ECSE 202 – Introduction to Software Development October 27, 2019
Assignment 4
Due November 11, 2019 at 5:00 pm

Background

The simulation in Assignment 3 has no user interface, it simply runs according to a set of hard-wired parameters. In this assignment you will add a simple user interface as depicted below in Figure 1.

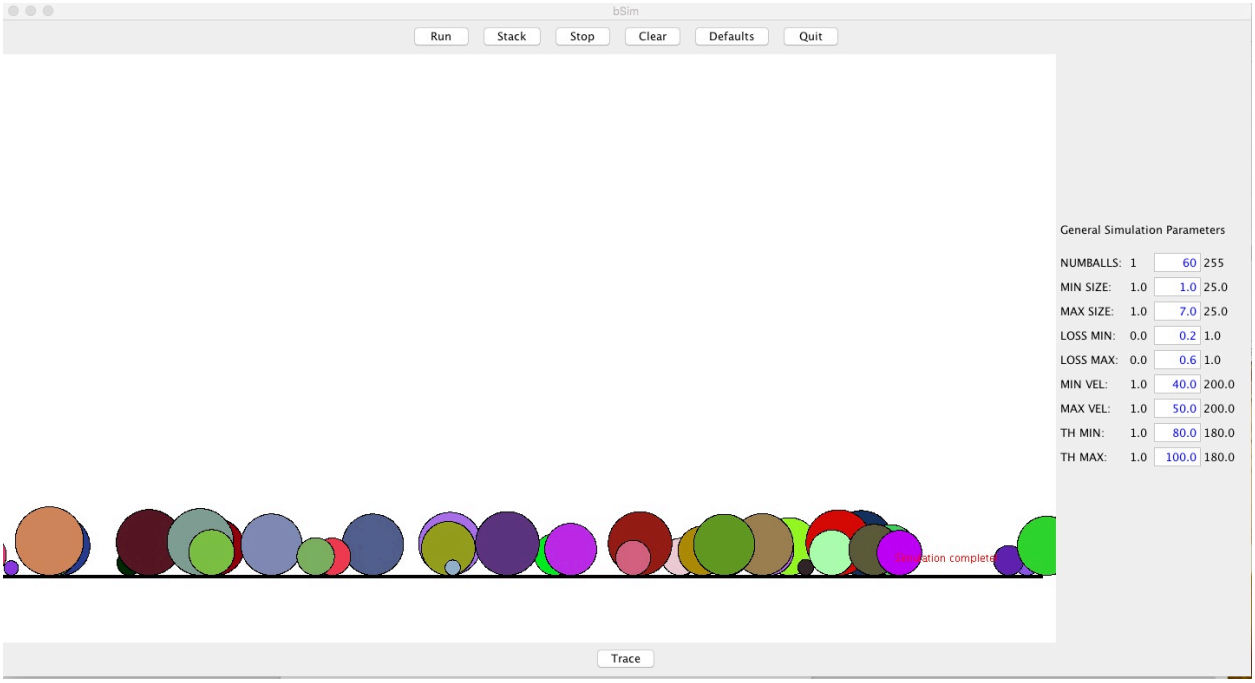


Figure 1

As can be seen in the figure, the simulation incorporates a JPanel that includes a means of entering simulation parameters. When the user types in a value into the corresponding field and hits return, the corresponding parameter is updated in the program. The JPanel uses the TableLayout manager to line up labels and fields accordingly. For the remainder of the user interface, the default BorderLayout manager is used to place the control buttons along the Northern border, the parameter JPanel along the East, and the single Trace button along the Southern border as shown in Figure 1. As its label implies, if pushed, the trace of each ball is plotted on the screen in the same color as the ball. Together these features allow considerable flexibility over the program in Assignment 3.

Requirements

1. Functionality

Your program must allow the user to perform the following functions. How you implement the User Interface is entirely up to you (e.g. JComboBox, push-buttons, etc.). In other words, you do not have to replicate the example in Figure 1.

Run: Starts a new simulation with parameters chosen from the current parameter set. You should be able to run multiple simulations with new balls added to the display at each Run instance.
Stack: Stacks **all** the balls created to date. For example, assume that you have run a simulation and stacked the balls. Now you run another simulation and stack again. This time all the balls in the previous stack are included in the current one. As long as you have not created a new bTree, all balls will be included when stacked.
Clear: Clears the current display and kills the simulation attached to each ball. If the simulation is run again with new balls created, balls created previously are effectively eliminated.
Stop: Stops the current simulation, freezing the current display.
Quit: Exits the program.

The example includes an additional button labeled Defaults. This is optional, and provides a means to restore simulation parameters to their default values.

In addition, your program must include a Trace mode that can be enabled or disabled. When enabled, the output resembles the output of Assignment 1 where a marker is placed at location traversed by the ball. In this implementation the color of the "dot" should match the color of the ball it represents. The display should be set up using with control functions laid across the top row, parameters along the right side of the display, and the Trace mode button at the bottom of the display. This is most easily accomplished using the BorderLayout manager. However you choose to implement selection, it must be event-driven so that the program reacts accordingly.

2. Parameter Entry

The key requirements are that i) all the parameters inputs should appear within a single JPanel that gets added to the display as shown, and ii) the user must be able to enter numerical parameters using the keyboard. The example in Figure 1 is fairly easy to implement overriding only the actionPerformed listener. You can implement this example quite simply using Int/Double field boxes that generate ActionEvents (see the Temperature Conversion example in the notes). By the way, you can also do this within GraphicsProgram, but a workaround is needed for it to work correctly (see below).

Implementation

Most of the work in this assignment is related to modifying bSim to incorporate user interface elements as well as re-packaging the simulation code and stacking code as methods that can be called from the user interface. In particular, you should create a doSim() method that generates a

set of balls, attaches them to the screen, and waits for the simulation to complete. Rather than immediately proceeding to ball stacking on completion, doSim() should return. Stacking functionality should be packaged as a method, doStack(), which traverses myTree and generates the appropriate display. Unfortunately, if you try to call doSim() on an event, it will run, but not update the display. This has to do with how GraphicsProgram is implemented and is beyond the scope of this course. If you're experienced writing Java code, then you can (if you haven't already in Assignment 3) use a different framework outside of acm. Otherwise, a workaround is shown below:

The usual way of handling a user selection would look something like the following:

1. Restructure bSim so that the simulation component (generate the set of bBalls) is set up as a method that can be called more than once, e.g., doSim();
2. Set up an entry for doSim() in a JComboBox or push button.
3. Dispatch to doSim() when the corresponding entry is selected as shown below:

```
public void itemStateChanged(ItemEvent e) {
        JComboBox source = (JComboBox)e.getSource();

        if (source==bSimC) {          \
            if (bSimC.getSelectedIndex()==1) {
                System.out.println("Starting simulation");
                doSim();
            }
            else if (bSimC.getSelectedIndex()==2) {
                System.out.println("Histogramming balls");
                doHist();
            }
            etc…
        }
    }
```

Unfortunately, do to the way in which acm is implemented, your simulation will run, but not update the screen. An inelegant solution to this problem is as follows:

```
public void init() {
    simEnable=false;                                    // this is an instance variable

    …other code…

        while(true) {
            pause(200);
            if (simEnable) {                            // Run once, then stop
                doSim();
                bSimC.setSelectedIndex(0);
```

```
                    simEnable=false;
                }
            }
}
```

The idea is to run the simulation from within the init() method rather than call it from itemStateChanged(). Variable simEnable allows the simulation to run exactly once if simEnable=true.

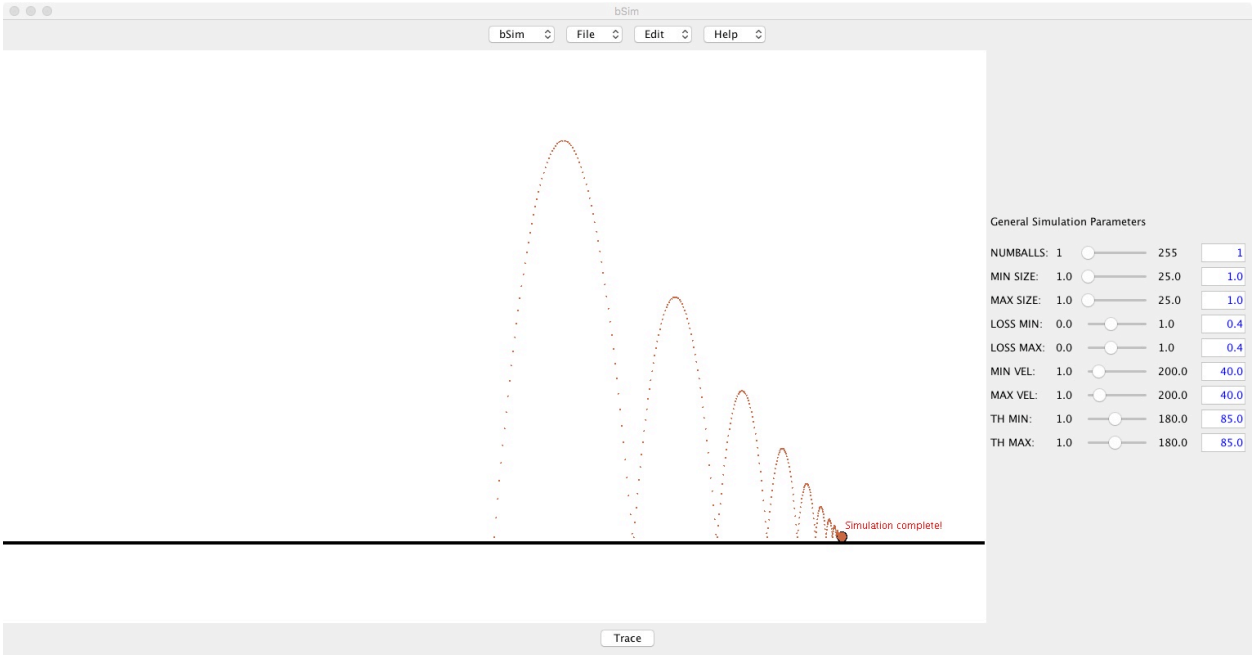
Back at itemStateChanged(), instead of calling doSim(), we simply change this to simEnable=true. This method is referred to as *polling* – you will encounter it again in ECSE 211, Design Principles and Methods.

Instructions

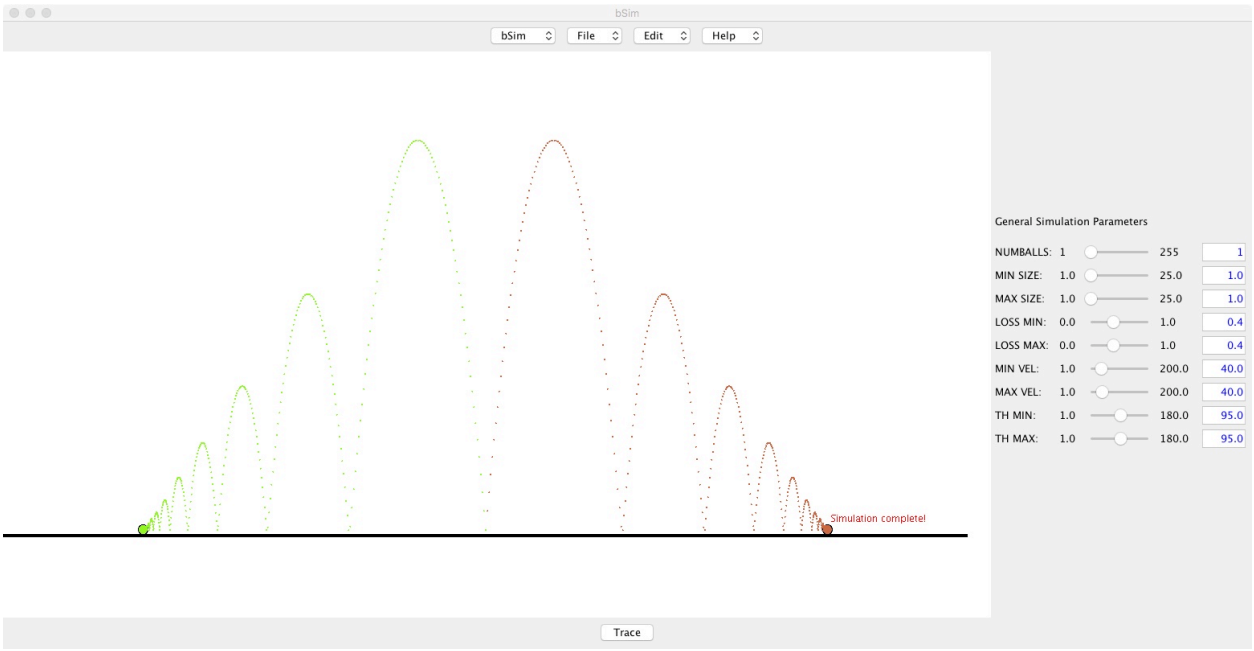
Modify the bSim class to implement the specified user interface. Take note that the display area shown in Figure 1 must match the display area in Assignment 3. In other words, you need to add additional space to incorporate the parameter panel. In A3 WIDTH corresponded to the entire display. Now it corresponds to the width of the display less the width of the parameter panel. You might also need to modify aBall to incorporate the trace feature.

To Hand In

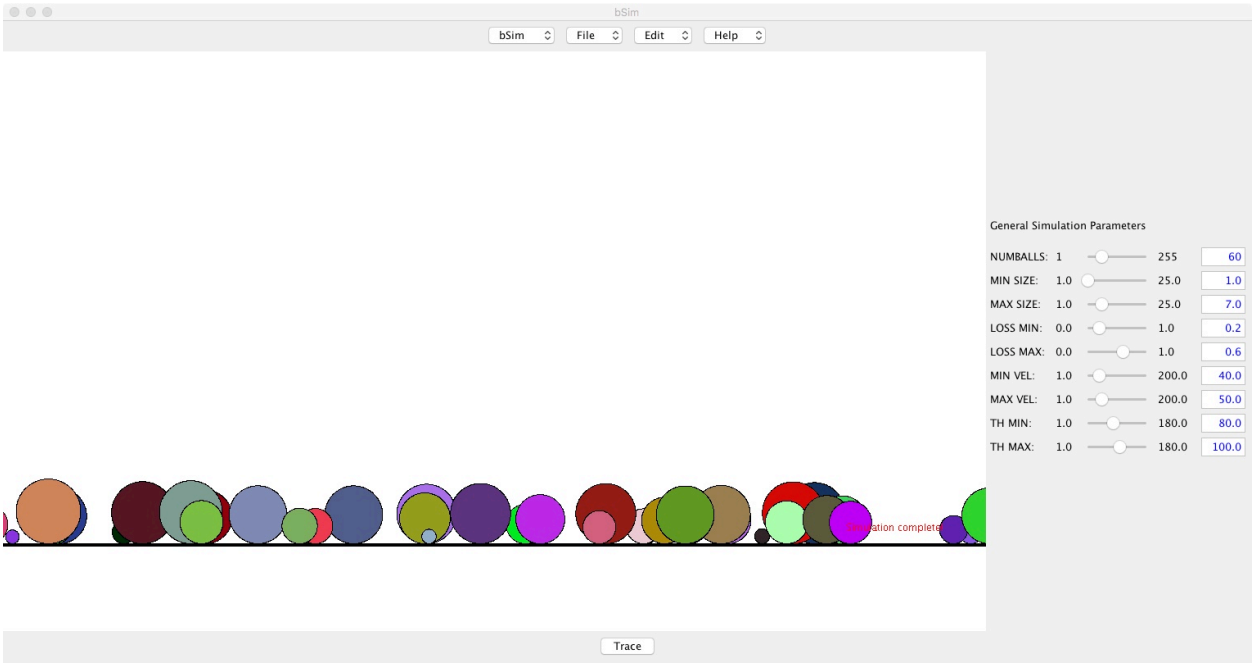
1. The source java files. Note – use the default package. Make sure that the random number generator is seeded with long integer value = 424242; this will ensure that your output matches ours if your simulations are correct.
2. Run a simulation with a single ball, starting at the center of the display field, with ball size = 1.0, loss = 0.40, velocity = 40.0, angle = 85.0 (you do this by making the min and max parameters equal to the specified value). Enable trace. This should replicate the output of Assignment 3, Part 1 as shown below. Take a screenshot of the display and hand it in as a pdf file named A4-1.pdf.


bSim
bSim
File
Edit
Help
Simulation complete!
General Simulation Parameters
NUMBALLS: 1 255 1
MIN SIZE: 1.0 25.0 1.0
MAX SIZE: 1.0 25.0 1.0
LOSS MIN: 0.0 1.0 0.4
LOSS MAX: 0.0 1.0 0.4
MIN VEL: 1.0 200.0 40.0
MAX VEL: 1.0 200.0 40.0
TH MIN: 1.0 180.0 85.0
TH MAX: 1.0 180.0 85.0
Trace

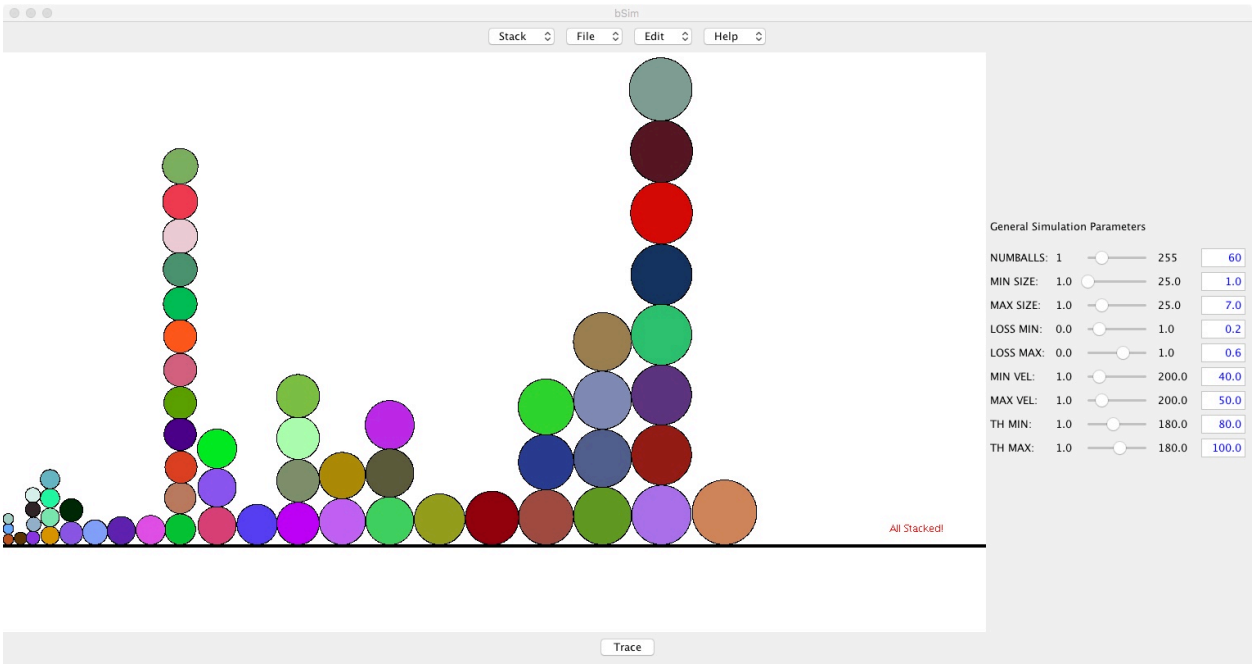
3. Without altering the display from Part (2), change the angle parameter from 85° to 95° and run the simulation again. Take a screenshot of the display and save as a pdf file named A4-2.pdf. It should produce the output shown below.



4. Without leaving the program, enter a clear command to remove the current bTree and erase the display (same conditions as when the program is started from scratch. Run a simulation with the exact parameters as the last run from Assignment 3. On completion, take a screenshot of the display and save as a pdf file named A4-3.pdf. It should produce the output shown below.



5. Finally, with the display as shown above, issue a stack command. Take a screenshot of the display and save as a pdf file named A4-4.pdf. It should produce the output shown below.



Notes:

Since this is the last Java assignment, we will pay particular attention to the documentation of your code. We know that assignments from previous years are floating around on the Web, and naturally we will run comparisons of your code against them. The point of these assignments is for you to develop the requisite skills that will carry you through the rest of the program.

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 27, 2019
simplified version posted December 6, 2019

```
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;

import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.JToggleButton;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

import acm.graphics.GLabel;
import acm.graphics.GObject;
import acm.graphics.GOval;
import acm.graphics.GPoint;
import acm.graphics.GRect;
import acm.gui.TableLayout;
import acm.program.GraphicsProgram;
import acm.util.RandomGenerator;

/**
 * The main class for Assignment 4.
 * This assignment adds an interactive layer to the ballistic ball demo
 * of Assignment 3.  It provides for the following:
 *
 * A simple menu-driven interface consisting of JButtons for selecting
 * commands and TextFields (Int/Double) for entering parameters.
 * The following commands are supported:
 *
 * 1. Run   Starts a simulation based on the current parameter set.  The
 *          simulation runs to completion before another simulation can
 *          be started.
 * 2. Stack Stacks all the balls in the current B-Tree as in A3.
 * 3. Stop  Stops a running simulation.  Balls freeze.
 * 4. Clear Completely resets the program to its initial state.
```

```
 * 5. Quit  Exits the program
 *
 * Optional:
 *
 * 6. Defaults - resets parameters to their default values.
 *
 * Trace Button - if toggled on, any ball created generates a trace of its
 *                path.
 *
 * Note:  The intent here is to augment an existing program without having
 *        to make extensive hanges.
 *
 * @author ferrie
 *
 */

public class bSim extends GraphicsProgram {

// Parameters used in this program

    public static final int PANELWIDTH = 350;           // Width of slider panel
    public static final int WIDTH = 1200 + PANELWIDTH;  // Total screen width
    public static final int HEIGHT = 600;               // Height above the ground plane
    public static final int OFFSET = 200;               // Height of ground plane
    public static final double SCALE = HEIGHT/100;      // pixels per meter
    public static final double PD = 1.0;                // trace point diameter
    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
    private static final long SEED = 424242;            // RNG seed

/**
```

```
 * 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);
        }

 /**
 * The init method is the entry point for this program. This is where we set up the GUI
 */

    public void init() {

// Window and global objects

        this.resize(WIDTH,HEIGHT+OFFSET);          // optional, initialize window size
        ScrWidth = this.getWidth() - PANELWIDTH;   // actual screen width
        rgen = RandomGenerator.getInstance();
        rgen.setSeed((long) SEED);                 // RNG
        myTree = new bTree();

// Set up the User Interface (UI)

        setButtons();                                // Add buttons
        setPanel();                                  // Set up Parameter Panel
        setActionCommands();                         // Must be done explicitly for NumFields

// Create the ground plane (after the panel has been created).

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

// Set up the Listeners
```

```
        addActionListeners();   // Buttons and NumField events

/**
 *  Main simulation loop
 *
 *  doSim and doHist will run if dispatched from an action listener, but the
 *  display will not update (limitation of acm?).  We run it indirectly by
 *  changing a control variable.
 */

        while(true) {
            pause(200);
            if (simEnable) {                        // Run once, then stop
                doSim();
                simEnable=false;
            }
        }

    }

// Event-handler Code:
//
//  The code below overrides the default event handlers for the
//  enabled listeners.
//

/**
 * Method to handle Action Commands from buttons and NumFields.
 */

    public void actionPerformed(ActionEvent e) {
        String cmd = e.getActionCommand();

// Buttons

        if (cmd.equals("Run")) {
            if (myLabel != null) remove(myLabel);
```

```
            System.out.println("Starting simulation");
            simEnable=true;
        }

        else if (cmd.equals("Stack")) {
            if (myLabel != null) remove(myLabel);
            System.out.println("Stacking balls");
            doStack();
        }

        else if (cmd.equals("Stop")) {
            if (myLabel != null) remove(myLabel);
            System.out.println("Stopping simulation");
            myTree.killSim();
        }

        else if (cmd.equals("Clear")) {
            System.out.println("Clearing simulation");
            removeAll();
            add(gPlane);
            myTree.killSim();
            myTree.clearBalls(this);
            myTree = new bTree();
        }

        else if (cmd.equals("Defaults")) {
            resetParameters();
        }

        else if (cmd.equals("Quit")) {
            System.out.println("Shutting down");
            System.exit(0);
        }

// NumFields

        else if (cmd.equals("NumBalls")) {
            PS_NumBalls=NumBalls.sIReadout.getValue();  // get value
```

```
        }
        else if (cmd.equals("MinSize")) {
            PS_MinSize=MinSize.sDReadout.getValue();    // get value
        }
        else if (cmd.equals("MaxSize")) {
            PS_MaxSize=MaxSize.sDReadout.getValue();    // get value
        }
        else if (cmd.equals("VoMin")) {
            PS_VoMin=VoMin.sDReadout.getValue();    // get value
        }
        else if (cmd.equals("VoMax")) {
            PS_VoMax=VoMax.sDReadout.getValue();    // get value
        }
        else if (cmd.equals("ThetaMin")) {
            PS_ThetaMin=ThetaMin.sDReadout.getValue();    // get value
        }
        else if (cmd.equals("ThetaMax")) {
            PS_ThetaMax=ThetaMax.sDReadout.getValue();    // get value
        }
        else if (cmd.equals("EMin")) {
            PS_EMin=EMin.sDReadout.getValue();      // get value
        }
        else if (cmd.equals("EMax")) {
            PS_EMax=EMax.sDReadout.getValue();    // get value
        }
    }

/**
 * The doSim method encapsulates the simulation code from
 * the previous assignments.  Note that the two parts of the
 * program have been separated and set up as two menu items.
 */

    private void doSim() {

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

        for (int i=0; i<PS_NumBalls; i++) {
```

```
            double iSize = rgen.nextDouble(PS_MinSize,PS_MaxSize);  // Current size
            double Xi = ScrWidth/(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(PS_EMin,PS_EMax);        // Current loss coefficient
            double iVel = rgen.nextDouble(PS_VoMin,PS_VoMax);       // Current velocity
            double iTheta = rgen.nextDouble(PS_ThetaMin,PS_ThetaMax); // Current launch angle
// Trace option
            if (myTrace.isSelected())
                iBall = new aBall(Xi,Yi,iVel,iTheta,iSize,iColor,iLoss,this);  // Generate instance
            else
                iBall = new aBall(Xi,Yi,iVel,iTheta,iSize,iColor,iLoss);
            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

// Display completed message

        myLabel = new GLabel("Simulation complete!");
        myLabel.setLocation(ScrWidth-myLabel.getWidth()-50,HEIGHT-myLabel.getHeight());
        myLabel.setColor(Color.RED);
        add(myLabel);
    }

/**
 * Ball stacking is now done as a separate method.
 *
 */

    void doStack() {

// First check to see if a tree has been created.  If yes, then
```

```
// proceed to stack balls from left to right in ascending size.

        if (myTree.getRoot() != null) {
            myTree.stackBalls();
            myLabel = new GLabel("All Stacked!");
            myLabel.setLocation(ScrWidth-myLabel.getWidth()-50,HEIGHT-myLabel.getHeight());
            myLabel.setColor(Color.RED);
            add(myLabel);
        }
    }

/**
 * Method to set up buttons used in simulation program
 */

    void setButtons() {
        myTrace = new JToggleButton("Trace");
        add(myTrace,SOUTH);
        myRun = new JButton("Run");
        add(myRun,NORTH);
        myStack = new JButton("Stack");
        add(myStack,NORTH);
        myStop = new JButton("Stop");
        add(myStop,NORTH);
        myClear = new JButton("Clear");
        add(myClear,NORTH);
        myDefaults = new JButton("Defaults");
        add(myDefaults,NORTH);
        myQuit = new JButton("Quit");
        add(myQuit,NORTH);
    }

    /**
     * This method sets up all the NumFields used in this program.
     */

        void setPanel() {
```

```
// Best done using the Table Layout Manager inside of a JPanel which
// subsequently gets added to the right side of the screen.
//
// First layout the general simulation parameters
//
        JPanel myPanel = new JPanel();
        myPanel.setLayout(new TableLayout(30, 1));
        myPanel.add(new JLabel("General Simulation Parameters"));
        myPanel.add(new JLabel("                                "));

        NumBalls = new panelBox("NUMBALLS: ",1,NUMBALLS,255);       // Number of balls
        myPanel.add(NumBalls.myPanel);

        MinSize = new panelBox("MIN SIZE: ",1.0,MINSIZE,25.0);      // Minimum ball size
        myPanel.add(MinSize.myPanel);

        MaxSize = new panelBox("MAX SIZE: ",1.0,MAXSIZE,25.0);      // Maximum ball size
        myPanel.add(MaxSize.myPanel);

        EMin = new panelBox("LOSS MIN: ",0.0,EMIN,1.0);         // Minimum energy loss
        myPanel.add(EMin.myPanel);

        EMax = new panelBox("LOSS MAX: ",0.0,EMAX,1.0);         // Maximum energy loss
         EMax.setFSlider(EMAX);
        myPanel.add(EMax.myPanel);

        VoMin = new panelBox("MIN VEL: ",1.0,VoMIN,200.0);          // Minimum ball velocity
        myPanel.add(VoMin.myPanel);

        VoMax = new panelBox("MAX VEL: ",1.0,VoMAX,200.0);          // Maximum ball velocity
        myPanel.add(VoMax.myPanel);

        ThetaMin = new panelBox("TH MIN: ",1.0,ThetaMIN,180.0); // Theta min
        myPanel.add(ThetaMin.myPanel);

        ThetaMax = new panelBox("TH MAX: ",1.0,ThetaMAX,180.0); // Theta max
        myPanel.add(ThetaMax.myPanel);
```

```
            add(myPanel,EAST);
        }

/**
 * Method to set up the action commands for objects that
 * do not do this in their constructors (i.e. NumFields).
 */
    void setActionCommands() {

// Only necessary to do this for the NumFields as this is
// done in the button constructors.

        NumBalls.sIReadout.addActionListener(this);
        NumBalls.sIReadout.setActionCommand("NumBalls");
        MinSize.sDReadout.addActionListener(this);
        MinSize.sDReadout.setActionCommand("MinSize");
        MaxSize.sDReadout.addActionListener(this);
        MaxSize.sDReadout.setActionCommand("MaxSize");
        EMin.sDReadout.addActionListener(this);
        EMin.sDReadout.setActionCommand("EMin");
        EMax.sDReadout.addActionListener(this);
        EMax.sDReadout.setActionCommand("EMax");
        VoMin.sDReadout.addActionListener(this);
        VoMin.sDReadout.setActionCommand("VoMin");
        VoMax.sDReadout.addActionListener(this);
        VoMax.sDReadout.setActionCommand("VoMax");
        ThetaMin.sDReadout.addActionListener(this);
        ThetaMin.sDReadout.setActionCommand("ThetaMin");
        ThetaMax.sDReadout.addActionListener(this);
        ThetaMax.sDReadout.setActionCommand("ThetaMax");
    }

/**
 * Method to restore parameters to their defaults
 */

    void resetParameters() {
        PS_NumBalls=NUMBALLS;       // Reset to defaults
```

```
        NumBalls.sIReadout.setValue(PS_NumBalls);
        PS_MinSize=MINSIZE;
        MinSize.sDReadout.setValue(PS_MinSize);
        PS_MaxSize=MAXSIZE;
        MaxSize.sDReadout.setValue(PS_MaxSize);
        PS_VoMin=VoMIN;
        VoMin.sDReadout.setValue(PS_VoMin);
        PS_VoMax=VoMAX;
        VoMax.sDReadout.setValue(PS_VoMax);
        PS_ThetaMin=ThetaMIN;
        ThetaMin.sDReadout.setValue(PS_ThetaMin);
        PS_ThetaMax=ThetaMAX;
        ThetaMax.sDReadout.setValue(PS_ThetaMax);
        PS_EMin=EMIN;
        EMin.sDReadout.setValue(PS_EMin);
        PS_EMax=EMAX;
        EMax.sDReadout.setValue(PS_EMax);
    }

/**
 * Instance variables - simulation parameters
 */

    private int PS_NumBalls=NUMBALLS;       // Global simulation parameters for the rgen
    private double PS_MinSize=MINSIZE;
    private double PS_MaxSize=MAXSIZE;
    private double PS_VoMin=VoMIN;
    private double PS_VoMax=VoMAX;
    private double PS_ThetaMin=ThetaMIN;
    private double PS_ThetaMax=ThetaMAX;
    private double PS_EMin=EMIN;
    private double PS_EMax=EMAX;

/**
 * Instance variables - Buttons
 */

    private JToggleButton myTrace;
```

```
    private JButton myRun;
    private JButton myStop;
    private JButton myStack;
    private JButton myClear;
    private JButton myDefaults;
    private JButton myQuit;

/**
 * Instance variables - panel boxes
 *
 */

    private panelBox NumBalls;
    private panelBox MinSize;
    private panelBox MaxSize;
    private panelBox VoMin;
    private panelBox VoMax;
    private panelBox ThetaMin;
    private panelBox ThetaMax;
    private panelBox EMin;
    private panelBox EMax;

/**
 * Instance variables - simulation
 */

    RandomGenerator rgen;
    bTree myTree;
    GObject obj;
    aBall match;
    aBall iBall;
    GPoint last;
    boolean simEnable;
    GLabel myLabel;
    GRect gPlane;
    int ScrWidth;
}
```

```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 kill simulation thread - simply sets
 * hasEnoughEnergy to false which exists the while loop.
 */

public void setbState (boolean state) {
    hasEnoughEnergy = state;
}

/**
 * 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*bSim.SCALE;
    double ScrY = bSim.HEIGHT - y*bSim.SCALE;
    GOval pt = new GOval(ScrX,ScrY,bSim.PD,bSim.PD);
    pt.setColor(bColor);
    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;
volatile 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

}
```

```
import java.awt.Color;

import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JSlider;

import acm.graphics.GRect;
import acm.gui.DoubleField;
import acm.gui.IntField;
import acm.gui.TableLayout;

/**
 * This class creates a panelBox object consisting of a NumFields with a set of
 * JLabels for defining minimum and maximum values.
 * @author ferrie
 *
 */
public class panelBox {

/**
 * This constructor creates an instance of a JPanel using the default
 * layout manager, and populates it as follows:
 *      Parameter: Name --- JLabel --- IntField --- JLabel
 *  @param Name - String, name of field
 *  @param min - Integer, min IntField value
 *  @param dValue - Integer, default IntField value
 *  @param max - Integer, max IntField value
 */

    public panelBox(String name, Integer min, Integer dValue, Integer max) {        //  Integer values
        myPanel = new JPanel();
        nameLabel = new JLabel(name);
        minLabel = new JLabel(min.toString());
        maxLabel = new JLabel(max.toString());
        sIReadout = new IntField(dValue);
        sIReadout.setForeground(Color.blue);
        myPanel.setLayout(new TableLayout(1,4));
        myPanel.add(nameLabel,"width=80");
```

```
        myPanel.add(minLabel,"width=25");
        myPanel.add(sIReadout,"width=60");
        myPanel.add(maxLabel,"width=50");
        imin=min;
        imax=max;
    }

    /**
     * This constructor creates an instance of a JPanel using the default
     * layout manager, and populates it as follows:
     *      Parameter: Name --- JLabel --- IntField --- JLabel
     *  @param Name - String, name of field
     *  @param min - Double, min IntField value
     *  @param dValue - Double, default IntField value
     *  @param max - Double, max IntField value
     */

    public panelBox(String name, Double min, Double dValue, Double max) {           // Floating point
     values
        myPanel = new JPanel();
        nameLabel = new JLabel(name);
        minLabel = new JLabel(min.toString());
        maxLabel = new JLabel(max.toString());
        sDReadout = new DoubleField(dValue);
        sDReadout.setForeground(Color.blue);
        myPanel.setLayout(new TableLayout(1,4));
        myPanel.add(nameLabel,"width=80");
        myPanel.add(minLabel,"width=25");
        myPanel.add(sDReadout,"width=60");
        myPanel.add(maxLabel,"width=50");
        fmin=min;
        fmax=max;
    }

/**
 * Instance Variables for this class
 *
 */
```

```
    public JPanel myPanel;
    private JLabel nameLabel;
    private JLabel minLabel;
    private JLabel maxLabel;
    public IntField sIReadout;
    public DoubleField sDReadout;
    public JSlider mySlider;
    private JLabel colorDisp;
    private int imin;
    private int imax;
    public double fmin;
    public double fmax;

}
```

```
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) {
        if (root == null) return;
        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
    }

/**
```

```
 * Returns a reference to the tree root
 *
 */

    public bNode getRoot() {
        return root;
    }

/**
 * killSim - kills all threads
 */

    void killSim() {
        if (root == null) return;
        recKill(root);
    }

    void recKill(bNode root) {
        if (root.left != null) recKill(root.left);
            root.data.setbState(false);
        if (root.right != null) recKill(root.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
     */
```

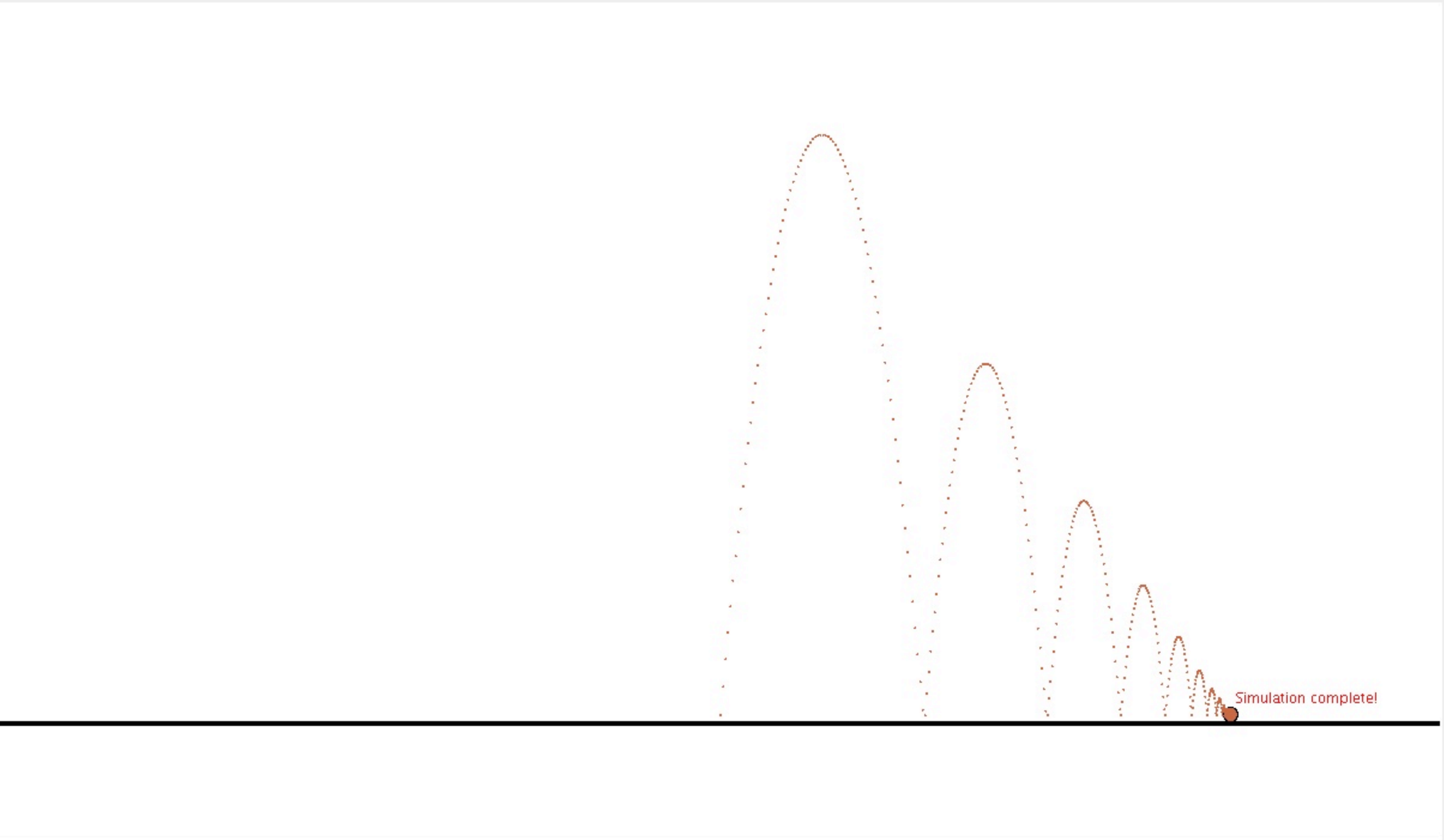
```
    static double LtoScreen(double length) {
        return length * SCALE;
    }

    /**
     * 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 File Edit Help



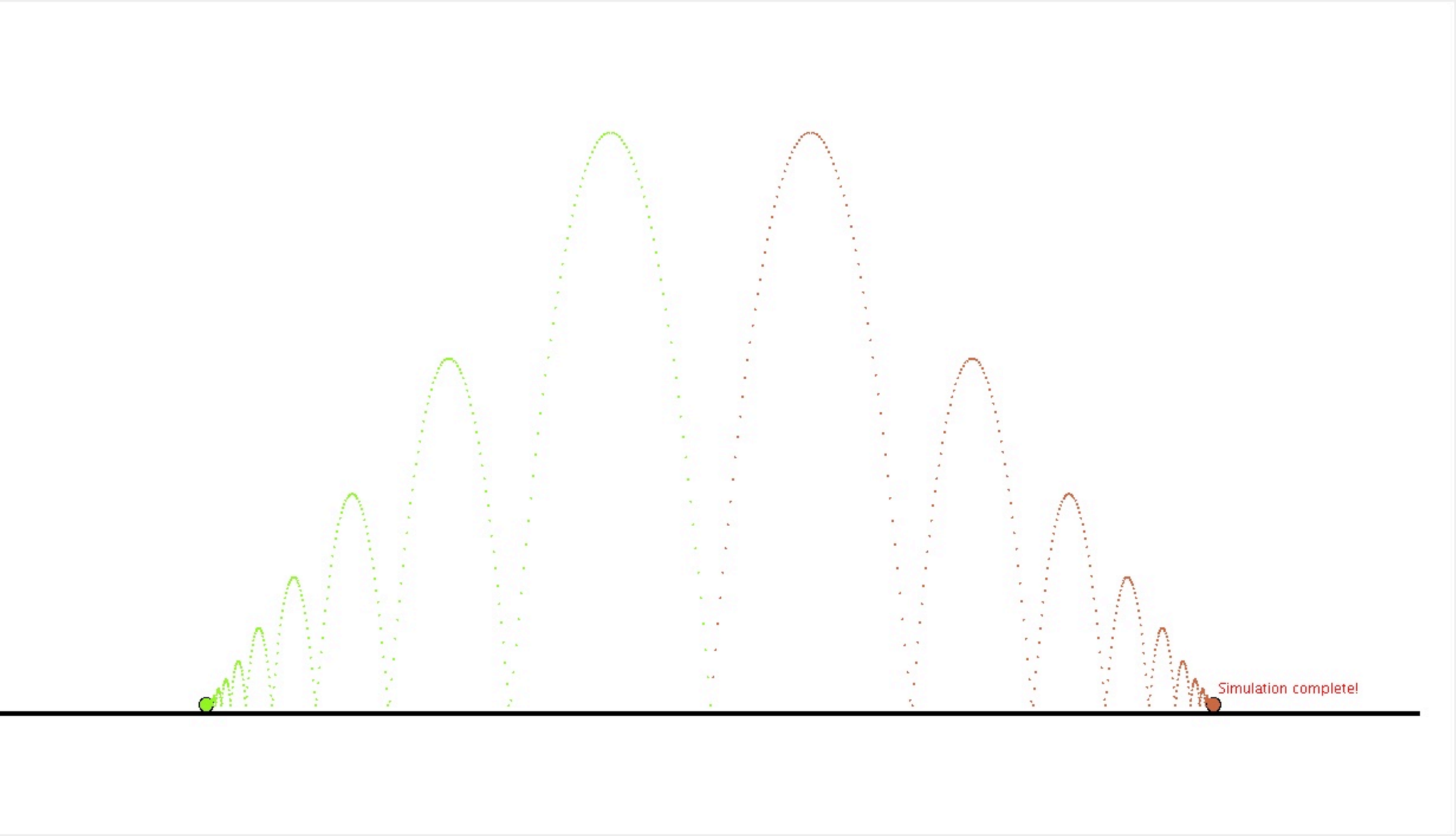
General Simulation Parameters

NUMBALLS:	1	255		1
MIN SIZE:	1.0	25.0		1.0
MAX SIZE:	1.0	25.0		1.0
LOSS MIN:	0.0	1.0		0.4
LOSS MAX:	0.0	1.0		0.4
MIN VEL:	1.0	200.0		40.0
MAX VEL:	1.0	200.0		40.0
TH MIN:	1.0	180.0		85.0
TH MAX:	1.0	180.0		85.0

Trace

bSim

bSim File Edit Help



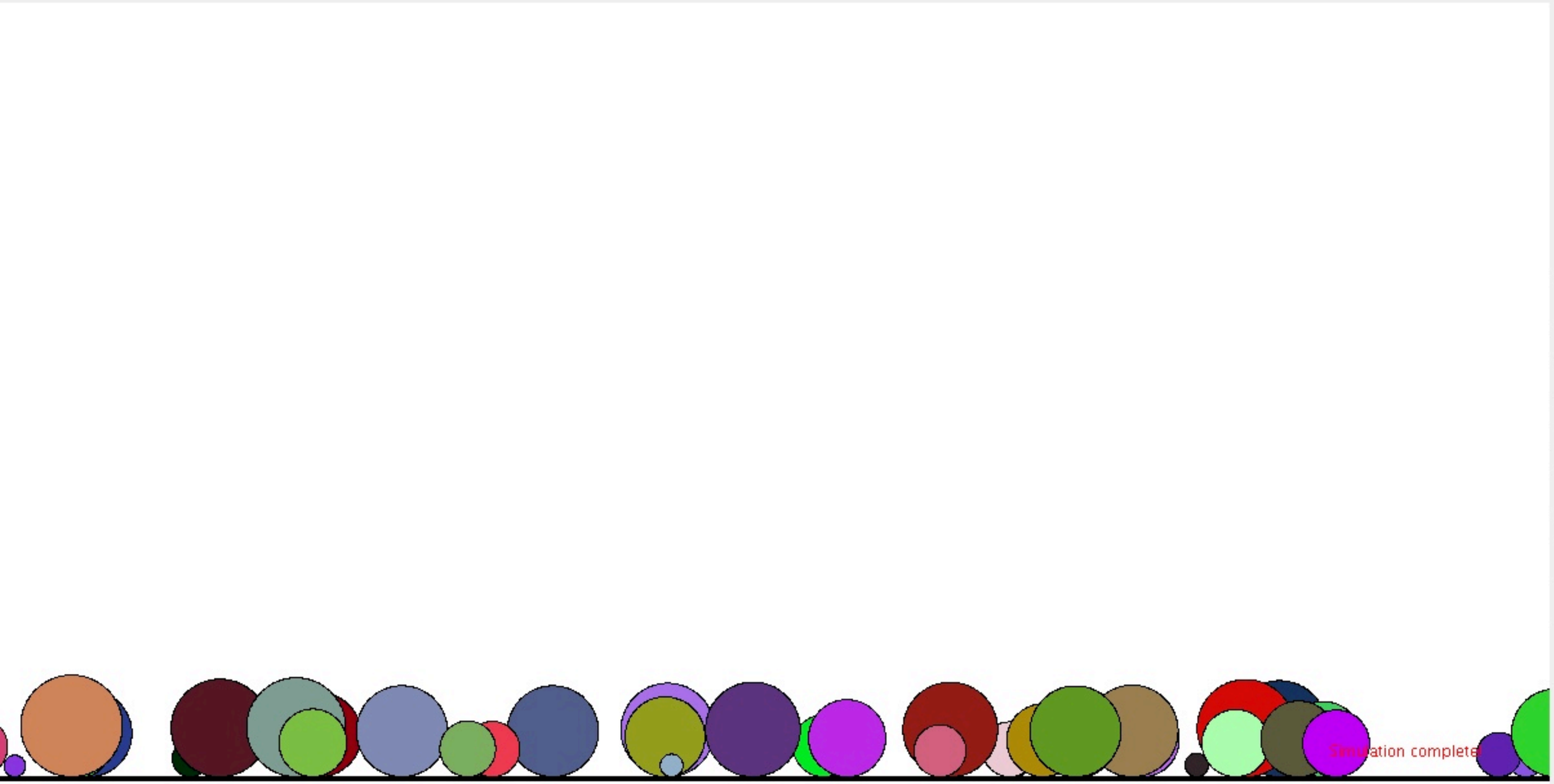
General Simulation Parameters

NUMBALLS:	1	255	1
MIN SIZE:	1.0	25.0	1.0
MAX SIZE:	1.0	25.0	1.0
LOSS MIN:	0.0	1.0	0.4
LOSS MAX:	0.0	1.0	0.4
MIN VEL:	1.0	200.0	40.0
MAX VEL:	1.0	200.0	40.0
TH MIN:	1.0	180.0	95.0
TH MAX:	1.0	180.0	95.0

Trace

bSim

bSim File Edit Help



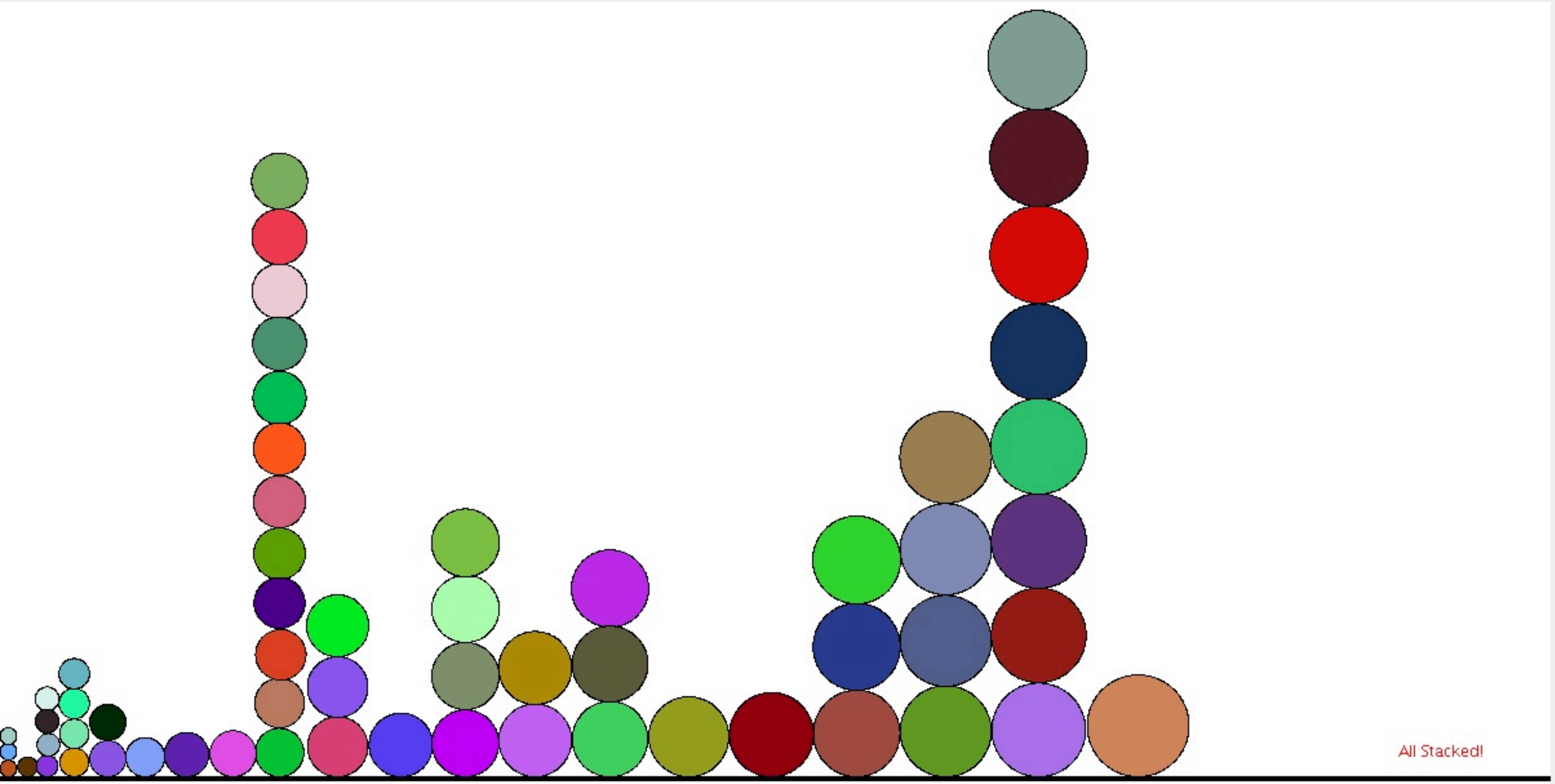
General Simulation Parameters

NUMBALLS:	1	255	60
MIN SIZE:	1.0	25.0	1.0
MAX SIZE:	1.0	25.0	7.0
LOSS MIN:	0.0	1.0	0.2
LOSS MAX:	0.0	1.0	0.6
MIN VEL:	1.0	200.0	40.0
MAX VEL:	1.0	200.0	50.0
TH MIN:	1.0	180.0	80.0
TH MAX:	1.0	180.0	100.0

Trace

bSim

Stack File Edit Help



General Simulation Parameters

NUMBALLS:	1	255	60
MIN SIZE:	1.0	25.0	1.0
MAX SIZE:	1.0	25.0	7.0
LOSS MIN:	0.0	1.0	0.2
LOSS MAX:	0.0	1.0	0.6
MIN VEL:	1.0	200.0	40.0
MAX VEL:	1.0	200.0	50.0
TH MIN:	1.0	180.0	80.0
TH MAX:	1.0	180.0	100.0

Trace