dialog box should display showing the player's final score.
- One problem which can occur is the need to remove objects from the game world under certain collision situations. However, nested iterators will throw exceptions if you attempt to remove an object from their collection while the iterator is active. The solution to this is during the collision detection phase to *mark* each object which needs removal, and then after all collisions have been handled to go back and do the removal.
- You should tweak the parameters of your program for playability after you get the animation working. Things that impact playability include the screen size, objects sizes and speeds, collision distance, and the initial numbers of objects.
- As before, your code must be contained in a package named (in this case) "**`a3`**", and it must be runnable with the command "**`java a3.Starter`**".

Deliverables

As with previous assignments, submission is to be done using ***SacCT***. Submit a single "ZIP" file containing both the *Java source code* and also the *compiled (".class")* files for all the classes in your program, *plus your sound files contained in the proper subdirectory*. As always, be sure to keep a *backup copy* of your work. All submitted work must be *strictly* your own.

Sample GUI

The following shows an example of what your game GUI might look like. Notice that it has a control panel on the left with the required command buttons, "File" and "Commands" menus, a score view panel showing the current game state, and a map view panel containing one frog (green unfilled circle), three lanes of vehicles (unfilled black rectangles for trucks and squares for cars), one row of rocks (filled blue squares), three channels of floaters (filled magenta rectangles for logs and yellow ovals for turtles), one row of 5 orange lily pads, and one red unfilled circle representing a bird flying through the air. For clarity, lines have been drawn showing the lanes (you are not required to do this, although you may). Note that the bird is not constrained to remain inside lanes like the other game objects. Note also that the world is "upside down" in this view (we will fix this in a subsequent assignment), and that the game is currently "paused" (as indicated by the fact that the top command button says "Play").

JC:jc
11/7/12