Department of Electrical and Computer Engineering
ECSE 202 – Introduction to Software Development
Assignment 2
Introduction to Programming in Java

Introduction

In this assignment you will use object-oriented design to build on the work you have already completed in Assignment 1. Namely, now that you have implemented the simulation of a single bouncing ball with a simple physics model, you will leverage your existing code to allow adding multiple bouncing balls to the simulation. As stated in the chapter on Objects and Classes in the Roberts text book *The Art and Science of Java*, the "idea of thinking of parts of your programs as black boxes is fundamental to the concept of object-oriented programming". Thus, you can design your program such that the code determining the bouncing of a ball resides within the "black box" of a ball class; you can then create (instantiate) as many balls as you want (with different initial parameters), and these balls can bounce separately based on the given initial parameters. For this assignment, you will rely heavily on concepts in Chapters 5 and 6 in the textbook, as well as the previous chapters.

Problem Description

For this assignment, you will randomly generate the parameters for 100 separate balls (initial x, y position of the ball, size of the ball, color of the ball, energy loss for the ball, horizontal speed of the ball). You will then simulate these 100 balls bouncing simultaneously (as you did for one ball in Assignment 1), but with each ball's simulation based on its own parameters. Note that for this simulation, the balls do not "interact" with each other (i.e. collide with each other) but simply pass through each other. Figure 1 again shows the basic algorithm that was used in Assignment 1 to determine the state of a bouncing ball at each time step in the simulation. The logic behind this algorithm remains (in fact you should be able to copy and paste most of the code that you had for assignment 1 into the appropriate place in Assignment 2, and then adjust it accordingly). However, this algorithm (and code) will now lie in a separate **ball** class, along with all the parameters (variables) that define a ball in the simulation (color, size, other initial conditions etc.)

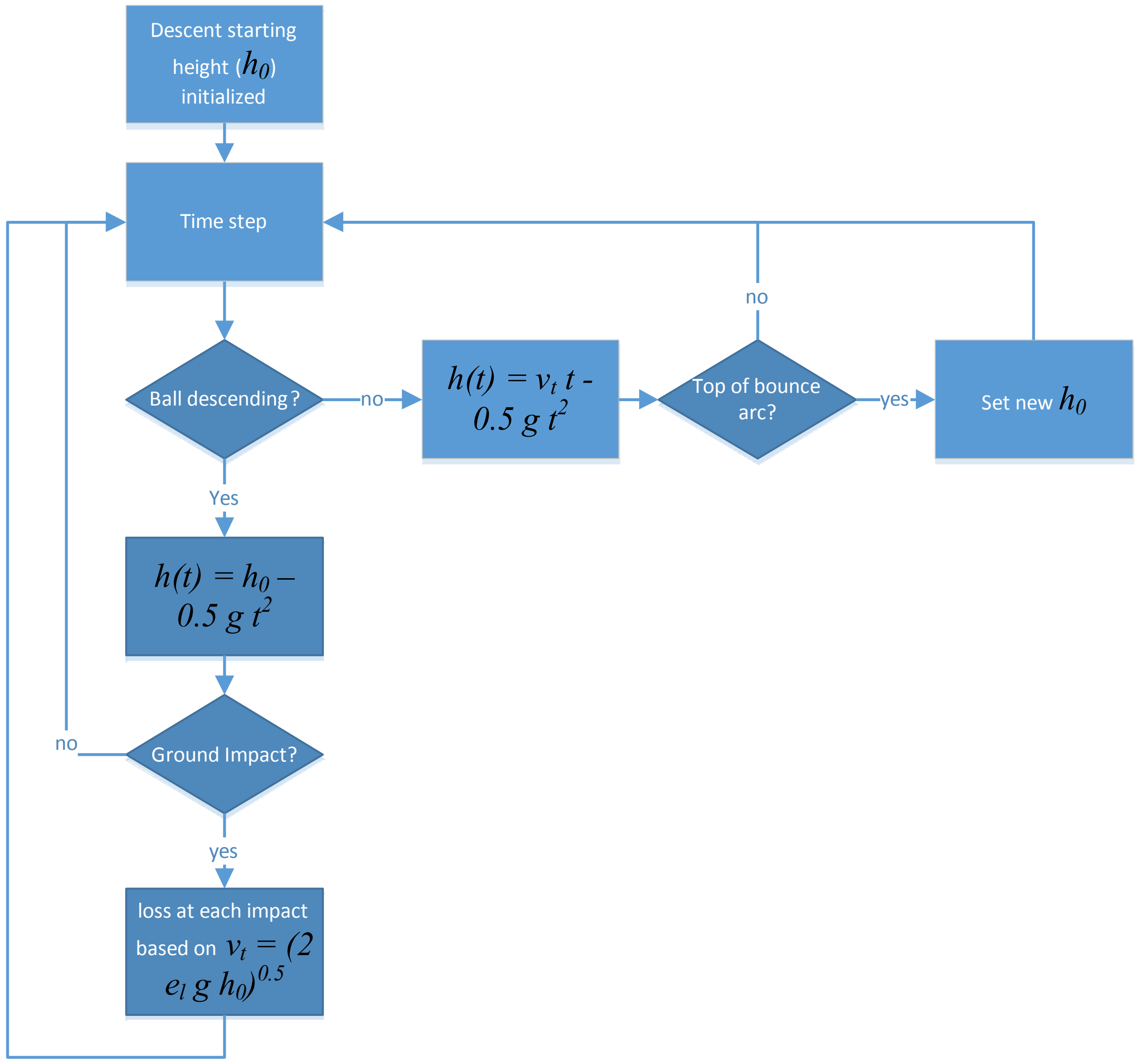
Descent starting height (h_0) initialized
Time step
Ball descending ?
no
Yes
$h(t) = h_0 - 0.5\, g\, t^2$
Ground Impact?
no
yes
loss at each impact based on $v_t = (2\, e_l\, g\, h_0)^{0.5}$
$h(t) = v_t\, t - 0.5\, g\, t^2$
Top of bounce arc?
no
yes
Set new h_0

Figure 1: Basic algorithm for simplified bouncing ball

When given a particular problem to implement, there is rarely one single way to do it. For simulating multiple bouncing balls, it is quite natural to place (code) the behavior of the ball (Figure 1) within a **ball** class. It is also natural to have a separate class (for our purposes we can call it **bSim**) that will run the simulation, including randomly generating the required parameters for the 100 balls, calling the appropriate methods to set up the graphics environment, and instantiating the 100 balls (using the **ball** class) while passing the constructor for this class the randomly generated parameters as you are creating each separate ball. But after that, there are different ways that you could implement the rest of the simulation. You could have this principal simulation class of the program also keep track of the total time (i.e. the while loop in Assignment 1 that incremented a time step and timed out after a predetermined amount of time), and call appropriate methods in the **ball** class for each of the 100 balls that update each ball to the appropriate position based on the particular's ball's current state. Another way of implementing the required simulation (the one that you will use here) is to program the

assignment in such a way that once you create an instance of a ball using the **ball** class, this instance of the ball is keeping track of time on its own and bouncing on its own within the simulation environment. Thus, the **ball** class itself will contain the while loop keeping track of global time, and each ball (instance of the **ball** class) will behave independently.
How do we allow each ball to behave independently? We take advantage of the **Thread** class in Java (https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html), and define our **ball** class to `extend` the **Thread** class. A threading is a concept that will be very important for your later courses and allows your application to have multiple threads of execution running concurrently.

Design Approach

Let us first consider the ball class, the template for which is shown below (you are to use this template in designing your implementation:

```
/**
 * This class provides a single instance of a ball falling under the
 * influence of gravity.  Because it is an extension of the Thread
 * class, each instance will run concurrently, with animations on the
 * screen as a side effect.  We take advantage here of the fact that
 * the run method associated with the Graphics Program class runs in
 * a separate thread.
 *
 * @author ferrie
 *
 */

public class gBall extends Thread {

     /**
      * The constructor specifies the parameters for simulation:
      *
      * @param Xi          double The initial X position of the center
      *                    of the ball
      * @param Yi          double The initial Y position of the center
      *                    of the ball
      * @param bSize       double The radius of the ball in simulation
      *                    units
      * @param bColor      Color  The initial color of the ball
      * @param bLoss       double Fraction [0,1] of the energy lost on
      *                    each bounce
      * @param bVel        double X velocity of ball
      */
	public gBall(double Xi, double Yi, double bSize, Color bColor,
			double bLoss, double bVel) {

		this.Xi = Xi;			// Get simulation parameters
		this.Yi = Yi;
		this.bSize = bSize;
		this.bColor = bColor;
		this.bLoss = bLoss;
		this.bVel = bVel;

	// Create instance of GOval with specified parameters

	}

	/**
	 * The run method implements the simulation from Assignment 1.
	 * Once the start method is called on the gBall instance, the
	 * code in the run method is executed concurrently with the main
	 * program.
	 * @param void
	 * @return void
	 */

	public void run() {
	// Simulation goes here...
	}
}
```

Example:

Using the **gBall** (gravity ball) class, create a simulation for a single ball, initially located at coordinates (simulation, not screen) (10,100), with size=6, Color=Red, loss coefficient=0.25, with an initial X velocity of 1.0 m/s.

```
//
//	Code to set up the graphics environment as per Assignment 1
//	goes here
//

	gBall redBall = new gBall(10.0,100.0,6.0,Color.RED,0.25,1.0);
	add(redBall.myBall);
	redBall.start();
```

Details:

The graphics program needs a reference to the GOval object created within the gBall object. Assume that myBall is declared as an instance variable as follows:

```
public GOval myBall;
```

Inside the constructor of the gBall class, an instance of GOval corresponding to a filled square is instantiated as follows:

```
myBall = new GOval(paramaters);
myBall.setFilled(true);
myBall.setFillColor(parameter);
```

Since myBall is an instance variable, it can readily be accessed using the “dot” notation as shown on the call to the add method on the previous page.

Since gBall is an extension of the Thread class, it inherits the corresponding methods – one of which is “start”. The effect of calling the start method on the gBall class instance is to call the run method associated with gBall. However, there is an important side effect to this method call. Rather than returning when the method has completed, it returns *immediately*, allowing the method to run concurrently with the calling program. In other words, the simulation embedded within the gBall instance runs in *parallel* with the main program and all other instances of the gBall class.

Another problem that we have to deal with is that we no longer have access to the pause method, used to control the speed at which the display is updated. Fortunately the Thread class includes a method called “sleep” which takes an argument of type long expressing the delay in milliseconds. We haven’t dealt with exceptions yet, so just include the following pattern in place of pause to achieve the same effect.

```
try {                              // pause for 50 milliseconds
    Thread.sleep(50);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

One more detail regarding the gBall class – stopping. When the while loop is exited, the run method terminates which results in the thread terminating. A good place to stop is when the ball runs out of energy and stops bouncing, which can easily be detected when the current height reaches the radius of the ball (if you defined the position by the center coordinates), or the ball diameter (if you defined the position by the upper left hand corner of its bounding box – this is the convention used by GOval). The effect of terminating the thread is that no further messages are sent to its corresponding ball, i.e., it stops moving).

To complete this program, we need a main class which must correspond to the template below:

```java
public class bSim extends GraphicsProgram {

// Parameters used in this program

     public void run() {

// Set up display, create and start multiple instances of gBall
     }
}
```

Your main program should set up the display as in Assignment 1. Next, define the following parameters:

```java
private static final int WIDTH = 1200;        // n.b. screen coords
private static final int HEIGHT = 600;
private static final int OFFSET = 200;
private static final int NUMBALLS = 100;      // # balls to sim.
private static final double MINSIZE = 3;      // Min ball size
private static final double MAXSIZE = 20;     // Max ball size
private static final double XMIN = 10;        // Min X start loc
private static final double XMAX = 50;        // Max X start loc
private static final double YMIN = 50;        // Min Y start loc
private static final double YMAX = 100;       // Max Y start loc
private static final double EMIN = 0.1;       // Min loss coeff.
private static final double EMAX = 0.3;       // Max loss coeff.
private static final double VMIN = 0.5;       // Min X velocity
private static final double VMAX = 3.0;       // Max Y velocity
```

The screen is laid out as a 1200 x 800 rectangle, with the ground plane sitting at 600 (offset of 200). You will run your final simulation with 100 balls (although for debugging purposes it is suggested that you use smaller values, e.g., 1). The gBall constructor needs 6 parameters – Xi, Yi, Size, Color, loss coefficient, and velocity. Use the RandomGenerator class shown in the slides to generate values for each gBall instance. For example, to generate a random loss parameter, one would use an instance of the RandomGenerator class as follows:

```java
          double iLoss = rgen.nextDouble(EMIN,EMAX);
```

Finally, to generate a simulation with NUMBALLS elements, one can easily set up a for loop that on each iteration generates a new set of random parameters, creates a gBall instance using these parameters, and starting the corresponding thread.

You might also consider writing a “helper” class with utility functions, e.g. methods for converting from simulation coordinates to screen coordinates, etc. Although this is not strictly required, it does make your code a lot easier to read and understand.

Instructions

1. Write Java classes, `bSim.java, gBall.java,` and `gUtil.java` (optionally), that implement the simulation outlined above. If you wish to implement using more classes than these three, you can do so. Do this within the Eclipse environment so that it can be readily tested by the course graders. For your own benefit, you should get in the habit of naming your Eclipse projects so that they can easily be identified, e.g., ECSE-202_A2.

To Hand In:

1. The source java files. Note – use the default package.
2. A screenshot file (pdf) showing the start of simulation (hint – you can make the sleep delay very large to have time to do the screen capture).
3. A screenshot file showing the end of simulation when all balls have stopper moving.

All assignments are to be submitted using myCourses (see myCourses page for ECSE 202 for details).

File Naming Conventions:

Fortunately myCourses segregates files according to student, so submit your .java files under the names that they are stored under in Eclipse, e.g., bSim.java, gBall.java, gUtil.java. We will build and test your code from here.

Your screenshot files should be named start.jpg and end.jpg respectively (again, this makes it possible for us to use scripts to manage your files automatically).

About Coding Assignments

We encourage students to work together and exchange ideas. However, when it comes to finally sitting down to write your code, this must be done *independently*. Detecting software plagiarism is pretty much automated these days with systems such as MOSS.

https://www.quora.com/How-does-MOSS-Measure-Of-Software-Similarity-Stanford-detect-plagiarism

Please make sure your work is your own. If you are having trouble, the Faculty provides a free tutoring service to help you along. You can also contact the course instructor or the tutor during office hours. There are also numerous online resources – Google is your friend. The point isn't simply to get the assignment out of the way, but to actually learn something in doing.

fpf/pl Sept 2018