Project 1d: Breakout

CS Y11

Tuesday 19th February, 2019

CLICK HERE FOR FILES NEEDED

Your job in this assignment is to write the classic arcade game of Breakout, which was invented by Steve Wozniak before he founded Apple with Steve Jobs. It is a large project, but entirely manageable as long as you break the problem up into pieces. The decomposition is discussed in this writeup, and there are several suggestions for staying on top of things in the "Strategy and tactics" section.
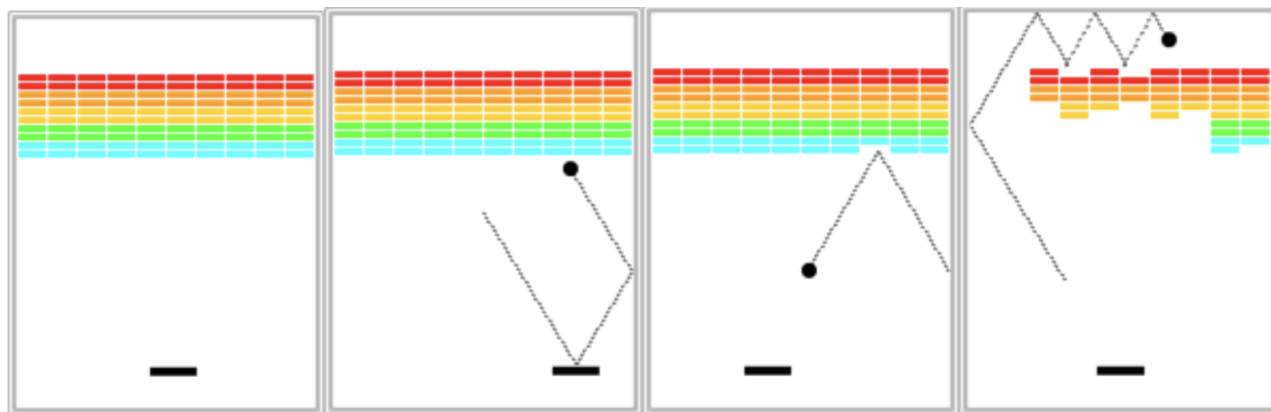

Figure 1: Game in action

Warm-up: Prime Checker (Optional)

A key challenge when writing large programs like Breakout is how best to decompose our solutions into manageable and effectively-implemented methods. In order to practice this skill, youll begin this assignment by writing a short method that takes in a positive integer greater than 1 as an input and returns a boolean indicating whether or not that integer is prime. As a reminder, a prime number that is defined such that its only factors are 1 and itself. In PrimeChecker.java, we provide a run method that tests whether or not a series of numbers are prime. Your job is to implement the isPrime method to check whether or not the number is prime.

Warm-up: Mouse Reporter (Optional but highly recommended)

To get you warmed up, first write a minimal program that leverages the essential concepts needed for Breakout.Write a **MouseReporter** that creates a GLabel on the left side of the screen. When the mouse is moved the label is updated to display the current x, y location of the mouse. If the mouse is touching the label it should turn **red**, otherwise it should be **blue**.

You should take advantage of the **setLabel** method that can be called on a **GLabel**. If you look in the documentation at the methods that are defined at the **GraphicsProgram** level, you will discover that there is a method

public GObject getElementAt(double x, double y)

that takes a position in the window and returns the graphical object at that location, if any. If there are no graphical objects that cover that position, **getElementAt** returns the special constant null. If there is more than one, **getElementAt** always chooses the one closest to the top of the stack,

which is the one that appears to be in front on the display. The starter code for **MouseReporter** stores the label as an instance variable and adds it to the screen. See figure 2 for an example.
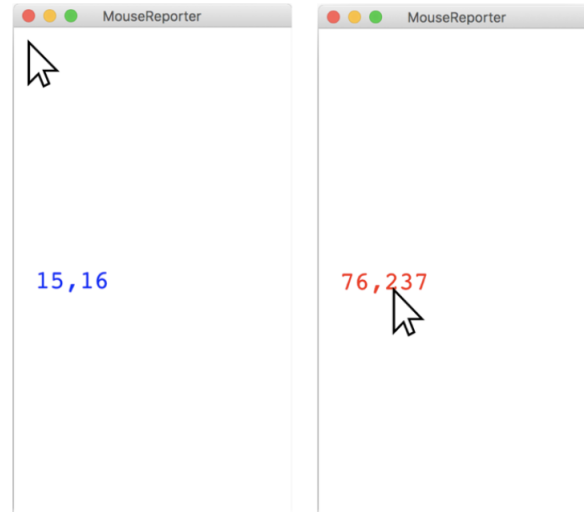

Figure 2: MouseReporter in action

1 Part I

1.1 Game Description

In Breakout, the initial configuration of the world appears as shown in the first in **Figure 1**. The colored rectangles in the top part of the screen are bricks, and the slightly larger rectangle at the bottom is the paddle. The paddle is in a fixed position in the vertical dimension, but moves left and right across the screen along with the mouse until it reaches the edge of its space.

A complete game consists of three turns. On each turn, a ball is launched from the center of the window toward the bottom of the screen at a random angle. That ball bounces off the paddle and the walls of the world, in accordance with the physical principle generally expressed as "the angle of incidence equals the angle of reflection" (which turns out to be very easy to implement as discussed later in this writeup). Thus, after two bounces one off the paddle and one off the right wall the ball might have the trajectory shown in the second diagram. (**Note that the dotted line is there to show the balls path and wont appear on the screen.**)

As you can see from the second diagram, the ball is about to collide with one of the bricks on the bottom row. When that happens, the ball bounces just as it does on any other collision, but the brick disappears. The third diagram shows what the game looks like after that collision and after the player has moved the paddle to put it in line with the oncoming ball. The play on a turn continues in this way until one of two conditions occurs:

1. The ball hits the lower wall, which means that the player must have missed it with the paddle. In this case, the turn ends and the next ball is served if the player has any turns left. If not, the game ends in a loss for the player.

2. The last brick is eliminated. In this case, the player wins, and the game ends immediately.

After all the bricks in a particular column have been cleared, a path will open to the top wall. When this situation occurs, the ball will often bounce back and forth several times between the top wall and the upper line of bricks without the user ever having to worry about hitting the ball with the paddle. This condition is called "breaking out" and gives meaning to the name of the game. It is important to note that, even though breaking out is a very exciting part of the players experience, you dont have to do anything special in your program to make it happen. The game is simply operating by the same rules it always applies: bouncing off walls, clearing bricks, and otherwise obeying the laws of physics.

1.2   Starter File

Take a look at the initial contents of **Breakout.java** file. This file takes care of the following details:

- It includes the imports you will need for writing the game.

- It defines the named constants that control the game parameters, such as the dimensions of the various objects. Your code should use these constants internally so that changing them in your file changes the behavior of your program accordingly.

Success in this assignment will depend on breaking up the problem into manageable pieces and getting each one working before you move on to the next. The next few sections describe a reasonable staged approach to the problem.

1.3   Set up the bricks

Before you start playing the game, you have to set up the various pieces. Thus, it probably makes sense to implement the *run method as two method calls*: one that sets up the game and one that plays it. An important part of the setup consists of creating the rows of bricks at the top of the game, which look like this:
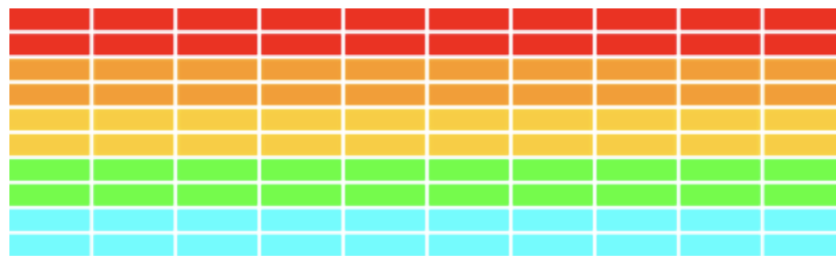

Figure 3: Yours can look different, but it should have atleast 5 different colors

The number, dimensions, and spacing of the bricks are specified using named constants in the starter file, as is the distance from the top of the window to the first line of bricks. The only value you need to compute is the x coordinate of the first column, which should be chosen so that the bricks are centered in the window, with the leftover space divided equally on the left and right sides. The color of the bricks remain constant for two rows and run in the following rainbow-like sequence: RED, ORANGE, YELLOW, GREEN, CYAN.
Look at this presentation which explains how the graphics library that you will be using works.

1.4   Create the Paddle

The next step is to create the paddle. At one level, this is considerably easier than the bricks. There is only one paddle, which is a filled **GRect**. You even know its position relative to the bottom of the window. The challenge in creating the paddle is to make it track the mouse. Mouse tracking makes use of the event-driven model discussed in this presentation.
Here, however, you only have to pay attention to the x coordinate of the mouse because the y position of the paddle is fixed. The only additional wrinkle is that you should not let the paddle move off the edge of the window. Thus, youll have to check to see whether the x coordinate of the mouse would make the paddle extend beyond the boundary and change it if necessary to ensure that the entire paddle is visible in the window.This entire part of the program takes fewer than 10 code lines, so it shouldnt take so long. The hard part lies in reading the Graphics presentation and understanding what you need to do.

2 Part II

2.1 Create the ball and get it to bounce off the walls

At one level, creating the ball is easy, given that its just a filled **GOval**. The interesting part lies in getting it to move and bounce appropriately. You are now past the "setup" phase and into the "play" phase of the game. To start, create a ball and put it in the center of the window. As you do so, keep in mind that the coordinates of the **GOval** do not specify the location of the center of the ball but rather its upper left corner. The mathematics is not any more difficult, but may be a bit less intuitive.

The program needs to keep track of the velocity of the ball, which consists of two separate components, which you will presumably declare as instance variables like this:

```
private double vx, vy;
```

The velocity components represent the change in position that occurs on each time step. Initially, the ball should be heading downward, and you might try a starting velocity of +3.0 for vy**(remember that y values in Java increase as you move down the screen)**. The game would be boring if every ball took the same course, so you should choose the vx component randomly. In generating random numbers , you should simply do the following:

1. Declare an instance variable **rgen**, which will serve as a random-number generator:

   ```
   private RandomGenerator rgen = RandomGenerator.getInstance()
   ```

   Remember that instance variables are declared outside of any method but inside the class.

2. Initialize the vxvariable as follows:

   ```
   vx = rgen.nextDouble(1.0, 3.0);
   if (rgen.nextBoolean(0.5)) vx = -vx;
   ```

   This code sets **vx** to be a random **double** in the range 1.0 to 3.0 and then makes it negative half the time. This strategy works much better for Breakout than calling

   ```
   nextDouble(-3.0, +3.0)
   ```

   which might generate a ball going more or less straight down. That would make life far too easy for the player.

Once youve done that, your next challenge is to get the ball to bounce around the world, ignoring entirely the paddle and the bricks. To do so, you need to check to see if the coordinates of the ball have gone beyond the boundary, taking into account that the ball has a nonzero size. Thus, to see if the ball has bounced off the right wall, you need to see whether the coordinate of the right edge of the ball has become greater than the width of the window; the other three directions are treated similarly. For now, have the ball bounce off the bottom wall so that you can watch it make its path around the world. You can change that test later so that hitting the bottom wall signifies the end of a turn.

Computing what happens after a bounce is extremely simple. If a ball bounces off the top or bottom wall, all you need to do is reverse the sign of **vy**. Symmetrically, bounces off the side walls simply reverse the sign of **vx**.

2.2 Checking for collisions

Now comes the interesting part. In order to make Breakout into a real game, you have to be able to tell whether the ball is colliding with another object in the window. As scientists often do, it helps to begin by making a simplifying assumption and then relaxing that assumption later. Suppose the ball were a single point rather than a circle. In that case, how could you tell whether it had collided with another object?What happens if you call

```
getElementAt(x, y)
```

where **x** and **y** are the coordinates of the ball? If the point **(x,y)** is underneath an object, this call returns the graphical object with which the ball has collided. If there are no objects at the point **(x,y)**, youll get the value **null**.

So far, so good. But, unfortunately, the ball is not a single point. It occupies physical area and therefore may collide with something on the screen even though its center does not. The easiest thing to do which is in fact typical of the simplifying assumptions made in real computer games is to check a few carefully chosen points on the outside of the ball and see whether any of those points has collided with anything. As soon as you find something at one of those points, you can declare that the ball has collided with that object.

In your implementation, the easiest thing to do is to check the four corner points on the square in which the ball is inscribed. Remember that a **GOval** is defined in terms of its bounding rectangle, so that if the upper left corner of the ball is at the point (x, y), the other corners will be at the locations shown in this diagram:
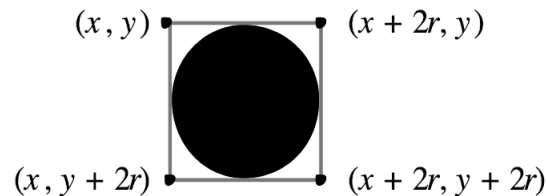
These points have the advantage of being outside the ball which means that getElementAt cant return the ball itself but nonetheless close enough to make it appear that collisions have occurred. Thus, for each of these four points, you need to:

1. Call **getElementAt** on that location to see whether anything is there.

2. If the value you get back is not **null**, then you need look no farther and can take that value as the **GObject** with which the collision occurred.

3. If **getElementAt** returns **null** for a particular corner, go on and try the next corner.

4. If you get through all four corners without finding a collision, then no collision exists.

It would be very useful to write this section of code as a separate method

```
private GObject getCollidingObject()
```

that returns the object involved in the collision, if any, and null otherwise. You could then use it in a declaration like

```
GObject collider = getCollidingObject();
```

which assigns that value to a variable called **collider**.

From here, the only remaining thing you need to do is decide what to do when a collision occurs. There are only two possibilities. First, the object you get back might be the paddle, which you can test by checking

```
if (collider == paddle) . . .
```

If it is the paddle, you need to bounce the ball so that it starts traveling up. If it isnt the paddle, the only other thing it might be is a brick, since those are the only other objects in the world. Once again, you need to cause a bounce in the vertical direction, but you also need to take the brick away. To do so, all you need to do is remove it from the screen by calling the **remove** method.

2.3   Finishing Up

If youve gotten to here, youve done all the hard parts. There are, however, a few more details you need to take into account:

- Youve got to take care of the case when the ball hits the bottom wall. In the prototype youve been building, the ball just bounces off this wall like all the others, but that makes the game pretty hard to lose. Youve got to modify your loop structure so that it tests for hitting the bottom wall as one of its terminating conditions.

- Youve got to check for the other terminating condition, which is hitting the last brick. How do you know when youve done so? Although there are other ways to do it, one of the easiest is to have your program keep track of the number of bricks remaining. Every time you hit one, subtract one from that counter. When the count reaches zero, you must be done. In terms of the requirements of the assignment, you can simply stop at that point, but it would be nice to give the player a little feedback that at least indicates whether the game was won or lost.

- Youve got to experiment with the settings that control the speed of your program. How long should you pause in the loop that updates the ball? Do you need to change the velocity values to get better play action?

- Youve got to test your program to see that it works. Play for a while and make sure that as many parts of it as you can check are working. If you think everything is working, here is something to try: Just before the ball is going to pass the paddle level, move the paddle quickly so that the paddle collides with the ball rather than vice-versa. Does everything still work, or does your ball seem to get "glued" to the paddle? If you get this error, try to understand why it occurs and how you might fix it.

3   Possible Extensions

Dont try to extend the program until you get the basic functionality working. Several of those are lots of these extensions are fun but dont start them until the basic program is working. If you add extensions too early, youll find that the debugging process gets really difficult.

1. *Add sounds.* You might want to play a short bounce sound every time the ball collides with a brick or the paddle. This extension turns out to be very easy. The starter project contains an audio clip file called bounce.au that contains that sound. You can load that sound by writing

```
AudioClip bounceClip = MediaTools.loadAudioClip("bounce.au");
bounceClip.play();
```

2. *Add messages.* The game is more playable if at the start it waits for the user to click the mouse before serving each ball and announces whether the player has won or lost at the end of the game. These are just **GLabelobjects** that you can add and remove at the appropriate time.

3. *Improve the user control over bounces.*The program gets rather boring if the only thing the player has to do is hit the ball. It is far more interesting, if the player can control the ball by hitting it at different parts of the paddle. The way the old arcade game worked was that the ball would bounce in both the x and y directions if you hit it on the edge of the paddle from which the ball was coming.

4. *Add in the "kicker."* The arcade version of Breakout lured you in by starting off slowly. But, as soon as you thought you were getting the hang of things, the program sped up, making life just a bit more exciting. As one example of this, you might consider adding this feature by doubling the horizontal velocity of the ball the seventh time it hits the paddle, figuring thats the time the player is growing complacent.

5. *Keep score.* You could easily keep score, generating points for each brick. In the arcade game, bricks were more valuable higher up in the array, so that you got morepoints for red bricks than cyan bricks. You could display the score underneath the paddle, since it wont get in the way there.

6. Use your imagination!

Thanks to Stanford's CS106A class for the idea and files for this assignment.