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

Introduction

In this Java assignment you will calculate the trajectory of a bouncing ball using a simplified physics model. For this Newtonian simulation, you will begin by writing a program that can output the x and y positions of the ball at sequnetial increments in time. You will then add to the program to visually animate the trajectory of the ball in a graphics window.
This assignment is based on Chapter 2, "Programming by Example", of *The Art and Science of Java* textbook by Eric Roberts. 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. You will need concepts from Chapters 3 and 4 as well.

Problem Description

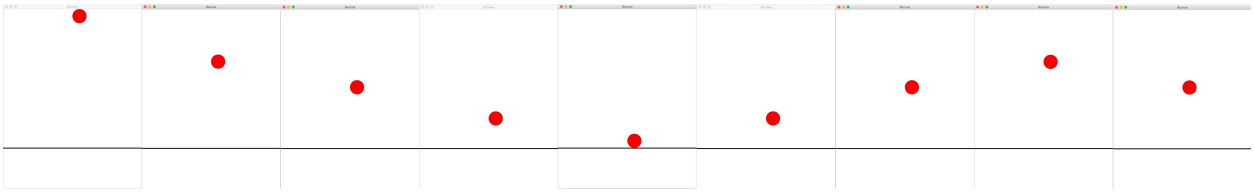


Figure 1: Snapshots from simulation of ball drop

Figure 1 shows a sequence of screen shots of the simulation of a 1 Kg ball dropped from a height of 10 meters, with the ball constrained to move in a vertical direction only. At each time step, the height of the ball is calculated and the position changed on the display. This can be done using simple Newtonian mechanics as follows:

If the ball starts off at height h_0 above the ground, then at each time instant, t, $h(t) = h_0 - 0.5\ g\ t^2$, where g is the gravitational constant, $9.8\ m/s^2$. Just before the ball collides with the ground, it attains a terminal (impact) velocity, v_t, which can be determined from conservation of energy, i.e., $0.5\ m\ v^2 = m\ g\ h_0$, so $v_t = (2\ g\ h_0)^{0.5}$. Assuming no loss of energy, height in the upward direction is given by $h(t) = v_t\ t - 0.5\ g\ t^2$. In practice, some fraction of the kinetic energy is lost, so v_t decreases over time. Assume that l represents the fraction of energy lost in the collision. Then $v_t = (2\ e_l\ g\ h_0)^{0.5}$, where $e_l = 1 - l$.

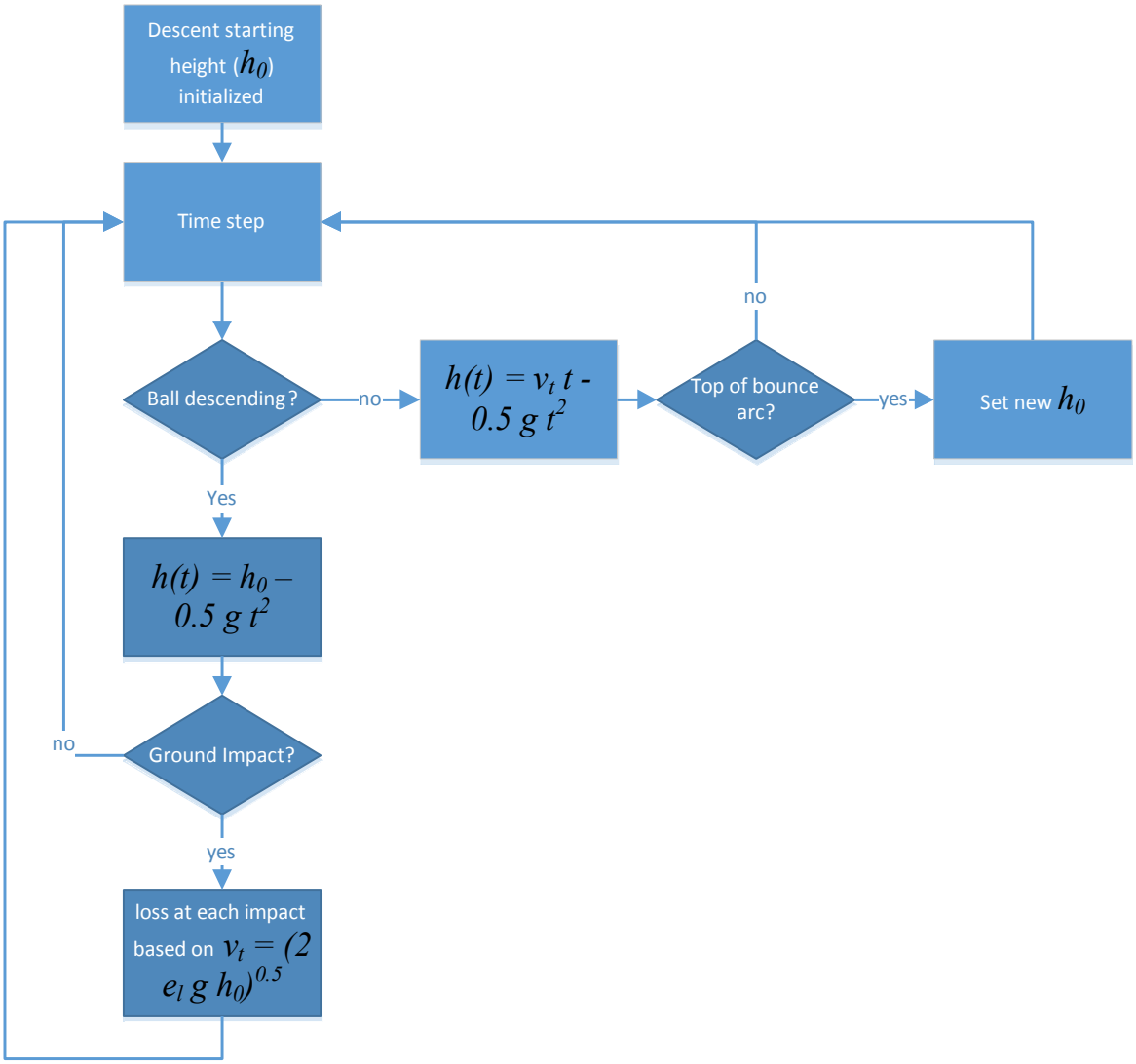


Figure 2: Basic algorithm for simplified bouncing ball

You program will need to implement this basic algorithm in Figure 2 including the code to figure out the various decision points.

So, where does one begin? One can break the program down into the following steps:

1. Create a display window with an instance of a ball and a line representing the ground as shown in Figure 1.
2. Read simulation parameters from the user – the initial height, h_0, and the energy loss, *l*, represented as a number in the range [0,1].
3. Initialize the simulation: set h_0, *time = 0; directionUp= false* (down); $v_t = (2\, g\, h_0)^{0.5}$ (the impact velocity is based on the starting height and can be calculated at the beginning); $e_l SqRt = (1 - l)^{0.5}$ (as can the energy loss factor)
4. Pseudocode for decisions in simulation loop:

   If direction is down then {

```
        h(t) = h0 – 0.5 g t^2
        If h(t) <=0 {           //ground impact
            h0 = h(t)           // new h0, also be used to determine direction
            initialUpPosition = h(t)        //should be close to 0
            directionUp = true
            time=0              //resetting time for upward arc
            vt = vt* e_lSqRt            //new impact energy loss
        }
}
else {                          //direction is up
        h(t) = initialUpPosition + v_t t - 0.5 g t^2
        if h(t) > h0 {          //still going up
            h0 = h(t)           //keeping track of last highest point
        }
        else {                  //starting to go down
            directionUp = false
            time = 0            //resetting time for going down
        }
}
```

Note that the steps above will be within in a larger loop that will be incrementing time. Also note that pseudocode does not necessarily follow the proper rules and syntax of a programming language (like Java) – so you cannot copy and paste it into a development environment like Eclipse. Pseudo code will often be a mixture of plain language and borrowed terms from a programming language, in order to better delineate the steps in a proposed algorithm like in Figure 2. You need to then translate the pseudocode into proper programming syntax. Let's look at the steps above in more detail:

To make things simpler, it is best to think of the ball *y* origin, *h(t)*, as the bottom of the ball (rather than e.g. the centre of the ball); this will make determining the ground impact point (i.e. when *h(t)=0*) clearer in the code. However, you will need to take this into account when determining how to display the ball based on these coordinates (discussed below).

As outlined in Figure 2, there are two separate blocks depending on whether the ball is going down or up, and two corresponding equations to determine the ball position *h(t)*.

Within the downward block, once the *h(t)* position is calculated, a decision is made on whether we have hit the ground, and if so, a set of variables are set correspondingly. Note that in an ideal world, the *initialUpPosition* (on impact) would be 0 (as assumed in Figure 2). However, since we will be incrementing time in discrete steps (described below), when we calculate *h(t)* upon "hitting" the ground, we might end up with a slightly negative value as we've "passed through" the ground in that time step. To keep the simulation more accurate, we can keep track of this value and use it as the position in the subsequent upward equation. Conversely, you could just assume the initial position when first heading up is 0, and have a slight accumulated error in the subsequent values of *h(t)*.

The upward block has a decision point determining whether the ball is still going up, and sets relevant variables according to this decision.

1. Setting up the program:

Chapter 2 provides several examples (i.e. code) for displaying simple graphical objects in a display window, as well as inputting user-values from a console and printing out calculated values in that console. 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: It is useful to define parameters as shown below. It not only makes code easier to read, but it also makes changing program behavior less prone to error.

```
private static final int WIDTH = 600; //pixels
private static final double G = 9.8;  // m/s^2
private static final double TIME_OUT = 30; //seconds
private static final double INTERVAL_TIME = .1; //seconds
```

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 on the following page.

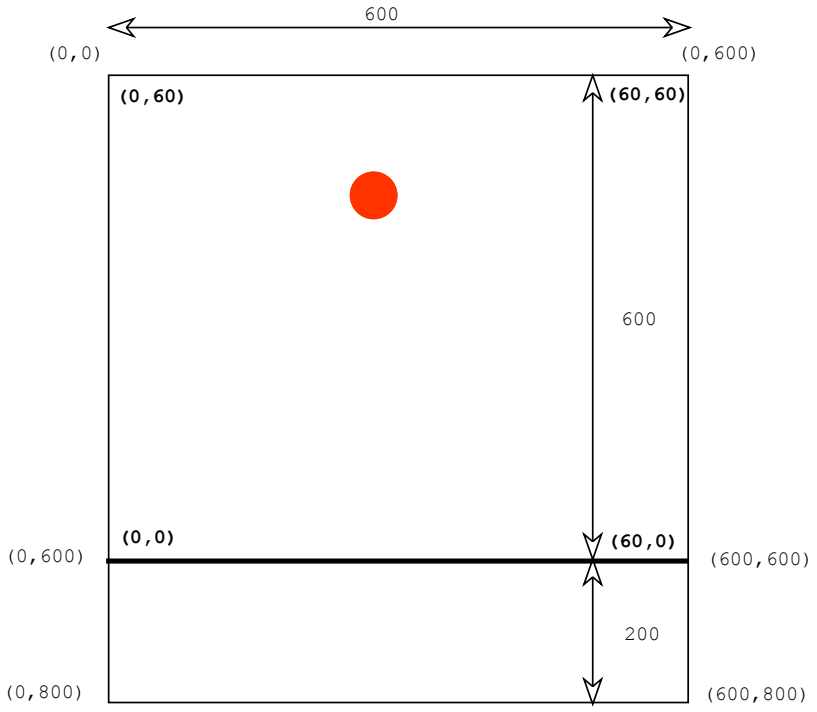


Figure 3

In the example shown in Figure 2, 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: $x = X * 10, \quad y = 600 - Y * 10$. The number 10 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. Keep this transform in mind when determining how to display the ball of a set diameter, with its *h(t)* local origin at the bottom.

2. 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). In this program you will need to input the initial drop height and energy loss parameters, both of which are represented as type `double` (computer approximation of real numbers using floating point representation). 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: ");`

3. 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, terminal (impact) velocity *vt.* In a Java program (class to be more specific), *vt* can be declared and initialized in a single line of code:

```
double vt = Math.sqrt(2*g*h0);
```

Note that you can use methods such as Math.srt() or Math.pow() for this assignment. Before the simulation begins, each variable should be initialized to a known state. Pay attention to order. The above expression cannot be evaluated until *h0* is read from the user input.

4. Program structure – the simulation loop.

The template for the simulation program has the following form where you need to fill in missing parts:

```
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 h0 = readDouble ("Enter height the height of
     the ball in meters [0,60]: ");
     .
     .

     // Initialize variables

     double vt = Math.sqrt(2*G*h0);
     .
     .

     // Simulation loop
```

```
while (totalTime < TIME_OUT) {
     if (!directionUp) {
          height = h0 - 0.5*G* Math.pow(time, 2);
          if (height <= 0) {
.
.
                    directionUp = true;
.
.

     }

.
.
println("Time: "+time + " X: "+ xPos +" Y: "+height);
.
.
time+= INTERVAL_TIME;
pause(INTERVAL_TIME*1000); //units are ms
.
}
```

Here is an example of the first few values in the console output starting from a height of 40 m with an energy loss of 0.1:

```
Time:0.0 X: 0.0 Y:40.0
Time:0.1 X: 0.0 Y:39.951
Time:0.2 X: 0.0 Y:39.804
```

Also note the use of the `pause()` function above to slow down the loop, in order to make the simulation animation run at a reasonable speed.

It is a good idea to start by implementing the algorithm while displaying the time, x, and y values so you can see if your output makes sense. You can then concentrate on displaying the ball in the proper coordinate frame.

The concepts that you need can be found in the text as follows:
- Algorithmic design (Ch 1)
- Programming syntax (Ch 1)
- Extending existing class and inheriting behaviour (methods etc.) (Ch 2)
- Variables, constants, operations (Ch 3)
- Expressions, assignment statements, etc. (Ch 3)
- Control statements / Blocks (iteration, conditional) (Ch 4)
- Logical operators (Ch 4)

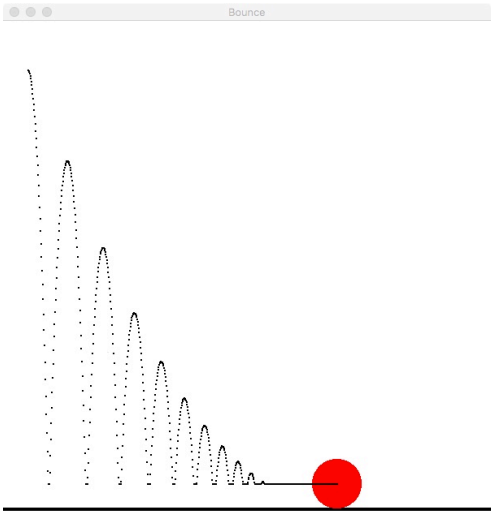
For example, let us look at the `while` loop. As the name implies, code within the loop is executed repeatedly until the looping condition is no longer satisfied or a `break` statement is executed within the loop. The general form of a while loop is

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

In the example above, the logical condition is set to check if a simulation time we defined at the beginning has been met, which means the loop executes (incrementing time) until we reach the TIMEOUT value.
For the final simulation you hand in, use a ball diameter of 6 metres, a time increment of 0.1 seconds, and a timeout of 30 seconds. When displaying the ball, use a conditional statement to make sure the ball displays as above the ground even if the `h0` value is slightly negative (at ground impact).

Instructions

1. Write a Java class, `Bounce.java`, that implements the simulation outlined above. 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_A1.
2. Test your program for various drop heights and collision losses. Make sure that the simulation produces correct results relative to the simple model used.
3. Bonus. Once you have your program working in the vertical direction, it is not difficult to incorporate motion along the horizontal axis (i.e. $x = v_x T$). Note: be careful here – for calculating height, t is reset to zero at each change of direction (time relative to last direction change). The variable T used here refers to time since the beginning of simulation. It is also useful to provide a *trace* of the ball trajectory as shown below in Figure 3.



To get the full 10 marks for the Bonus, your simulation should replicate the output of

Figure 3. Hand in a screen capture corresponding to Figure 3 for your simulation.

To Hand In:

1. The source file, Bounce.java.
2. A screen capture file for the Bonus.

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.

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