Department of Electrical and Computer Engineering
ECSE 202 – Introduction to Software Development
Assignment 1 – Fall 2019
Introduction to Programming in Java

Introduction

This assignment is based on Chapter 2 of the textbook by Eric Roberts, "Programming by Example". Even if you have never written a computer program before, it is possible to write useful programs by learning a few basic "patterns" as discussed in the chapter. In preparation for this assignment, make sure that you've read and understand the examples in Chapter 2.
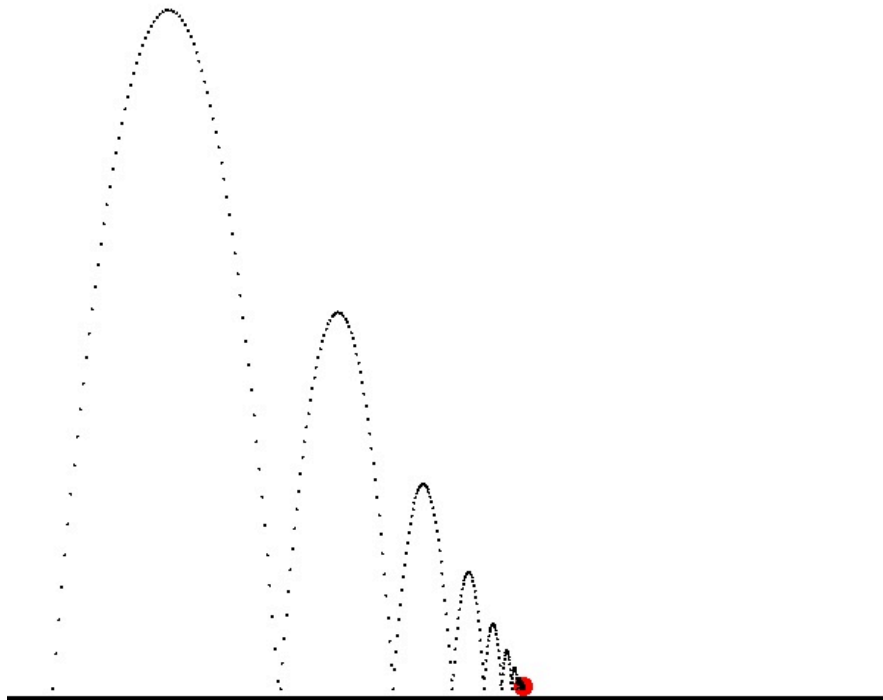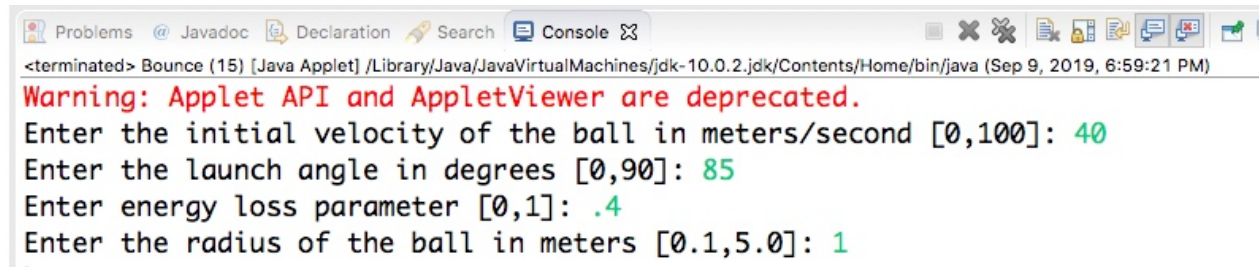
Problem Description

Figure 1: Simulation of projectile motion with bouncing..

Figure 1 shows a trace of the trajectory of a 1 Kg ball undergoing projectile motion (PM). It is launched with an initial velocity of 40 m/s, at an angle of 85°. As it moves through space, it loses energy due to friction with air, and on each bounce it loses 40% of its kinetic energy. In this assignment you will write a program to generate this simulation using the equations of motion from your basic mechanics course.

The program begins by prompting the user for 4 parameters as follows:

Figure 2: Reading simulation parameters from the console

The first parameter, *Vo*, represents the initial velocity of the ball in meters/second, followed by the launch angle, *theta* in degrees. Collision loss is specified by the energy loss parameter, *loss*, expressed as a number in the range [0,1]. Finally, the radius of the ball, *bSize*, in meters is used to account for friction with the air. The Add2Integers program example in the slides shows you how to read parameters interactively. *Vo* and *theta* are self-explanatory in light of the PM equations described below, as is the radius *bSize*. The loss parameter needs a bit more of an explanation. Just before the ball collides with the ground, its total energy is approximately ½ M V^2, i.e., kinetic energy. The loss parameter specifies what fraction of this energy is lost in collision, e.g., 0.4 means a loss of 40%.

Simulation Algorithm

A psuedocode description of the algorithm you are to implement is as follows:

1. Read parameters from user.
2. Draw a ground plane on the screen and place ball at initial position.
3. Initialize simulation parameters.
4. Calculate X, Y, Vx, Vy for current value of t.
5. Determine corresponding position of ball in screen coordinates and move it there.
6. If Vy is negative and Y <= radius of ball, calculate kinetic energy, decrease it by the loss factor, recalculate V.
7. If kinetic energy < threshold, terminate.
8. Go to Step 4

Projectile Motion Equations:

The position and velocity of a mass undergoing projectile motion is summarized by the following equations:

$$V_t = \frac{mg}{4\pi\, k\, bSize^2}$$

$$X = \frac{V_t V_o \cos(\vartheta)}{g}(1 - e^{\frac{-gt}{V_t}})$$

$$Y = \frac{V_t}{g}(V_o \sin(\vartheta) + V_t)(1 - e^{\frac{-gt}{V_t}}) - V_t t$$

$$V_x = \frac{X(t) - X(t - \Delta t)}{\Delta t}$$

$$V_y = \frac{Y(t) - Y(t - \Delta t)}{\Delta t}$$

V_t corresponds to terminal velocity, the point at which the force due to air resistance balances gravity. At this point the ball falls with constant velocity. It is dependent on surface area, $4\pi\, bSize^2$, and the parameter, k. For this assignment, assume m=1.0 kg and k=0.0016

Expressions for X and Y take into account loss due to air resistance through the V_t term.

Approximate velocities in X and Y directions by taking the discrete derivative of X and Y respectively. The Δt term is the delay between adjacent samples. Assume $\Delta t = 0.1$ second.

The corresponding Java code looks something like this:

```
double Vt = g / (4*Pi*bSize*bSize*k);                    // Terminal velocity

double Vox=Vo*Math.cos(theta*Pi/180);                    // X component of initial velocity

double Voy=Vo*Math.sin(theta*Pi/180);                    // Y component of initial velocity

X = Vox*Vt/g*(1-Math.exp(-g*time/Vt));                        // X position

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

(since the ball position is determined at the center, the lowest value of Y is the radius of the ball.)

```
Vx = (X-Xlast)/TICK;                        // Estimate Vx from difference

Vy = (Y-Ylast)/TICK;                        // Estimate Vy from difference
```

Detecting Collision with Ground

A collision is detected when:
  1. Vy is negative (ball falling towards ground) and
  2. Y is <= the radius of the ball

A simple strategy when this happens is to recompute V_o assuming some energy loss through collision. Assuming that the total energy of the ball before collision is in kinetic energy, then:

```
KEx = 0.5*Vx*Vx*(1-loss);          // Kinetic energy in X direction after collision
KEy = 0.5*Vy*Vy*(1-loss);          // Kinetic energy in Y direction after collision

so

Vox = Math.sqrt(2*KEx);            // Resulting horizontal velocity
Voy = Math.sqrt(2*KEy);            // Resulting vertical velocity
```

We can then restart the simulation of the next parabolic arc by setting X to 0 and Y to the lowest point on the trajectory, the ball radius. Note: variable X only describes displcement relative to the current starting point. If we simply used this value in the plot, each parabola would overlap. To get around this, we sum the displacements at the conclusion of each parabola in variable Xo (which stands for offset). So the actual position in simulation coordinates is (Xo+X,Y). Another detail that needs to be taken care of is that upon computing X, Y, Vx, and Vy, you will need to record the values of X and Y to be used in the following iteration, i.e., in variables Xlast and Ylast.

Writing the Program:

Chapter 2 provides several examples (i.e. code) for displaying simple graphical objects in a display window. Essentially you create a class that extends the acm class, GraphicsProgram, and provide a run() method (i.e. your code). There are a couple of items not described in Chapter 2 that will be useful here.

Parameters: Your simulation program requires two kinds of inputs, variables such as V_o, read in from the console input, and parameters such as *WIDTH* and *HEIGHT* that define fixed values used by your program. To help simplify the design of your first program, here the list of parameters used by the program we wrote to implement the simulation. The first set defines parameters related to the display screen.

private static final int WIDTH = 600;   // defines the width of the screen in pixels
private static final int HEIGHT = 600;   // distance from top of screen to ground plane
private static final int OFFSET = 200;   // distance from bottom of screen to ground plane

The remaining parameters define parameters related to the simulation and are expressed in simulation (not screen) coordinates.

private static final double g = 9.8;   // *MKS gravitational constant 9.8 m/s^2*
private static final double Pi = 3.141592654;   // To convert degrees to radians
private static final double Xinit = 5.0;   // *Initial ball location (X)*
private static final double Yinit = bSize;   // *Initial ball location (Y)*
private static final double TICK = 0.1;   // *Clock tick duration (sec)*
private static final double ETHR = 0.01;   // If either Vx or Vy < ETHR STOP
private static final double XMAX = 100.0;   // Maximum value of X
private static final double YMAX = 100.0;   // Maximum value of Y
private static final double PD = 1;   // Trace point diameter

private static final double SCALE = HEIGHT/XMAX;   // Pixels/meter

Display:

When you create an instance of a graphics program, the display "canvas" is automatically created using default parameters. To create a window of a specific size, use the `resize` method as shown below:

```
public void run() {
      this.resize(WIDTH,HEIGHT);
```

where `WIDTH` and `HEIGHT` are the corresponding dimensions of the display canvas in pixels. Since we are doing a simulation of a physical system, it would be convenient to set up a coordinate system on the screen that mimics the layout of the simulation environment. This is shown schematically in Figure 3 below.
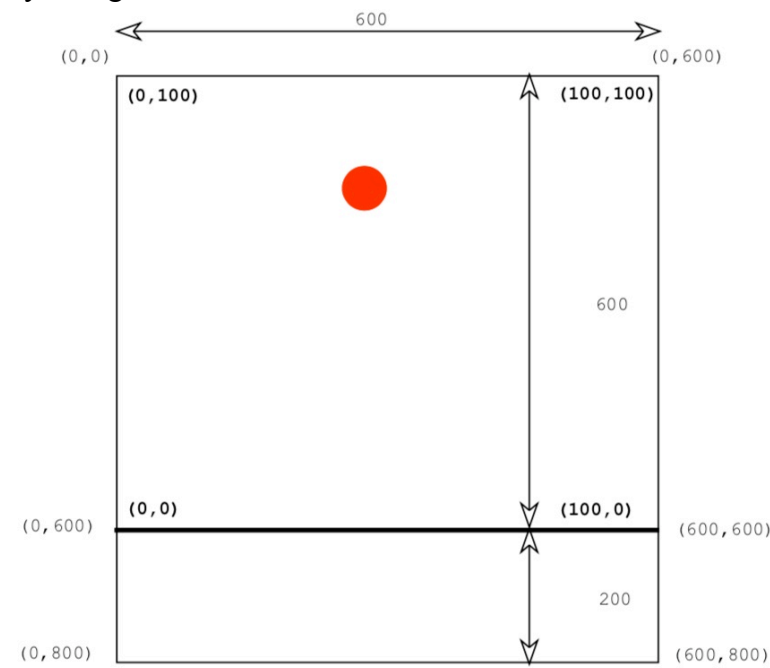

Figure 3

In the example shown in Figure 3, WIDTH=600 and HEIGHT=600+200. Let (x,y) represent a point in pixel coordinates and (X,Y) the corresponding point in simulation coordinates.   The ground plane is represented by a filled rectangle beginning at (0,600) with width=600 and height=3 pixels. The upper left corner of this rectangle (0,600) corresponds to (0,0) in simulation coordinates.   It is easy to infer the following relations between pixel and simulation coordinates: $x = X * 6$, $y = 600 – Y * 6$. The number 6 is the *scale factor* between pixel and simulation units. Since the y axis is in a direction opposite to the Y axis, we need to invert the scale by subtracting the scaled Y value from HEIGHT less the offset of the ground plane, 200. Generally we use parameters to define these equations as follows: $x = X * SCALE$, $y = HEIGHT – Y * SCALE$.

Getting input from the user:

If you've programmed in Java before, you know how to obtain input from the user using the Scanner class. If this is your first time, then it is suggested that you include the acm classes in your program. How to do so will be explained in the tutorial sessions (you could also consult the course text). Chapter 2 shows how to read integer values using the readInt method; readDouble is the analogous method for real numbers, and has a similar form: double value = readDouble("Enter value: ");

Initialization:

Good software design practice usually entails thinking about the representation for a problem before any code is written. Each of the variables within the Simulation Loop needs to be explicitly represented as a Java datatype, in this case double or int. Consider, for example, the X component of the initial projectile velocity, V_o. In a Java program (class to be more specific), V_o can be declared and initialized in a single line of code:

```
double Vox=Vo*Math.cos(theta*Pi/180);
```

Before the simulation begins, each variable should be initialized to a known state. Pay attention to order. The above expression cannot be evaluated until V_o is read from the user input.

Program structure – the simulation loop.

The template for the simulation program has the following form:

```
Public class Bounce extends GraphicsProgram {

    private static final int WIDTH = 600;
    .
    .

    public void run() {
         this.resize(WIDTH,HEIGHT);

    // Code to set up the Display shown in Figure 2.
    // (follow the examples in Chapter 2)
    .
    .
    .

    // Code to read simulation parameters from user.

    double Vo = readDouble ("Enter the initial velocity of
    the ball in meters/second [0,100]: ");.
    .
    .
```

```
      // Initialize variables

      double Vox=Vo*Math.cos(theta*Pi/180);.
      .
      .

      // Simulation loop

      while(true) {
        X = Vox*Vt/g*(1-Math.exp(-g*time/Vt));
        Y = bSize + Vt/g*(Voy+Vt)*(1-Math.exp(-g*time/Vt))-
            Vt*time;
        Vx = (X-Xlast)/TICK;
        Vy = (Y-Ylast)/TICK;
      .
      .

      // Display update

      ScrX = (int) ((X-bSize)*SCALE);
      ScrY = (int) (HEIGHT-(Y+bSize)*SCALE);
      myBall.setLocation(ScrX,ScrY);        // Screen units
      .
      .
      }
   }
```

You can find most of what you need to code this assignment in the examples in Chapter 2. The only detail that has not been covered at this point is the `while` loop. As the name implies, code within the loop is executed repeatedly until the looping condition is no longer satisfied or a `break` statement is executed within the loop. The general form of a while loop is

```
   while (logical condition) {
        <code>
   }
```

In the example above, the logical condition is set to true, which means the loop executes until the program is terminated or a `break` executed as shown below:

```
   while (true) {
        <code>
        if ((KEx <= ETHR) | (Key <= ETHR)) break;

   }
```

In the above example, the simulation will run until the remaining energy in either the vertical or horizontal direction falls to a fraction of the initial input.

Instructions

The easiest way to implement this program is to write it in stages, testing each stage to make sure that it produces the correct output. Further, your program should include the following lines of code so that program variables can be traced at each time step:

add an additional parameter:
      private static final boolean TEST = true;   // print if test true

add the following statement in your simulation loop:
      if (TEST)
      System.out.printf("t: %.2f X: %.2f Y: %.2f Vx: %.2f Vy:%.2f\n",
                          time,Xo+X,Y,Vx,Vy);

When TEST is true, your program will output a table of values so that the program can be validated.

1. Write an initial version of Java class, Bounce.java, that implements a single parabolic trajectory – from launch to first contact with the ground. It is easier to implement this code first to make sure that your equations are implemented correctly. Save your plot (e.g. using screen capture) to a file named A1-1.pdf and your numerical output to a file named A1-1.txt. Use the following inputs: Initial velocity 40, launch angle 85, energy loss 0.4, ball radius 1.
2. Add in remaining code to handle contact bounce. If the kinetic energy is less than parameter ETHR, terminate the simulation. Otherwise initiate the following arc. With printing enabled, run a simulation with the same inputs as in Part 1. Save your plot to a file named A1-2.pdf and the numerical output to file A1-2.txt.
3. Verify that your code correctly accounts for air resistance by setting energy loss to 0 (you should see a similar decaying pattern, but over a longer interval. To speed things up, we will increase ball radius to incur greater air resistance and reduce the initial velocity. Use the following inputs: Initial velocity 25, launch angle 85, energy loss 0, ball radius 1.2. Disable printing for this run. Save your plot to file A1-3.pdf.

To Hand In:

1. The source file, Bounce.java (the final version).
2. Files A1-1.pdf, A1-2.pdf, A1-3.pdf, A1-1.txt and A1-2.txt/

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

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 (especially if one is copying from previous assignments for this course).

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

We know that assignments from previous sessions are available on the Web. If any of this code is found in your submission, it will not be graded (grade = 0) and subject to Faculty disciplinary procedures.

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/Sep 11, 2019