Department of Electrical and Computer Engineering
ECSE 202 – Introduction to Software Development
Assignment 3
Keeping Track of Objects

Introduction

This assignment is about data structures and objects, specifically the use of B-Trees to organize data in an explicit order. It extends the work done previously in Assignment 2 to include a mechanism for keeping track of all objects generated in order to i) determine when the entire set of balls has stopped moving and ii) to access each ball in order of size. You will use this new capability to generate the program described below. This assignment makes use of material from the Data Structures lectures.

Problem Description

Extend the program in Assignment 2 as follows. When the last ball stops moving (Figure 1a), determine the set of ball sizes and arrange the balls in stacks as shown in Figure 1b. Each stack is comprised of balls of approximately the same size (the precise meaning of *approximate* in this case will be formally defined a bit later). Your program should operate as follows:

1. When launched, the simulation starts and runs until the last ball stops moving. At this point it should stop and prompt the user with the message "CR to continue" . The display should appear as shown in Figure 1a.
2. When the mouse is clicked, the program proceeds with the ball sort and produces the display shown in Figure 1b, with an "All Stacked" message replacing the earlier "CR to continue."

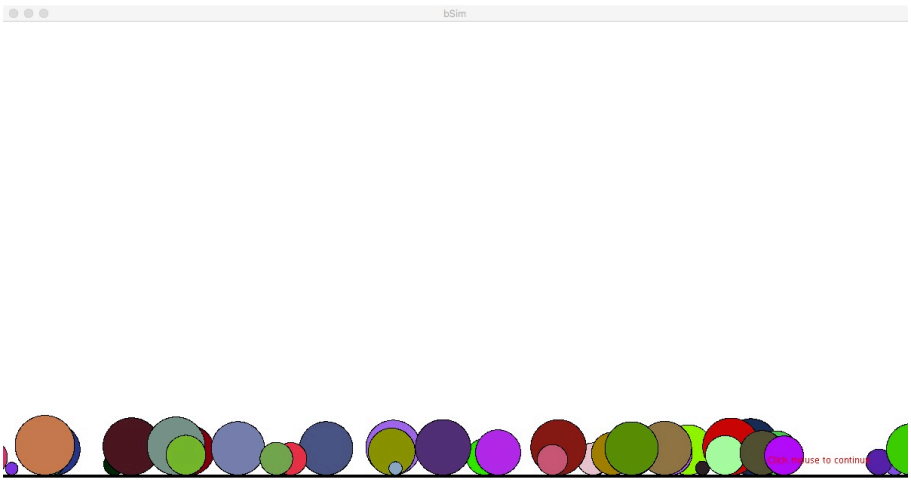


Figure 1a

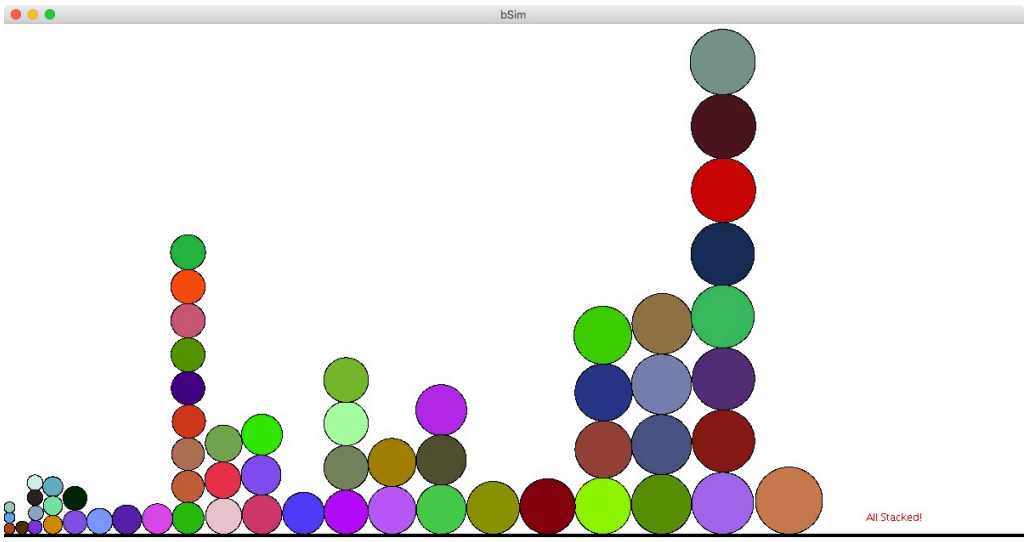


Figure 1b

As shown in Figure 1b, for the 60 balls generated in Figure 1a, there are 21 distinct size classes – i.e., a size class is defined as a set of balls of *approximately* the same size as follows:

1. Given a set of balls ordered from smallest to largest.
2. For I = 1 to Number of balls in set
3. If current size – last size > DELTASIZE
4. Start a new stack
5. Else
6. Put current ball on top of last ball
7. End

Algorithm 1

For testing and submission of your program, please make sure to use the following parameters. Anything defined as a *double* corresponds to world (simulation) coordinates in units of meters, kilogams, seconds (MKS).

```
// 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;   // pixel/meter
private static final int NUMBALLS = 60;           // # balls
private static final double MINSIZE = 1.0;        // Min radius
private static final double MAXSIZE = 7.0;        // Max radius
```

```
private static final double EMIN = 0.2;           // Min loss
private static final double EMAX = 0.6;           // Max loss
private static final double VoMIN = 40.0;         // Min velocity
private static final double VoMAX = 50.0;         // Max velocity
private static final double ThetaMIN = 80.0;      // Min angle
private static final double ThetaMAX = 100.0;     // Max angle
```

Within the aBall class, the value of k = 0.0001. Your program should generate parameters in the same order as Assignment 2. The random number generator should have an initial seed value set by: rgen.setSeed((**long**) 424242); This should produce the same display as shown in Figure 1b.

Design Approach

The first requirement is a data structure to hold the aBall objects generated and methods for adding new data to the B-Tree, and for tree traversal. Using the bTree class code posted on myCourses is a good starting point, but you will need to modify it to store aBall objects. Consequently in the run method of your bSim class, one of the things that you need to do is create an instance of the bTree class:

```
bTree myTree = new bTree();
```

You will have to modify the addNode method to accommodate the aBall object,

```
void addNode(aBall iBall);      // the argument is of type aBall
```

create a new method based on the in-order traversal routine that scans the B-Tree and checks the status of each aBall,

```
boolean isRunning();            // returns true if simulation still running
```

and finally a second new method based on the in-order traversal routine to move a ball to its sort order position instead of printing,

```
void stackBalls();              // this will move selected aBalls to their sorted
                                // position
```

The stackBalls method operates by traversing the B-Tree and moving each ball to either the top of the last ball placed, or at the start of a new stack to the right (Algorithm 1). The balls should be in contact with other as shown in Figure 1b without overlap.

Here is how this code might appear in your simulation class:

```
// Set up random number generator & B-Tree

      RandomGenerator rgen = RandomGenerator.getInstance();
      bTree myTree = new bTree();
```

```
      rgen.setSeed((long) 424242);

// In the aBall generation loop

      aBall iBall = new aBall(Xi,Yi,iVel,iTheta,iSize,iColor,iLoss);
      add(iBall.myBall);
      myTree.addNode(iBall);

// Following the aBall generation loop

      while (myTree.isRunning());              // Block until termination
      Code to add a GLabel to the display      // Prompt user
      Code to wait for a mouse click           // Wait
      myTree.stackBalls();                     // Lay out balls in order
```

Modifications will also be needed to the aBall class. You will need to create an instance variable that indicates whether the while loop is active (that can be interrogated by the isRunning method), and a method to move a ball to a specified location, void moveTo(double x, double y), where (x,y) are in simulation coordinates.

Working Inside of a Recursion

Recall the form of a recursive tree traversal:

```
      private void traverse_inorder(bNode root) {
            if (root.left != null) traverse_inorder(root.left);
            <code to process data at the current node>
            if (root.right != null) traverse_inorder(root.right);
      }
```

Any variables declared locally inside of traverse_inorder() disappear when the current instance returns. Suppose you wanted to place each ball in succession from left to right. If X and Y represent the position of the ball, then you could **not** do this:

```
      private void traverse_inorder(bNode root) {
            double X,Y,lastSize=0;
            if (root.left != null) traverse_inorder(root.left);
            // processing for current node
            Get size of ball at current node.
            Update values of X and Y to determine where to place it.
            moveTo(X,Y) to place the ball there
            //
            if (root.right != null) traverse_inorder(root.right);
      }
```

In order to work, variables such as X, Y, lastSize, etc., need to be *persistent* across recursive calls to traverse_inorder. An easy way to do this is to make them instance variables of the bTree() class where traverse_inorder is defined. It's not hard to see how stackBalls() can be derived from traverse_inorder() with a little bit of thought.

Instructions

1. Modify the bTree class, `bTree.java`, to accommodate aBall objects and write the additional methods described earlier. For this assignment, the value of parameter DELTASIZE = 0.1.
2. Modify the aBall class, `aBall.java`, to provide a method for returning run state and moving the and placing the balls as specified.
3. Modify the bSim class, `bSim.java`, to complete the program design.
4. Before submitting your assignment, run your own tests to verify correct operation.

To Hand In:

1. The source java files. Note – use the default package.
2. A screenshot file (pdf) showing the state of the simulation at the user prompt as in Figure 1a.
3. A screenshot file showing the end of the program with the balls in sort order as in Figure 1b.

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

Note:

1. Instance variables must be defined as **private** and accessed using appropriate getter and setter methods.
2. You must use the parameters defined above.
3. Code must be appropriately commented using Javadoc convention as a minimum standard.

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

Your screenshot files should be named A3-1.pdf and A3-2.pdf 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, especially when the assignment is a variation from the previous semester.

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 October 10, 2019

```java
import java.awt.Color;

import acm.graphics.GLabel;
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 = 60;                         // # balls to simulate
    private static final double MINSIZE = 1.0;                      // Minumum ball radius
    private static final double MAXSIZE = 7.0;                      // Maximum ball radius
    private static final double EMIN = 0.2;                         // 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) 424242);

// Create a bTree for storing links to balls generated

    bTree myTree = new bTree();

// 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
            myTree.addNode(iBall);                                              // Add link to tree
            iBall.start();                                                      // Start this instance
        }

// Wait until simulation stops

        while (myTree.isRunning());         // Block until simulation terminates


// For standalone application with no console, use graphics display

        GLabel myLabel = new GLabel("Click mouse to continue");
        myLabel.setLocation(WIDTH-myLabel.getWidth()-50,HEIGHT-myLabel.getHeight()-50);
        myLabel.setColor(Color.RED);
        add(myLabel);
        waitForClick();
        myLabel.setLabel("All Stacked!");
        myTree.stackBalls();        // Lay out balls from left to right in size order
    }
}
```

```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;

    // 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);
//                  System.out.println("Bounce");

        // 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 - getBall
 */

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

```
}

/**
 * Access methods for instance variables - getSize
 */

public double getSize () {
    return bSize;
}

/**
 * Method to indicate if the thread is running or not.
 * Thread runs as long as the ball still has enough energy.
 */

public boolean getbState () {
    return hasEnoughEnergy;
}

/**
 * Method to move corresponding GOval to specified location
 */

void moveTo(double x, double y) {
    myBall.setLocation(gUtil.XtoScreen(x-bSize),gUtil.YtoScreen(y+bSize));
}



/**
 * 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;
private boolean hasEnoughEnergy = true;         // loop control variable must now
                                                // be an instance variable to return state



// 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


}
```

```
import java.awt.Color;

import acm.graphics.GLabel;
import acm.graphics.GOval;

/**
 * Implements a B-Tree class for storing bBall objects
 * @author ferrie
 *
 */

public class bTree {

    // Instance variables

    bNode root=null;

    /**
     * addNode method - adds a new node by descending to the leaf node
     *                  using a while loop in place of recursion.  Ugly,
     *                  yet easy to understand.
     */


        public void addNode(aBall data) {

            bNode current;

    // Empty tree

            if (root == null) {
                root = makeNode(data);
            }

    // If not empty, descend to the leaf node according to
    // the input data.

            else {
```

```
                current = root;
                while (true) {
                    if (data.getSize() < current.data.getSize()) {

    // New data < data at node, branch left

                        if (current.left == null) {             // leaf node
                            current.left = makeNode(data);      // attach new node here
                            break;
                        }
                        else {                                  // otherwise
                            current = current.left;             // keep traversing
                        }
                    }
                    else {
    // New data >= data at node, branch right

                        if (current.right == null) {            // leaf node
                            current.right = makeNode(data);     // attach
                            break;
                        }
                        else {                                  // otherwise
                            current = current.right;            // keep traversing
                        }
                    }
                }
            }

        }

/**
 * makeNode
 *
 * Creates a single instance of a bNode
 *
 * @param   int data   Data to be added
 * @return  bNode node Node created
 */
```

```
        bNode makeNode(aBall data) {
            bNode node = new bNode();                   // create new object
            node.data = data;                           // initialize data field
            node.left = null;                           // set both successors
            node.right = null;                          // to null
            return node;                                // return handle to new object
        }


/**
 * inorder method - inorder traversal via call to recursive method (debugging tool)
 */

    public void inorder() {
        traverse_inorder(root);
    }

/**
 * traverse_inorder method - recursively traverses tree in order and prints each node.
 * This is used for debugging purposes to check if the tree is properly built.
 */

    private void traverse_inorder(bNode root) {
        if (root.left != null) traverse_inorder(root.left);
        System.out.println(root.data.getSize());
        if (root.right != null) traverse_inorder(root.right);
    }

/**
 * isRunning predicate - determines if simulation is still running
 */

    boolean isRunning() {
        running=false;
        recScan(root);
        return running;
    }
```

```
    void recScan(bNode root) {
        if (root.left != null) recScan(root.left);
        if (root.data.getbState()) {
            running=true;
            return;
        }
        if (root.right != null) recScan(root.right);
    }

/**
 * clearBalls - removes all balls from display
 * (note - you need to pass a reference to the display)
 *
 */

    void clearBalls(bSim display) {
        recClear(display,root);
    }

    void recClear(bSim display,bNode root) {
        if (root.left != null) recClear(display,root.left);
        display.remove (root.data.getBall());
        if (root.right != null) recClear(display,root.right);
    }

/**
 * stackBalls - rearranges all the balls in the display in size order
 * from left to right.  Balls of the same size are stacked one on top
 * of another
 *
 */

    void stackBalls() {
        lastSize = 0;                   // Indicates start of stacking
        Xcurrent = 0;                   // Start at left hand side of window
        Ycurrent = 0;                   // Start on the ground
        recStack(root);
```

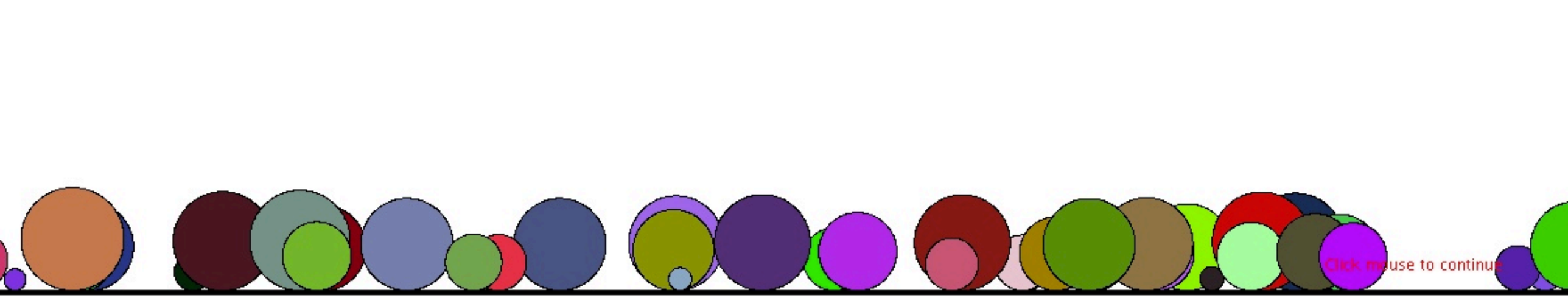
```
}

void recStack(bNode root) {

    if (root.left != null) recStack(root.left);     // traverse left

//
// If the current ball is not the same size as the last ball, start
// a new stack.
//

    if (root.data.getSize()-lastSize > DELTASIZE) {
        if (lastSize == 0) {
            Xcurrent  = root.data.getSize();    // Determine new stack position
            Ycurrent  = Xcurrent;
        }
        else {
            Xcurrent += lastSize+root.data.getSize();
            Ycurrent = root.data.getSize();
        }
        root.data.moveTo(Xcurrent, Ycurrent);   // and move ball there
    }

//
// Otherwise, move the current ball on top of the last ball
//
    else {
        Ycurrent += 2*lastSize;                 // Increment Y position
        root.data.moveTo(Xcurrent, Ycurrent);   // and move ball there
    }

    lastSize = root.data.getSize();                 // For next move

    if (root.right != null) recStack(root.right);   // traverse right
}
```

```
// Example of a nested class //

    public class bNode {
        aBall data;
        bNode left;
        bNode right;
    }

// Instance variables (visible to all methods)

    private boolean running;
    private double Xcurrent;
    private double Ycurrent;
    private double lastSize;

// Parameters

    private static final double DELTASIZE = 0.1;


}
```

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

    private static final int WIDTH = 1800;                          // 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

All Stacked!