Department of Electrical and Computer Engineering
ECSE 202 – Introduction to Software Development
Assignment 2
Generalizing via Objects

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 ballistic ball with a simple physics model, you will leverage your existing code to allow adding multiple 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 trajectory 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 move 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

The goal of this assignment is to build a simulation for 100 balls with randomly chosen parameters. All balls are launched from the same X coordinate (center of the field) with initial height corresponding to ball radius. Figure 1 shows several snapshots from a video sequence generated by this simulation.

Figure 1 – a sequence of 3 frames from a video of the simulation sequence.

At minimum, you will need at least 2 classes for this simulation, the aBall class which generates an instance of ball in motion according to Assignment 1, and the bSim class which sets up the display, generates parameters and an instance of each ball, and runs until terminated by the user.

The first task is to design the aBall class according to the following constructor:

```
public aBall (double Xi, double Yi, double Vo, double theta,
              double bSize, Color bColor, double bLoss)
```

where

Xi: is the X coordinate of the initial launch position (meters)
Yi: is the Y coordinate of the initial launch position (meters)
Vo: is the initial launch velocity (meters/second)

theta: is the initial launch angle (degrees, as measured from the ground plane)
bSize: the radius of the ball (meters)
bColor: ball color
bLoss: energy loss coefficient

This class essentially encapsulates the code you wrote for Assignment 1.

The second task is to write the bSim class which randomly generates the parameters for 100 separate balls. The parameters used by this class are summarized below.

```
private static final int WIDTH = 1200;                  // n.b. screen coordinates
private static final int HEIGHT = 600;
private static final int OFFSET = 200;
private static final double SCALE = HEIGHT/100;         // pixels per meter
private static final int NUMBALLS = 100;                // # balls to simulate
private static final double MINSIZE = 1.0;              // Minumum ball radius (meters)
private static final double MAXSIZE = 10.0;             // Maximum ball radius (meters)
private static final double EMIN = 0.1;                 // Minimum loss coefficient
private static final double EMAX = 0.6;                 // Maximum loss coefficient
private static final double VoMIN = 40.0;               // Minimum velocity (meters/sec)
private static final double VoMAX = 50.0;               // Maximum velocity (meters/sec)
private static final double ThetaMIN = 80.0;            // Minimum launch angle (degrees)
private static final double ThetaMAX = 100.0;           // Maximum launch angle (degrees)
```

Designing the aBall class

The aBall class must do the following:

1. Create an instance of a GOval from the specified parameters (constructor). Further, since we need to add this object to the display, a corresponding get method needs to be included, i.e., getBall();.
2. Once the instance is generated, the simulation loop is run until the ball runs out of steam. Since messages are sent to the GOval inside of aBall, the ball will animate automatically until the simulation terminates. Most of the code you wrote in Assignment 1 can be copied with little change.
3. Since we want each ball to move independently, the corresponding objects must run *concurrently*. Although this is well beyond the scope of an introductory course, Java makes it very easy to make objects run concurrently. All that is required is for the aBall to extend the *thread* class.

You can find out more about the thread class here:
https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html).

The concept of threading will be very important for your later courses (especially Design Principles and Methods, ECSE 211) and allows your application to have multiple threads of execution running concurrently.

Design Approach

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

```
public class aBall extends Thread {

/**
* The constructor specifies the parameters for simulation.  They are
*
* @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 Vo  double The initial velocity of the ball at launch
* @param theta  double Launch angle (with the horizontal plane)
* @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
*/

public aBall(double Xi, double Yi, double Vo, double theta,
double bSize, Color bColor, double bLoss) {

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

/**
* The run method implements the simulation loop from Assignment 1.
* Once the start method is called on the aBall 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 **aBall** 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 velocity of 1.0 m/s. at an angle of 30°.

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

  aBall redBall = new aBall(10.0,100.0,1.0,30.0,6.0,Color.RED,0.25);
  add(redBall.getBall());
  redBall.start();
```

Details:

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

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

To make the resulting GOval accessable outside of aBall, we add an appropriate "getter" as follows:

```
public GOval getBall() {
  return myBall;
}
```

Since aBall 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 aBall class instance is to call the run method associated with aBall. 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 aBall instance runs in *parallel* with the main program and all other instances of the aBall 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();
}
```

For reasons that will become clear in later courses, the sleep duration for the aBall thread should be half as long as the time step. For example, if time is updated every 100 mS (0.1 S), then the sleep duration (above) should be half of that (50 mS).

One more detail regarding the aBall class – stopping. When the while loop is exited, the run method terminates which results in the thread terminating. For the while loop to continue running, two conditions must be met:

1. The total energy (KEx+KEy) must be greater than some minimum energy ETHR.
AND
2. The total energy must be less than it was on the previous bounce.

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:

```
public class bSim extends GraphicsProgram {

// Parameters used in this program

     public void run() {

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

Your main program should set up the display as in Assignment 1. 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 aBall constructor needs 7 parameters – Xi, Yi, Vo, theta, bSize, bColor, and bLoss. Use the RandomGenerator class shown in the slides to generate values for each aBall instance. For example, to generate a random loss parameter, one would use an instance of the RandomGenerator class as follows:

```
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 aBall 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, aBall.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.
2. Edit the parameters of bSim to replicate the output of Assignment 1, i.e., generate a single aBall instance with Xi=5m, Yi=1m, Vo=40 m/s, theta=85°, loss=0.4, radius=1m. Verify that it traces the identical path to the bounce class. Take a snapshot of the screen when the simulation reaches steady-state and save as A2-1.pdf.
3. Repeat Question 2, but this time with parameters Xi=95m, Yi=1m, Vo=40 m/s, theta=95°, loss=0.4, radius=1m. Verify that the path is identical to Question 2, except that the ball moves from right to left. Take a snapshot of the screen when the simulation reaches steady-state and save as A2-2.pdf.
4. Perform a simulation with 100 balls using the parameters listed on Page 2. In order for us to validate your simulation we need a means to ensure that all simulations produce identical results. After creating a the random number generator object do the following:

   ```
   rgen.setSeed((long) 0.12345);
   ```

   This will produce the same sequence of psuedo random numbers for each program run, terminating at exactly the same state. Take a snapshot of the screen when the simulation reaches steady-state and save as A2-3.pdf.

To Hand In:

1. The source java files. Note – use the default package.
2. Files A2-1, A2-2, and A2-3.

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, aBall.java, gUtil.java. We will build and test your code from here.

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 September 29, 2019

```
import java.awt.Color;

import acm.graphics.GOval;
import acm.graphics.GRect;
import acm.program.GraphicsProgram;
import acm.util.RandomGenerator;

/**
 * The main class in Assignment 2.
 * Here the canvas and ball objects are created and the resulting simulation
 * runs its course.
 *
 * @author ferrie
 *
 */

public class bSim extends GraphicsProgram{

/**
 * This is the main class which nominally uses the default constructor. To get around
 * problems with the acm package, the following code is explicitly used to make
 * the entry point unambiguous.  This is beyond the requirements for this assignment
 * and is provided for convenience.
 */

    public static void main(String[] args) {                        // Standalone Applet
        new bSim().start(args);
        }

 /**
  * There is no user I/O in this program.  Behavior is governed by the
  * parameters below.
  */

// Parameters used in this program

    private static final int WIDTH = 1200;                          // n.b. screen coordinates
    private static final int HEIGHT = 600;
```

```
    private static final int OFFSET = 200;
    private static final double SCALE = HEIGHT/100;                 // pixels per meter
    private static final int NUMBALLS = 1;                        // # balls to simulate
    private static final double MINSIZE = 1.0;                      // Minumum ball radius
    private static final double MAXSIZE = 10.0;                     // Maximum ball radius
    private static final double EMIN = 0.1;                         // Minimum loss coefficient
    private static final double EMAX = 0.6;                         // Maximum loss coefficient
    private static final double VoMIN = 40.0;                       // Minimum velocity
    private static final double VoMAX = 50.0;                       // Maximum velocity
    private static final double ThetaMIN = 80.0;                    // Minimum launch angle
    private static final double ThetaMAX = 100.0;                   // Maximum launch angle

    /**
     * The run method is the entry point for this program.
     */

    public void run() {
        this.resize(WIDTH,HEIGHT+OFFSET);           // optional, initialize window size

// Create the ground plane

    GRect gPlane = new GRect(0,HEIGHT,WIDTH,3);
    gPlane.setColor(Color.BLACK);
    gPlane.setFilled(true);
    add(gPlane);

// Set up random number generator

    RandomGenerator rgen = RandomGenerator.getInstance();
    rgen.setSeed((long) 0.12345);

// Generate a series of random bBalls and let the simulation run till completion

        for (int i=0; i<NUMBALLS; i++) {
            double iSize = rgen.nextDouble(MINSIZE,MAXSIZE);    // Current size
            double Xi = WIDTH/(2*SCALE);                        // Launch X is always center of screen
            double Yi = iSize;                                  // Launch Y is current ball radius.
            Color iColor = rgen.nextColor();                    // Current color
```

```
            double iLoss = rgen.nextDouble(EMIN,EMAX);          // Current loss coefficient
            double iVel = rgen.nextDouble(VoMIN,VoMAX);         // Current velocity
            double iTheta = rgen.nextDouble(ThetaMIN,ThetaMAX); // Current launch angle
            aBall iBall = new aBall(95.0,1.0,40.0,95.0,1.0,Color.RED,0.40,this);
//          aBall iBall = new aBall(5.0,1.0,40.0,85.0,1.0,Color.RED,0.40,this);
//          aBall iBall = new aBall(Xi,Yi,iVel,iTheta,iSize,iColor,iLoss);      // Generate instance
            add(iBall.getBall());                                               // Add to display list
            iBall.start();                                                      // Start this instance
        }
    }
}
```

```java
import java.awt.Color;

import acm.graphics.GOval;

/**
 * This class provides a single instance of a ball on a ballistic trajectory
 * subject to air resistance (aBall).
 *
 * 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 aBall extends Thread {

    /**
     * The constructor specifies the parameters for simulation.  They are
     *
     * @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 Vo        double The initial velocity of the ball at launch
     * @param theta     double Launch angle (with the horizontal plane)
     * @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
     */

    public aBall(double Xi, double Yi, double Vo, double theta, double bSize, Color bColor, double bLoss)
     {

        this.Xi = Xi;                                       // Get simulation parameters
        this.Yi = Yi;
        this.Vo = Vo;
        this.theta = theta;
        this.bSize = bSize;
```

```
    this.bColor = bColor;
    this.bLoss = bLoss;

    // Create instance of ball using specified parameters

    myBall = new GOval(gUtil.XtoScreen(Xi),gUtil.YtoScreen(Yi),
                       gUtil.LtoScreen(2*bSize),gUtil.LtoScreen(2*bSize));
    myBall.setFilled(true);
    myBall.setFillColor(bColor);

}

/**
 * Optional: this constructor adds an additional argument which provides a
 *           a link back to bSim.  This allows access to any of the methods
 *           accessible to bSim.
 */

public aBall(double Xi, double Yi, double Vo, double theta, double bSize, Color bColor, double bLoss,
 bSim link) {

    this.Xi = Xi;                                       // Get simulation parameters
    this.Yi = Yi;
    this.Vo = Vo;
    this.theta = theta;
    this.bSize = bSize;
    this.bColor = bColor;
    this.bLoss = bLoss;
    this.link = link;                                   // Link to caller

    // Create instance of ball using specified parameters

    myBall = new GOval(gUtil.XtoScreen(Xi),gUtil.YtoScreen(Yi),
                       gUtil.LtoScreen(2*bSize),gUtil.LtoScreen(2*bSize));
    myBall.setFilled(true);
    myBall.setFillColor(bColor);

}
```

```
/**
 * 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() {

    // Main animation loop - do this until program halted by closing window
    // Units are MKS with mass = 1.0 Kg

    double Vt = g / (4*Pi*bSize*bSize*k);   // Terminal velocity
    double time = 0;                        // time (reset at each interval)
    double KEx=ETHR,KEy=ETHR;               // Kinetic energy in X and Y directions
    double X,Xo,Xlast,Vx,Y,Vy,Ylast,Elast;  // Position and velocity variables as defined above

    double signVox = 1;                     // Carries the sign of Vox.
    double Vox=Vo*Math.cos(theta*Pi/180);   // Initial velocity components in X and Y
    double Voy=Vo*Math.sin(theta*Pi/180);
    if (Vox < 0) signVox = -1;

    Xo=Xi;                                  // Initial X position
    Y=Yi;                                   // Initial Y position
    Ylast=Y;                                // Y position at end of previous iteration (use this
                                            // to estimate Y velocity).
    Xlast=Xo;                               // Same for X.
    Elast=0.5*Vo*Vo;

    boolean hasEnoughEnergy = true;         // loop control variable

    // Simulation loop - compute position and velocity using Newtonian mechanics

        while(hasEnoughEnergy) {
```

```
            X = Vox*Vt/g*(1-Math.exp(-g*time/Vt));              // Update position
            Y = bSize + Vt/g*(Voy+Vt)*(1-Math.exp(-g*time/Vt))-Vt*time;

            Vx = (X-Xlast)/TICK;                                // Estimate X and Y velocities
            Vy = (Y-Ylast)/TICK;

            Xlast = X;                                          // For next iteration
            Ylast = Y;


        // Check to see if we've hit the ground.  If yes, inject energy loss,
        // force current value of Y to ball radius, restart Yi for next
        // interation.

            if ((Vy<0)&&(Y<=bSize)) {

                KEx = 0.5*Vx*Vx*(1-bLoss);              // Kinetic energy in X direction after
                 collision
                KEy = 0.5*Vy*Vy*(1-bLoss);              // Kinetic energy in Y direction after
                 collision
                Vox = Math.sqrt(2*KEx)*signVox;         // Resulting horizontal velocity
                Voy = Math.sqrt(2*KEy);                 // Resulting vertical velocity
        //
        // If the energy in the system after collision is less
        // than threshold ETHR, terminate the simulation.
        //

                if (KEx+KEy < ETHR || KEx+KEy >= Elast)
                    hasEnoughEnergy = false;
                else
                    Elast=KEx+KEy;

//                  System.out.println("E= "+(KEx+KEy)+"  KEx= "+KEx+"  KEy= "+KEy);
//                  System.out.printf("t: %.2f  X: %.2f  Y: %.2f  Vx: %.2f  Vy: %.2f\n",time,X,Y,Vx,Vy);


        // Otherwise reset for next iteration and continue.
```

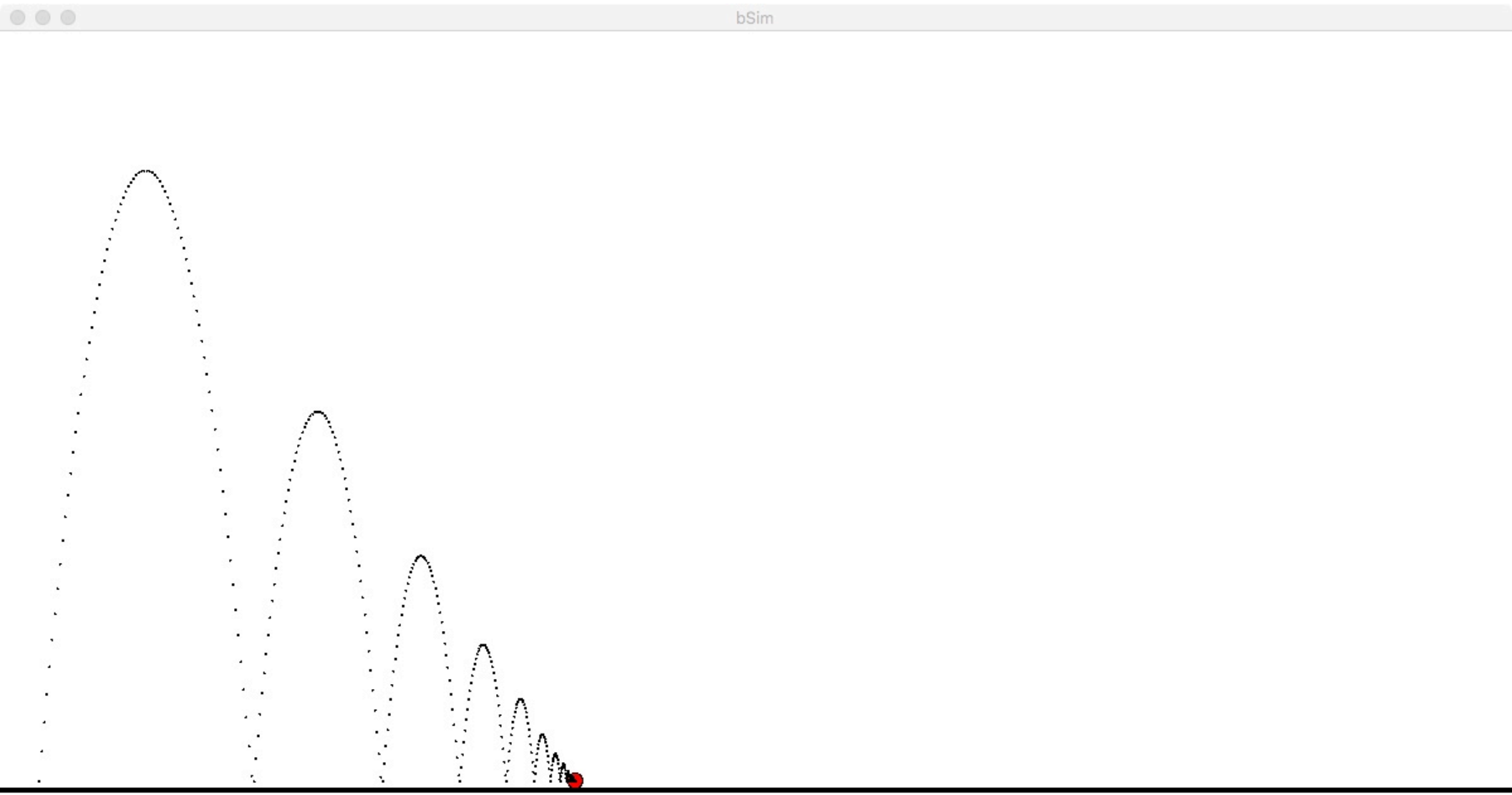
```
                time=0;                                     // Reset current interval time
                Y=bSize;                                    // Physically limit ball penetration
                Xo+=X;                                      // Add to the cumulative X displacement
                X=0;                                        // Reset X to zero for start of next interval
                Xlast=X;                                    // Reset last X and Y positions
                Ylast=Y;
            }

    // Update ball position on screen

            double ScrX = gUtil.XtoScreen(Xo+X-bSize);
            double ScrY = gUtil.YtoScreen(Y+bSize);
            myBall.setLocation(ScrX,ScrY);      // Screen units

    // If a link has been provided, plot the corresponding trace points

            if (link != null) {
                trace(Xo+X,Y);
            }

    // Delay and update clocks

            try {
                Thread.sleep((long) (TICKmS/2));
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

            time+=TICK;

        }
}

/**
 * Access methods for instance variables
 */
```

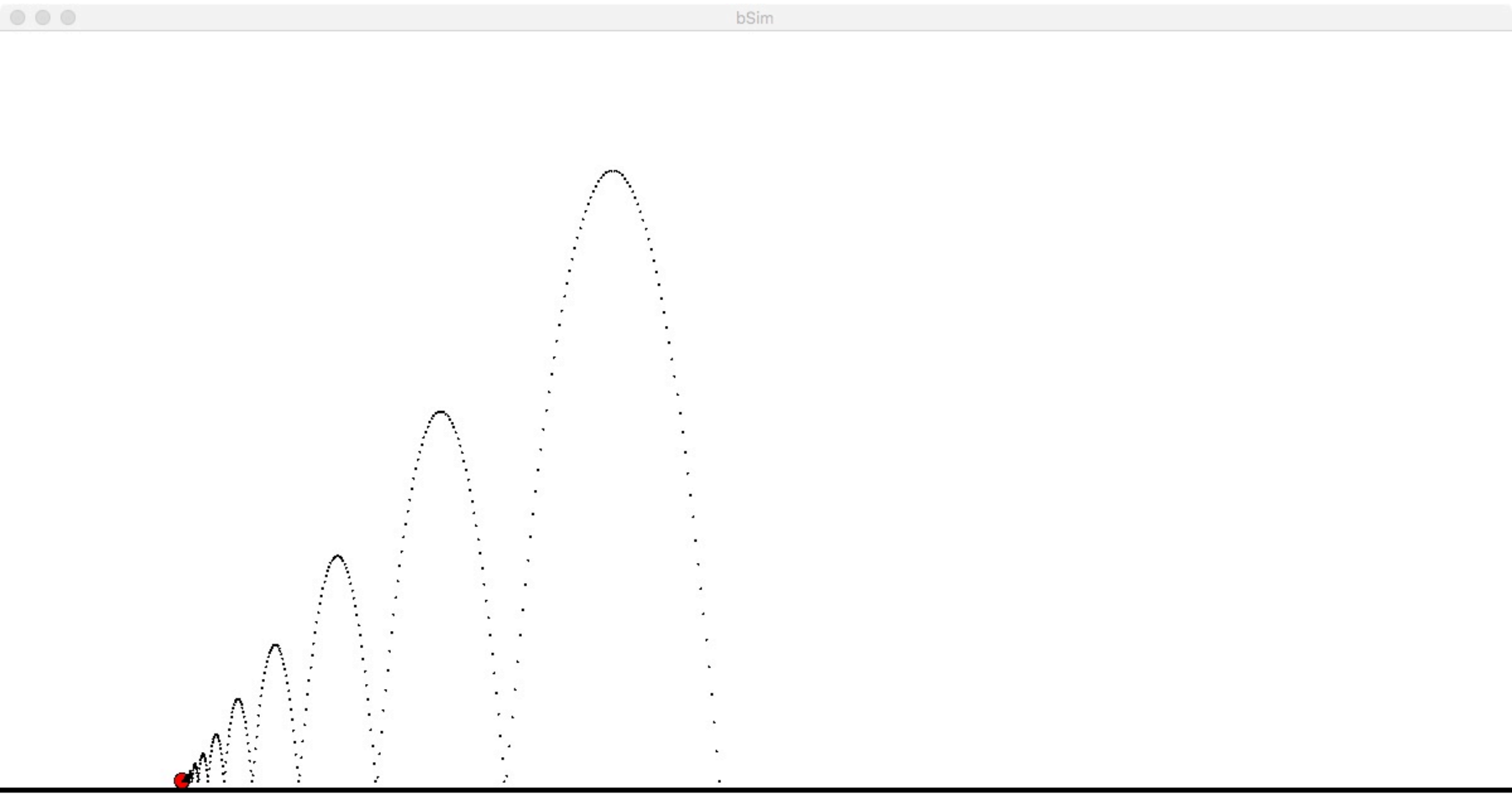
```
public GOval getBall () {
    return myBall;
}

/**
 * Trace method from Assignment 1
 */

private void trace(double x, double y) {
    double ScrX = x*SCALE;
    double ScrY = HEIGHT - y*SCALE;
    GOval pt = new GOval(ScrX,ScrY,PD,PD);
    pt.setColor(Color.BLACK);
    pt.setFilled(true);
    link.add(pt);
}


/**
 * Instance Variables & Class Parameters
 */

// Instance Variables

private GOval myBall;
private double Xi;
private double Yi;
private double Vo;
private double theta;
private double bSize;
private Color bColor;
private double bLoss;
private bSim link;

// Program constants

private static final double Pi=3.141592654;     // Pi
private static final double g=9.8;              // Gravitational acceleration
```

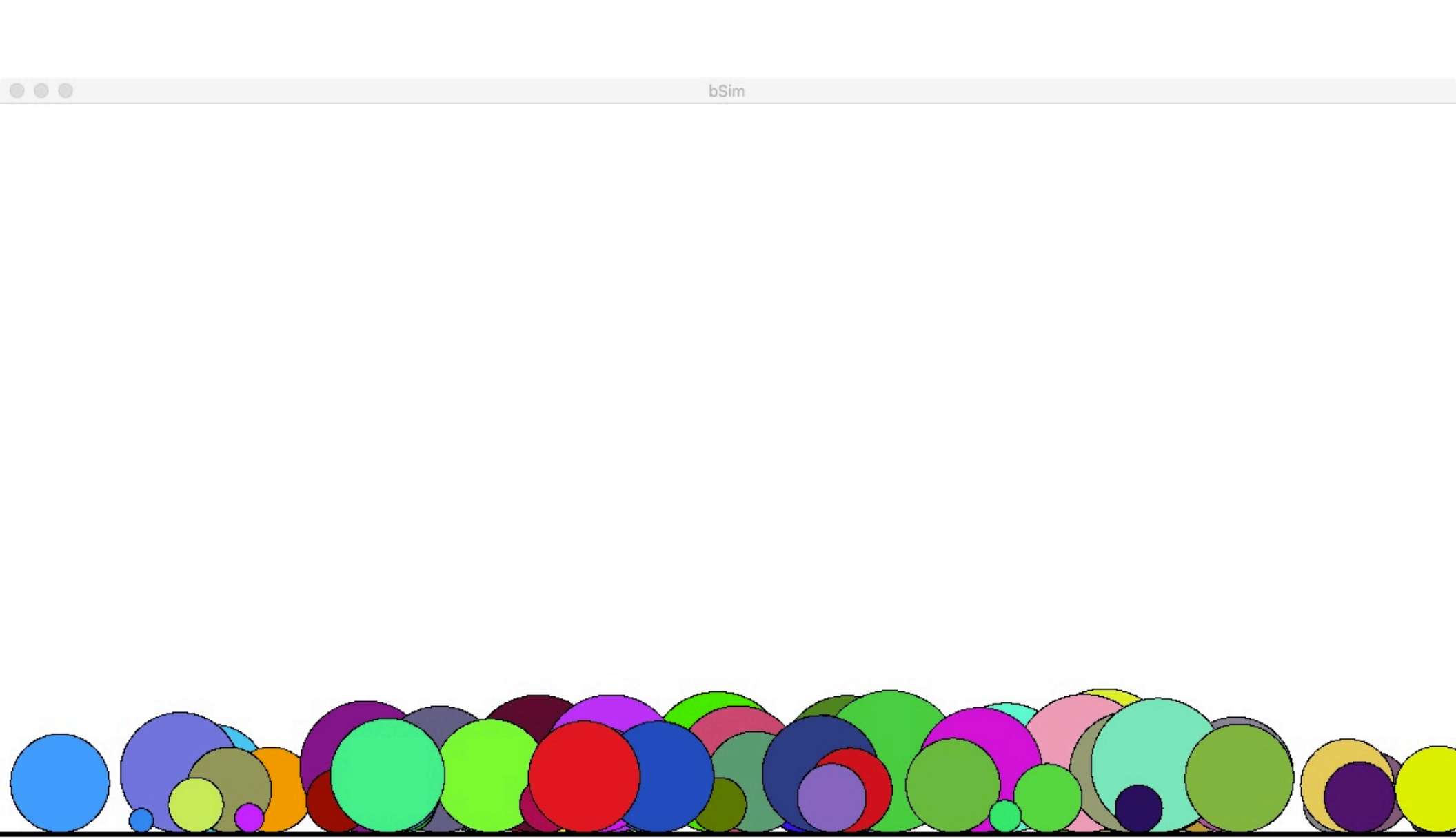
```
    private static final double TICK = 0.1;          // Clock tick duration (seconds)
    private static final double TICKmS = TICK*1000; // Clock tick duration (milliseconds)
    private static final double ETHR = 0.01;         // If KEx+KEy < ETHR stop
    private static final double k = 0.0001;          // Vt constant

    // Need these parameters to implement trace point plotting.

    private static final int WIDTH = 1200;           // n.b. screen coordinates
    private static final int HEIGHT = 600;
    private static final int OFFSET = 200;
    private static final double SCALE = HEIGHT/100; // pixels per meter
    private static final double PD = 1;              // Trace point diameter


}
```

```java
/**
 * Some helper methods to translate simulation coordinates to screen
 * coordinates
 * @author ferrie
 *
 */
public class gUtil {

    private static final int WIDTH = 600;                           // n.b. screen coordinates
    private static final int HEIGHT = 600;
    private static final int OFFSET = 200;
    private static final double SCALE = HEIGHT/100;                 // Pixels/meter

    /**
     * X coordinate to screen x
     * @param X
     * @return x screen coordinate - integer
     */

    static double XtoScreen(double X) {
        return X * SCALE;
    }

   /**
    * Y coordinate to screen y
    * @param Y
    * @return y screen coordinate - integer
    */

    static double YtoScreen(double Y) {
        return HEIGHT - Y * SCALE;
    }

    /**
     * Length to screen length
     * @param length - double
     * @return sLen - integer
     */
```

```
    static double LtoScreen(double length) {
        return length * SCALE;
    }

    /**
     * Delay for <int> milliseconds
     * @param int time
     * @return void
     */

    static void delay (long time) {
        long start = System.currentTimeMillis();
        while (true) {
            long current = System.currentTimeMillis();
            long delta = current - start;
            if (delta >= time) break;
        }
     }
}
```


bSim


bSim


bSim