CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
Fall 2015 / Dr. Muyan

Assignment #3: Interactive Graphics and Animation

Due Date: Tuesday, November 17th

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 will 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) the movement (animation) of game objects are 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 A2 properly, your program included an instance of a **`MapView`** class that is an observer that displayed the game elements on the console. It 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 should 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 the 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`**. 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 its observable; it gets invoked via a call to its **`update()`** method as before; etc.).

- Map View Graphics

As specified in A1, each game object has its own different graphical representation (shape). Dogs are circles, cats are equilateral triangles, and nets are squares. 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)`**. 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 and size, 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 ovals.

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 the **`actionPerformed()`** method (also in the **`Game`** class). **`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 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). 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 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.

3. 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 buttons ("fight" or "kitten"), and the objects involved in the collision were chosen arbitrarily. Now, the type of collision will be detected automatically during collision detection, so the "fight" and "kitten" 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 as before. When two cats collide, a kitten is created nearby. When a cat collides with a dog, the dog is scratched, its color changes, and its speed is reduced by 1. Collisions also generate a *sound* (see below). As before, nothing needs to happen when a dog collides with another dog. 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 four situations:

(1) when a cat collides with a dog (such as cat "hissing" or a dog bark),
(2) when two cats collide and produce a kitten (such as a "meow"),
(3) when the net scoops animals (such as a different "bark" or a mechanical sounds), and
(4) some sort of appropriate background sound that loops continuously during animation.

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, 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.

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 Dog Catcher. The game is to have two modes: "*play*" and "*pause*". The normal game play with animation as implemented above is "play" mode. But when 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 (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 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 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 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 *dogs* 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. However, 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"). Clicking in a location where there are no objects causes all objects to become unselected.

A new ***Heal*** command (Action) is to be added to the game, invocable from a new "Heal" GUI button. When the heal command is invoked, all selected dogs are to be cured of their

scratches, and revert to their original color and speed. The heal 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 Heal command should be disabled while in play mode; likewise, commands that involve playing the game (e.g. moving the net or scooping) 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. For example, you could use **`drawOval()`** for selected dogs, and **`fillOval()`** for unselected dogs. Consider using **`fillPolygon()`** for cats.
- 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*, moving an object "upwards on screen" (moving "north", direction = 0) in your game will require you to decrease the Y coordinate of that object (likewise to move the object down, you need to increase its Y coordinate). Decreasing the Y coordinate to move north (and increasing the Y coordinate to move south) is unintuitive (i.e. the world is "upside down"), but 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, squares 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 A4.
- 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\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

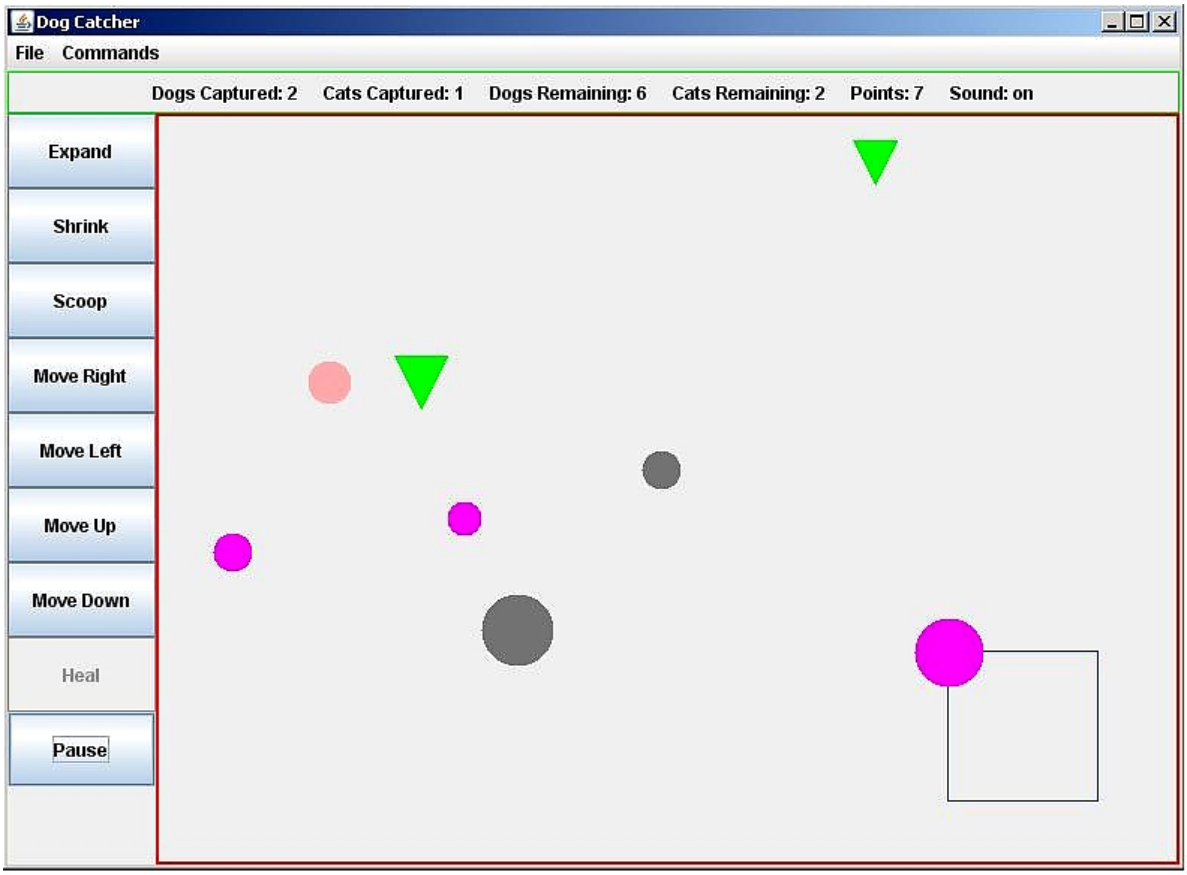
constant "**File.separator**"). Thus, to specify a file "**meow.wav**" in a directory "**sounds**" immediately below the current directory, use:

```
String slash = File.separator ;    // predefined Java constant
String meowFileName = ".” + slash + “sounds” + slash + “meow.wav” ;
```

- **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.
- 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 there are no more dogs remaining in the panel, the game is over. Animation should stop, and a dialog box should display showing the player's final score.
- You may discover that newly-born kittens quickly bump into their parents, generating more kittens, producing a chain-reaction of kitten production. One simple way of controlling this is to add a boolean flag to the **Cat** class indicating whether the cat is a kitten or not. A newly-created cat can start out as a kitten, and then after some fixed number of moves, becomes a cat. Then, your collision handling can ignore collisions between cats when one of them is a kitten. You are free to find some other solution.
- Another problem you will need to solve, is that when two cats collide (or when a dog collides with a cat), the collision will be detected repeatedly as one object passes through the other object. 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 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.
- You should tweak the parameters of your program for playability after you get the animation working. Things that impact playability include the screen size (e.g. 700x600), range of objects sizes (e.g. between 20 and 50), speed of moving the net (e.g. move 5 pixels in each clock tick), and etc. You should also specify the maximum number of cats allowed in the GameWorld (e.g. 30 cats) and should not create new kittens if this maximum number is already reached.
- As before, you may not use a "GUI Builder" for this assignment.
- As before, your code must be contained in a package named (in this case) "**a3**", and that it be runnable with the command "**java a3.Starter**".

Appendix: 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* Heal 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 "Command" menus, a score view panel showing the current game state, and a map view panel in the middle containing *6 dogs* (circles with variations in color based on number of scratches); *2 cats* (green triangles); and the *net*. Note also that the world is "upside down" -- the triangles for cats are facing down and to move the net up, we actually decrease the Y coordinate of the net (and to move the net down, we actually increase the Y coordinate of the net). We will fix this in A4.



Deliverables

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

1. Create a *single* file in "ZIP" format containing (1) the *Java source code (".java")* for all the classes in your program, and (2) the compiled (".class") files for your program *(3) your sound files contained in the proper subdirectory* (UML is not needed). Be sure your ZIP file contains the proper subpackage hierarchy. Also include in your ZIP a text file called "readme" in which you describe any changes or additions you made to the assignment specifications.

2. *Verify your submission file.* To do this, copy your ZIP file to an empty folder on your machine, unzip it, open a command prompt window, change to the folder where you unzipped the file, and type the command "**java a3.Starter"**. If this does not properly execute your program (e.g. sound does not work), your ZIP file is not properly structured; go back to step (1) and fix it. Substantial penalties will be applied to submissions which have not been properly verified according to the above steps.

3. Login to ***SacCT***, 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!