CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

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

Assignment #3: Interactive Graphics and Animation

Due Date: Friday, April 20th [2½ 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 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

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)`**. 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. 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.

Bricks should be represented as solid colored rectangles with a different-colored border (note that you can draw a "bordered rectangle" by first drawing a filled rectangle and then drawing an unfilled rectangle of the same size in a different color over the top of it). Speed bricks should be distinguishable from regular bricks in some way (for example by being a different colored or having a different-colored border). Balls should be represented as a circle. The paddle should be represented as an elongated oval or as a round-edged rectangle. Edges should be represented as narrow rectangles. The wall (that is, the collection of bricks) should be located near the top part of the panel (but not touching the top edge). There should not be gaps between the bricks in the wall and there should not be gaps between the brick wall and the left or right edges. The paddle should be located near the bottom edge, and the initial location for the ball should be lower than the wall but higher than the paddle. (However, see below regarding the orientation of "up" on the panel.)

- Bonus Bricks

In this assignment your program is to support a new ability: converting bricks into <u>*bonus bricks*</u>. Bonus bricks are identified by special highlighting (for example, being a different color from either Regular or Speed Bricks, or being marked with a special indicator) when they are drawn, and they are worth twice as many points as normal bricks when the ball hits them. Both Regular Bricks and Speed Bricks can become Bonus Bricks, and this conversion can be reversed (that is, a Bonus Brick can be converted back to its former kind (Regular or Speed Brick). In other words, a Bonus brick is a brick whose characteristics are augmented *at runtime*; this implies that the *Decorator* design pattern should be used to implement Bonus Bricks (see below for further details).

<u>2. Animation Control</u>

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 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 time values.

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 ("Annihilate Brick", "Paddle Hit", 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 "Annihilate", "Side Hit", "Top Hit", "Brick Hit", and "Paddle Hit" 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:

- Ball vs. Side: the ball starts moving in the opposite horizontal direction (the 's' command from Assignment 1).

- Ball vs. Upper Edge: the ball starts moving in the opposite vertical direction (the 'u' command).

- Ball vs. Brick: the ball starts moving in the opposite vertical direction, the speed of the ball is increased 20% if the brick is a speed brick, the point value of the brick is added to the score, and the brick is removed from the world (a combination of the 'b', 'i', and 'a' commands).

- Ball vs. Paddle: the ball starts moving in the opposite vertical direction. In addition, if the point of collision is within ¼ of the width of the paddle from either the left or right side of the paddle, a small random change is added to the ball's horizontal motion. Note that this combines the effects of the 'p' and 'c' commands from Assignment 1. Note also that there is no longer a need for keeping track of the "mode" of the paddle; the effect of being in "random mode" occurs when the ball strikes near the edge of the paddle.

- Ball vs. (Invisible) Bottom Edge: The ball is lost (discarded). If there are balls remaining, a new ball is put into play; otherwise, the game is over (the 'n' command).

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 particular, clearly different sounds for *at least* the following situations:

(1) when the ball hits the paddle,
(2) when the ball hits a brick,
(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, 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.

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 bricks are selectable; the selection process should be ignored by any other kind of object. Selected bricks can be decorated using new Apply Bonus and Undo Bonus commands (see below).

- Command Enabling/Disabling

Commands should be enabled only when their functionality is appropriate. For example, the Apply Bonus and Undo Bonus commands should be disabled while in play mode; likewise, commands that involve playing the game (e.g. moving the paddle) 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.

Bonus Brick Implementation

In order to implement the Bonus Brick functionality, two new commands will need to be added as described below. Each command is to be invokable from a new appropriately-labeled GUI button and also from a Command menu item. The new commands should be implemented using the command pattern and should be enabled only while in paused mode.

- **Apply Bonus** – converts the selected bricks (if any) to bonus bricks. Apply Bonus works by wrapping a Bonus Brick around a selected brick (that is, *decorating* a normal brick) at runtime, removing the original brick from the world and replacing it with the decorated BonusBrick:

  ```
  BonusBrick bb = new BonusBrick(theSelectedBrick);
  remove theSelectedBrick from the world;
  add bb to the world;
  ```

  If a selected brick is already a Bonus Brick then Apply Bonus has no effect on it.

- **Undo Bonus** – converts all the bricks which were converted to Bonus Bricks by the most recent "Apply Bonus" command back to their original brick type (that is, removes the decoration on the set of most-recently decorated Bonus Bricks). Multiple "undo" operations must be supported, and the Undo operation must be invokable from either the GUI button, the corresponding menu item, or by holding down the CTRL key and pressing 'Z'. Use the Memento design pattern to implement this functionality, as described under "Additional Notes" below.

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.

- In this assignment, the user controls the paddle while in Play mode using the mouse. When the mouse is moved left or right without any button pressed, the paddle is moved to the left or right respectively. Paddle movement should be constrained so that it stops at the left/right sides.
- 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.
- 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 program files, 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 paddle will be at the top 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.

- The new class **`BonusBrick`** must be able to decorate both Regular Bricks and Speed Bricks. Following the Decorator pattern implies that the new class must derive from the same superclass as both Regular and Speed Bricks (or equivalently, must implement the same interface), must define a constructor which accepts and saves an argument of type **`Brick`**, and must implement **`IDrawable`** by providing a **`draw()`** method which draws its saved brick and also draws the "decoration" for the brick. A simple decoration would be to draw an extra-wide line around the edge of the brick in a different color.

- The appropriate way to implement the Undo operation is to use the Memento design pattern. Specifically, each Apply Bonus invocation should first create a "memento" containing the current state of the brick wall and save that memento in a state list. Apply Bonus can then alter the wall contents however it needs to. Subsequent invocation of "Undo" would then fetch the most recent wall state memento and rebuild the wall from that memento, restoring the old state. Each memento describing a wall state contains a description of each brick in the wall; each brick description contains the relevant attributes for that brick (which thus allows rebuilding the brick). (Note that brick descriptions themselves could in turn be implemented as mementos, but this is not a requirement).

- Note that CTRL-Z can be stored in an **`InputMap`** as a **`KeyStroke`** object obtained by calling the **`KeyStroke`** method **`getKeyStroke()`** as follows:

```
import javax.swing.KeyStroke;
import static java.awt.event.InputEvent.CTRL_DOWN_MASK;
import static java.awt.event.KeyEvent.VK_Z;
...
KeyStroke ctrlZ = KeyStroke.getKeyStroke(VK_Z, CTRL_DOWN_MASK);
```

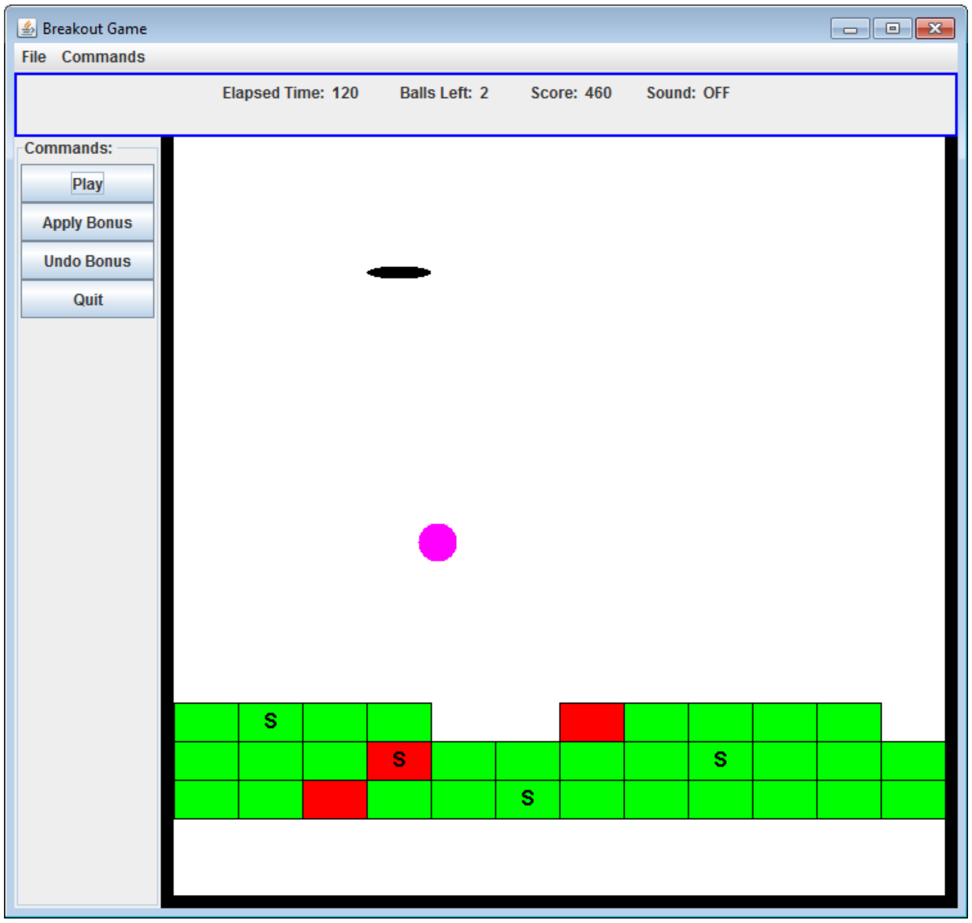
- You may "hard code" knowledge of frame and panel sizes into your program. Hard-coding such sizes make determining things like location boundaries easier to accomplish, although it makes the program a bit less flexible in the long run.
- Because the sound can be turned on and off by the user (using 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 “Command” menus, a score view panel showing the current game state, and a map view panel in the middle containing three visible edges (narrow black rectangles), the paddle, the ball, and a wall made up of a number of bricks. Some bricks have already been “annihilated” due to ball collisions, some bricks have been converted to “bonus bricks” (they are different color, although this may be hard to see if you are looking at a black-and-white copy), and some bricks are marked with “S” indicating they are speed bricks. 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
4/3/12