WCSS 2008 MASON Tutorial

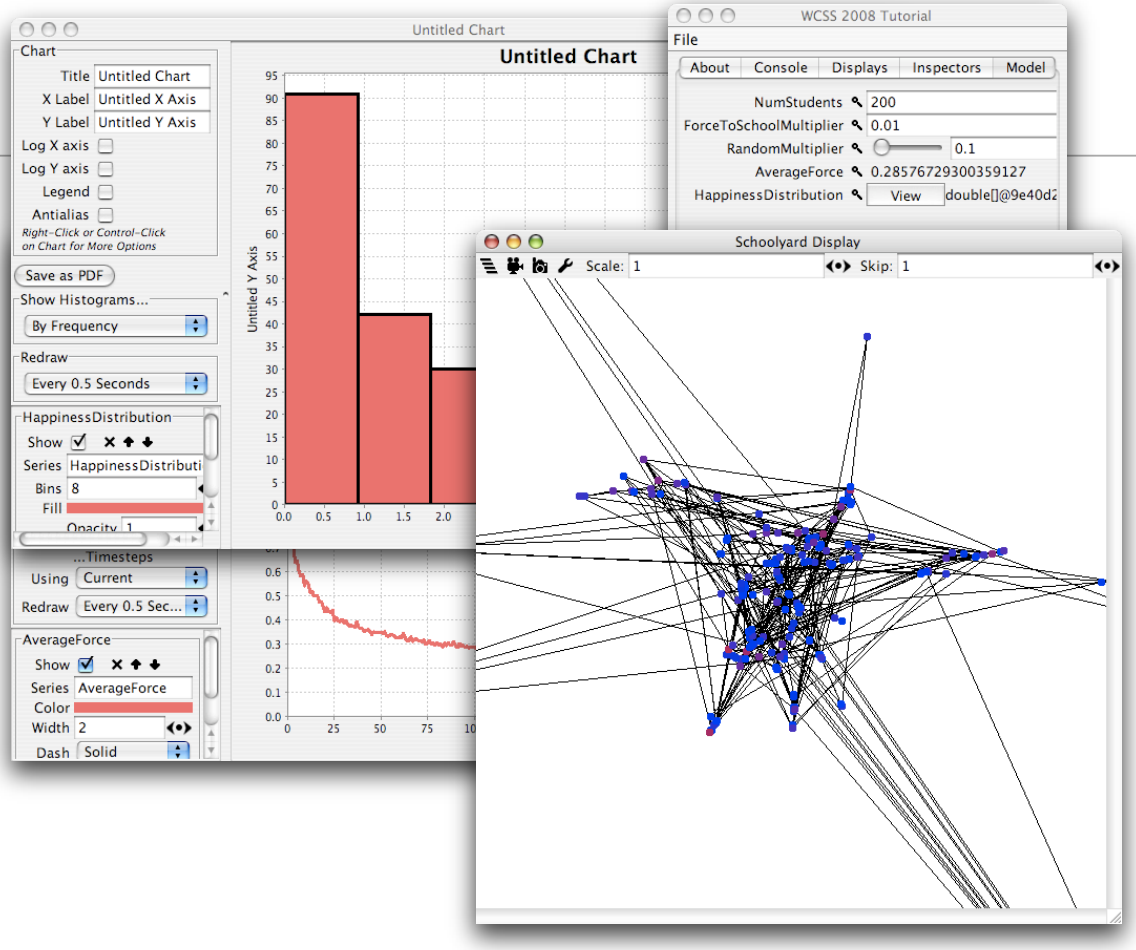
Sean Luke
George Mason University

http://cs.gmu.edu/~eclab/projects/mason/

MASON is a joint production of
the GMU Department of Computer Science and
the GMU Center for Social Complexity

The Example

- Student cliques formed on the schoolyard.



- Using:
  - Students as Agents
  - Embedded in continuous, unbounded 2D space
  - Connected via weighted networks of mutual friends / enemies

The Tutorial

- Get MASON Running
- Create a Minimum Simulation
- Add a Field and some Agents in the Field which do nothing
- Make the Agents do something
- Fire up the GUI
- Portray the Field
- Add a Network Field
- Get the Agents to do things based on the Network
- Portray the Network
- Make the Agents inspectable and color them according to happiness
- Make the model inspectable and add Histogram and Time Series charts

The Point of MASON

- MASON was built to do many simultaneous simulations of many agents on cluster computers, while visualizing the results on front-end machines.
- We began work on MASON because no other multiagent simulator met our design constraints:
  - Fast but replicable
  - Sporting a small and cleanly designed model core
  - Able to serialize and migrate models from machine to machine (and platform to platform!) and keep them separate from visualization cruft.
  - Sporting good quality parallelization
  - Entirely self-contained (so it could be used as a subelement of another program, or have two simulations run in parallel in separate threads)
  - Sporting good quality, reasonably fast 2D and 3D visualization

Step 0: Set up MASON

- Install Java3D
- The **mason** directory and each of the **jar** files in the "jar" directory must be added to your CLASSPATH
- The WCSS tutorial code is in **mason/sim/app/wcss**
- The WCSS tutorial documentation is in **tutorialDocs/tutorial**

- To compile an example, copy its code **out of its numbered subdirectory** and **into the wcss** directory, then compile the java files. **Example:**
  1. delete the **java** and **class** files directly in **mason/sim/app/wcss**
  2. copy the files from **mason/sim/app/wcss/8** into **mason/sim/app/wcss**
  3. compile the new **java** files in **mason/sim/app/wcss**

Students.java

Step 1: Create a Minimum Simulation

- Create a directory in MASON called **mason/sim/app/wcss/**
- In that directory, create a file called **Students.java**
- Enter the text at right into the file
- Compile the file with **javac Students.java**
- Run the simulation with **java sim.app.wcss.Students**

- *(All the examples for each step are already in a subdirectory — you can just pull them out to follow along).*

```
package sim.app.wcss;
import sim.engine.*;

public class Students extends SimState
    {
    public Students(long seed)
        {
        super(seed);
        }

    public void start()
        {
        super.start();
        }

    public static void main(String[] args)
        {
        doLoop(Students.class, args);
        System.exit(0);
        }
    }
```

Step 1: Create a Minimum Simulation

- What this means.
- **SimState** is the abstract class for models. Your model subclasses it.
- The standard SimState constructor requires a random number genrator **seed**
- **start** is the method called when the simulation starts but before any agents have been pulsed.
- You can make your own main loop, which calls start(), then pulses the schedule some *n* times, then calls finish(). But the provided **doLoop** function does it with style.

```
package sim.app.wcss;
import sim.engine.*;

public class Students extends SimState
    {
    public Students(long seed)
        {
        super(seed);
        }

    public void start()
        {
        super.start();
        }

    public static void main(String[] args)
        {
        doLoop(Students.class, args);
        System.exit(0);
        }
    }
```

Output

- MASON fires up but since there's nothing for the simulation, it quits again.

```
MASON Version 12.  For further options, try adding ' -help' at end.
Job: 0 Seed: 1215962773036
Starting sim.app.wcss.Students
Exhausted
```

Student.java

Step 2: Add a Field and some Agents

- We'll start by defining the Agents. They'll do nothing for the time being. In fact they're nothing more than empty classes.
- In the same directory, create a file called **Student.java**
- Add the following text to the file.

```
package sim.app.wcss;

public class Student
    {
    }
```

Students.java

Step 2: Add a Field and some Agents

- *Modify the **Students.java** file*
- Add the following new imports:

```
import sim.util.*;
import sim.field.continuous.*;
```

- Add two new instance variables:

```
public Continuous2D yard = new Continuous2D(1.0,100,100);

public int numStudents = 50;
```

Step 2: Add a Field and some Agents

- What this means.

```
public Continuous2D yard = new Continuous2D(1.0,100,100);
public int numStudents = 50;
```

- **Continuous2D** is a **Field**: a representation of space. In particular, Continuous2D represents continuous 2-dimensional space. It takes three constructor arguments:
  - The discretization level (for neighborhood lookup efficiency; we won't bother with this)
  - The width
  - The height
- Continuous2D is actually infinite: the width and height are just for GUI guidelines. We set it to 100x100, which is a good schoolyard size.

Students.java

Step 2: Add a Field and some Agents

- Revise the start() method:

```
public void start()
    {
    super.start();

    // clear the yard
      yard.clear();

    // add some students to the yard
    for(int i = 0; i < numStudents; i++)
        {
        Student student = new Student();
        yard.setObjectLocation(student,
            new Double2D(yard.getWidth() * 0.5 + random.nextDouble() - 0.5,
                         yard.getHeight() * 0.5 + random.nextDouble() - 0.5));
        }
}
```

- Compile (**javac *.java**) and run (**java sim.app.wcss.Students**)

Step 2: Add a Field and some Agents

- What this means.

```
yard.setObjectLocation(student,
    new Double2D(yard.getWidth() * 0.5 + random.nextDouble() - 0.5,
                 yard.getHeight() * 0.5 + random.nextDouble() - 0.5));
```

- **random** is a **MersenneTwisterFast** random number generator.
- **nextDouble** returns a value between 0.0 and 1.0.
- **setObjectLocation** takes an object and puts it in the field at the given location.
- For Continuous2Ds, locations are **Double2D** objects. These are immutable instances holding an **x** and **y** value. They're similar to **java.awt.Point2D.Double,** but they're not modifiable once created.

Output

- Still boring! Note that the random number seed has changed due to the new start time.
- By the way, if you'd like to fix the seed (to 1000 say), try **java sim.app.wcss.Students -seed 1000**

```
MASON Version 12.  For further options, try adding ' -help' at end.
Job: 0 Seed: 1215964716844
Starting sim.app.wcss.Students
Exhausted
```

Students.java

Step 3: Make the Agents Do Something

- The students wander about randomly a tiny bit but not stray from the center of the schoolyard too far. We'll add two instance variables in **Students.java:**

```
double forceToSchoolMultiplier = 0.01;
double randomMultiplier = 0.1;
```

- Now let's add the students to our schedule. Each one will get get "stepped" (pulsed, fired, whatever) repeatedly once every timestep. We add the following line to the start() method:

```
// add some students to the yard
for(int i = 0; i < numStudents; i++)
    {
    Student student = new Student();
    yard.setObjectLocation(student,
        new Double2D(yard.getWidth() * 0.5 + random.nextDouble() - 0.5,
                     yard.getHeight() * 0.5 + random.nextDouble() - 0.5));

    schedule.scheduleRepeating(student);
    }
```

Step 3: Make the Agents Do Something

- What this means.

```
schedule.scheduleRepeating(student);
```

- **schedule** is the simulator's **Schedule**, a representation of time. You schedule **Steppable** objects on the Schedule, to be **stepped** (fired, pulsed) at various times.
- This version of **scheduleRepeating** schedules a Steppable to be repeatedly stepped, forever, once per unit timestep, starting at the current time.

Student.java

Step 3: Make the Agents Do Something

- Now that the Students have been added to our schedule to be repeated once every timestep, we need to make them actually *capable* of responding to a pulsing call from the schedule. To do this we will make them implement the **Steppable** interface. Then we'll have the agents move themselves around the yard when stepped. First add the following imports to **Student.java:**

```
import sim.engine.*;
import sim.field.continuous.*;
import sim.util.*;
```

- Now we will declare Students to be **Steppable:**

```
public class Student implements Steppable
```

Student.java

Step 3: Make the Agents Do Something

- The **Steppable** interface requires the method **Step**, which will be called by the Schedule. Our agents will move randomly but not too far from the school.

```
public void step(SimState state)
    {
    Students students = (Students) state;
    Continuous2D yard = students.yard;
    Double2D me = students.yard.getObjectLocation(this);
    MutableDouble2D sumForces = new MutableDouble2D();

    // add in a vector to the center of the yard, so we don't go too far away
    sumForces.addIn(new Double2D(
      (yard.width * 0.5 - me.x) * students.forceToSchoolMultiplier,
      (yard.height * 0.5 - me.y) * students.forceToSchoolMultiplier));

    // add a bit of randomness
    sumForces.addIn(new Double2D(
      students.randomMultiplier * (students.random.nextDouble() * 1.0 - 0.5),
      students.randomMultiplier * (students.random.nextDouble() * 1.0 - 0.5)));

    sumForces.addIn(me);
    students.yard.setObjectLocation(this, new Double2D(sumForces));
    }
```

Step 3: Make the Agents Do Something

- What this means.

```
Double2D me = students.yard.getObjectLocation(this);
MutableDouble2D sumForces = new MutableDouble2D();
```

- **getObjectLocation** returns the current location of an object in a field.
- **MutableDouble2D** is a modifiable version of **Double2D**. Essentially it's the same concept as **java.awt.Point2D.Double,** but with more functionality.

```
sumForces.addIn(new Double2D(
  (yard.width * 0.5 - me.x) * students.forceToSchoolMultiplier,
  (yard.height * 0.5 - me.y) * students.forceToSchoolMultiplier));
```

- **addIn** adds a MutableDouble2D to a Double2D or similar object, and sets the MutableDouble2D to the sum.

```
sumForces.addIn(me);
```

- Here we're taking the force vector and adding in our current position, resulting in a new position for us to assume.

Output

- Now are agents are doing something! Nothing particularly complicated though, since our frame rate is 22000 ticks per second!
- Press Control-c to cancel the simulation.
- By the way, if you'd like to run the simulation until timestep 100000 has completed, you can say
  java sim.app.wcss.Students -until 100000

```
MASON Version 12.  For further options, try adding ' -help' at end.
Job: 0 Seed: 1215966150794
Starting sim.app.wcss.Students
Steps: 25000 Time: 24999 Rate: 20,833.33333
Steps: 50000 Time: 49999 Rate: 21,626.29758
Steps: 75000 Time: 74999 Rate: 22,026.43172
Steps: 100000 Time: 99999 Rate: 22,104.33245
Steps: 125000 Time: 124999 Rate: 22,123.89381
Steps: 150000 Time: 149999 Rate: 22,026.43172
Steps: 175000 Time: 174999 Rate: 22,104.33245
...
```

StudentsWithUI.java

Step 4: Fire up the GUI

- Create a file called **StudentsWithUI.java** This class is responsible for handling the GUI for our simulation. Initially it will just be in charge of starting and stopping the schedule; but later we'll make it in charge of displaying the fields as well. Add:

```
package sim.app.wcss;
import sim.display.*;
import sim.engine.*;
import javax.swing.*;

public class StudentsWithUI extends GUIState
    {
    public static void main(String[] args)
        {
        Console c = new Console(new StudentsWithUI(););
        c.setVisible(true);
        }

    public StudentsWithUI() { super(new Students( System.currentTimeMillis())); }
    public StudentsWithUI(SimState state) { super(state); }
    public static String getName() { return "WCSS 2008 Tutorial"; }
  }
```

Step 4: Fire up the GUI

- What this means.

```
public class StudentsWithUI extends GUIState
```

- GUIState is the abstract superclass for the primary object for building and managing the GUI. It holds a **simulation** (your SimState) and a **Controller** which manages the schedule.

```
public static void main(String[] args)
    {
    Console c = new Console(new StudentsWithUI(););
    c.setVisible(true);
    }
```

- **Console** is an elaborate GUI Controller. It needs the GUIState as a parameter. This is the standard way of starting the UI.

Step 4: Fire up the GUI

- What this means (part 2).

```
public StudentsWithUI() { super(new Students( System.currentTimeMillis())); }
public StudentsWithUI(SimState state) { super(state); }
```

- One standard constructor for a GUIState takes a SimState.
- The default constructor creates the SimState on the fly, using the current timestamp milliseconds as the random number generator seed.

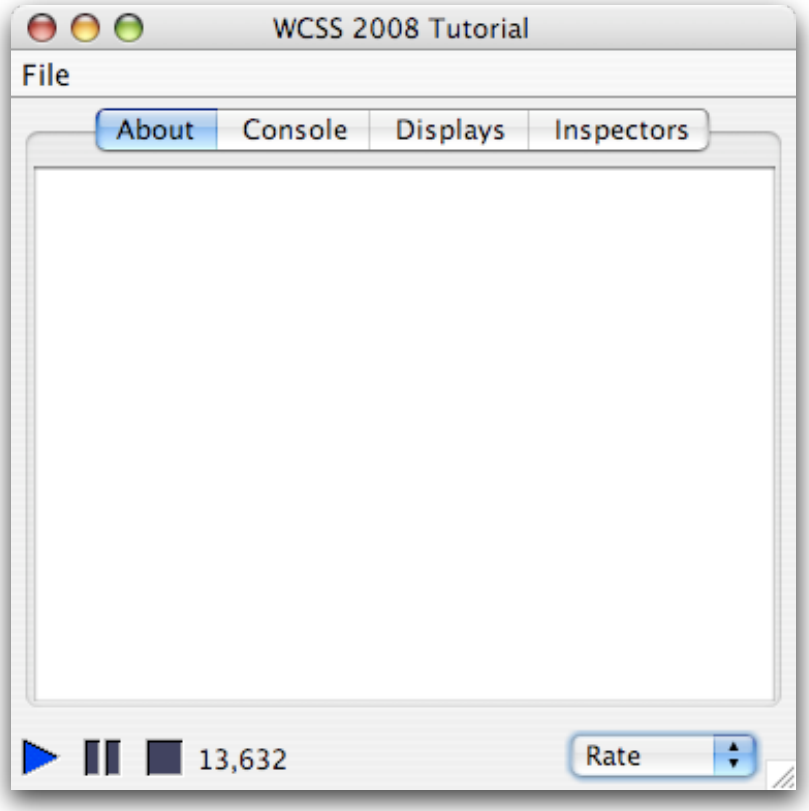
```
public static String getName() { return "WCSS 2008 Tutorial"; }
```

- GUIStates have names. They appear as the title bar of the Console.

Output

- Run as
  java sim.app.wcss.StudentsWithUI
- Press the Play button
  (the dark triangle)
- Change the **Time** pop-up menu to
  Rate to see the frame rate

- 13,000 is slower than 22,000. This is mostly due to the overhead of managing the GUI event loop.



StudentsWithUI.java

Step 5: Portray the Field

- Add some new imports

```
import sim.portrayal.continuous.*;
import sim.portrayal.simple.*;
import java.awt.Color;
```

- Add instance variables for a Portrayal (which draws the field), a Display (the GUI component which houses Portrayals), and a JFrame (the window which houses the Display).

```
public Display2D display;
public JFrame displayFrame;
ContinuousPortrayal2D yardPortrayal = new ContinuousPortrayal2D();
```

Step 5: Portray the Field

- What this means.

```
public Display2D display;
public JFrame displayFrame;
ContinuousPortrayal2D yardPortrayal = new ContinuousPortrayal2D();
```

- **Field Portrayals** are classes responsible for drawing fields and letting the user manipulate objects stored within them. **ContinuousPortrayal2D** is the default portrayal for the **Continuous2D** field.

- A **Display2D** is a GUI widget which holds some number of Field Portrayals, usually layered on top of one another.

- The **JFrame** will be the window which holds the Display2D. Usually Display2Ds provide their own JFrames.

StudentsWithUI.java

Step 5: Portray the Field

- Add an **init** method which creates the display, has the display generate a frame for you, registers the frame with the Console, and attaches the portrayal to the display. This method is called when the GUI is first fired up.

```
public void init(Controller c)
    {
    super.init(c);

    // make the displayer
    display = new Display2D(600,600,this,1);
    // turn off clipping
    display.setClipping(false);

    displayFrame = display.createFrame();
    displayFrame.setTitle("Schoolyard Display");

    // register the frame so it appears in the "Display" list
    c.registerFrame(displayFrame);
    displayFrame.setVisible(true);
    display.attach( yardPortrayal, "Yard" );
    }
```

Step 5: Portray the Field

- What this means.
- A **Display2D** constructor takes a width, a height, a GUIState, and timesteps between redraws
- Ordinarily the Display2D clips the drawing of its fields when they're zoomed out. But as our Continuous2D is infinite, we'd like to see objects outside the width x height box. We **setClipping** to false.

```
display = new Display2D(600,600,this,1);
display.setClipping(false);

displayFrame = display.createFrame();
displayFrame.setTitle("Schoolyard Display");

c.registerFrame(displayFrame);
displayFrame.setVisible(true);
display.attach( yardPortrayal, "Yard" );
```

- Displays can sprout their own JFrames (windows) for us using **createFrame.**
- We **register** the JFrame with the Console to include it in the Console's list. This enables us to hide the JFrame and get it back via the Console. It also lets the Console cleanly dispose of the window when we quit.
- We **attach** the Portrayal to the Display2D to display it inside the Display2D.

StudentsWithUI.java

Step 5: Portray the Field

- Add an **quit** method which destroys the display and frame. This method is called when the program quits.

```
public void quit()
    {
    super.quit();

    if (displayFrame!=null) displayFrame.dispose();
    displayFrame = null;
    display = null;
    }
```

- Add default **start()** and **load()** methods. These methods are called when the play button is pressed or when a simulation is loaded from a checkpoint file. Typically they do the same thing, so we'll have them call a method called **setupPortrayals** which we'll write next:

```
public void start()
    {
    super.start();
    setupPortrayals();
    }
```

```
public void load(SimState state)
    {
    super.load(state);
    setupPortrayals();
    }
```

StudentsWithUI.java

Step 5: Portray the Field

- Add our **setupPortrayals** method. Here we will tell the Portrayal which field it is portraying and how it's doing it. We'll also reset the display so it re-registers itself with the console in preparation for being stepped each timestep.

```
public void setupPortrayals()
    {
    Students students = (Students) state;

    // tell the portrayals what to portray and how to portray them
    yardPortrayal.setField( students.yard );
    yardPortrayal.setPortrayalForAll(new OvalPortrayal2D());

    // reschedule the displayer
    display.reset();
    display.setBackdrop(Color.white);

    // redraw the display
    display.repaint();
    }
```

Step 5: Portray the Field

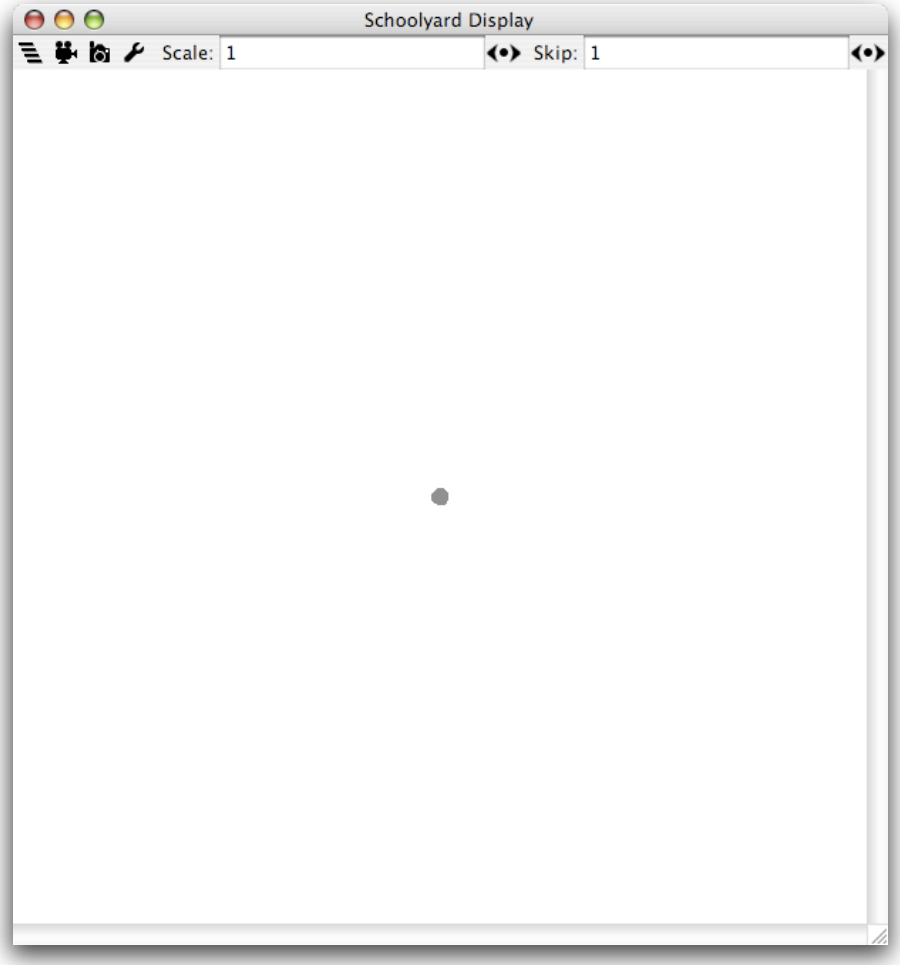
- What this means.

```
yardPortrayal.setField( students.yard );
yardPortrayal.setPortrayalForAll(new OvalPortrayal2D());

display.reset();
display.setBackdrop(Color.white);

display.repaint();
```

- We first tell the portrayal which field it's portraying.
- Most FieldPortrayals rely on underlying **SimplePortrayal**s to draw the individual elements in the field. Here we're stating that all objects in the field are to be drawn with **OvalPortrayal2D.**
- When you **reset** a display, it registers itself with the GUI system to be pulsed every timestep. We need to do that.
- The **backdrop** is the background of the displayed region.

Output

- Run again, and press Play.
- Note that now there's a display popped up which shows our schoolyard. All the students are clustered in the center of the schoolyard, moving around slightly randomly.
- Very exciting.
- The frame rate has dropped to about 270. But if you close the display window, it's back up to about 13,000.
- You can get the display window back by choosing **Displays**, then **Show All.**



Students.java

Step 6: Add a Network Field

- We'll add a Network of friendship among the students. It'll be an undirected weighted graph. Positive weights indicate friendship, negative weights indicate dislike. First we add a new import:

```
import sim.field.network.*;
```

- Next we create a new instance variable representing the Network. **false** indicates that the Network graph structure is undirected.

```
public Network buddies = new Network(false);
```

Step 6: Add a Network Field

- Change the **start** method to clear out the network and add the students to it as graph nodes:

```
// clear the yard
yard.clear();

// clear the buddies
buddies.clear();

// add some students to the yard
for(int i = 0; i < numStudents; i++)
     {
     Student student = new Student();
     yard.setObjectLocation(student,
         new Double2D(yard.getWidth() * 0.5 + random.nextDouble() - 0.5,
                      yard.getHeight() * 0.5 + random.nextDouble() - 0.5));
     buddies.addNode(student);
     schedule.scheduleRepeating(student);
     }
```

Students.java

Step 6: Add a Network Field

- At the end of the **start** method, add some code to allow each student to have at least one mutual like and one mutual dislike of another student (some students will get more than one).

```
Bag students = buddies.getAllNodes();
for(int i = 0; i < students.numObjs; i++)
    {
    Object student = students.objs[i];

    Object studentB = null;
    do              // who does he like?
        {
        studentB = students.objs[random.nextInt(students.numObjs)];
        } while (student == studentB);
    double buddiness = random.nextDouble();
    buddies.addEdge(student, studentB, new Double(buddiness));

    do              // who does he dislike?
        {
        studentB = students.objs[random.nextInt(students.numObjs)];
        } while (student == studentB);
    buddies.addEdge(student, studentB, new Double( -buddiness));
    }
```

Step 6: Add a Network Field

- What this means.

```
Bag students = buddies.getAllNodes();
```

- This gets all the nodes in the network and returns it as a **Bag**. This function presumes you will not modify this Bag — the network relies on it.

```
for(int i = 0; i < students.numObjs; i++)
    {
    Object student = students.objs[i];
```

- A **Bag** is the same basic structure as an **ArrayList** or **Vector.** The difference is that the Bag's underlying array is public. It's two variables: **objs** is the array and **numObjs** is the number of elements in the array (which is often less than the array size). Bag is much faster than ArrayList.

```
buddies.addEdge(student, studentB, new Double(buddiness));
```

- We add an edge from one student to another in the graph. It's undirected. The edge is weighted with a **Double** representing friendship or hatred.

Output

- Nothing new. The students aren't taking advantage of their relationships yet.

Student.java

Step 7: Have the Students Use the Network

- The students will move away from their enemies and towards their friends. In **Student.java**, add this import:

```
import sim.field.network.*;
```

- Now we need to limit the amount of force the student hatred will impart, or else they could wander off the yard. Add the following instance variable:

```
public static final double MAX_FORCE = 3.0;
```

Student.java

Step 7: Have the Students Use the Network

- We modify the **step** method to add some new forces. Here's the first half.

```
public void step(SimState state)
   {
   Students students = (Students) state;
   Continuous2D yard = students.yard;
   Double2D me = students.yard.getObjectLocation(this);
   MutableDouble2D sumForces = new MutableDouble2D();

  // Go through my buddies and determine how much I want to be near them
   MutableDouble2D forceVector = new MutableDouble2D();
   Bag out = students.buddies.getEdges(this, null);
   for(int buddy = 0 ; buddy < out.numObjs; buddy++)
       {
       Edge e = (Edge)(out.objs[buddy]);
       double buddiness = ((Double)(e.info)).doubleValue();

      // I could be in the to() end or the from() end.getOtherNode is a
      //cute function which grabs the guy at the opposite end from me.
       Double2D him = students.yard.getObjectLocation(e.getOtherNode(this));
```

Student.java

Step 7: Have the Students Use the Network

- Here is the back half, where the buddy force vectors are calculated.

```
    if (buddiness >= 0)  // the further I am from him the more I want to go to him
        {
        forceVector.setTo((him.x - me.x) * buddiness, (him.y - me.y) * buddiness);
        if (forceVector.length() > MAX_FORCE)  // I'm far enough away
            forceVector.setLength(MAX_FORCE);
        }
    else  // the nearer I am to him the more I want to get away from him, up to a limit
        {
        forceVector.setTo((him.x - me.x) * buddiness, (him.y - me.y) * buddiness);
        if (forceVector.length() > MAX_FORCE)  // I'm far enough away
            forceVector.setLength(0.0);
        else if (forceVector.length() > 0)
            forceVector.setLength(MAX_FORCE - forceVector.length());  // invert the distance
        }
    sumForces.addIn(forceVector);
    }

// add in a vector to the center of the yard, so we don't go too far away
sumForces.addIn(new Double2D(  ... and so on....
```

Step 7: Have the Students Use the Network

- What this means.

```
Bag out = students.buddies.getEdges(this, null);
```

- Since the network is undirected, we can get all edges attached to a node with **getEdges**, which takes the node in question and an optional Bag (in this case **null**), which it will use. It then returns a Bag. You can modify this Bag, it's not used internally.

```
Edge e = (Edge)(out.objs[buddy]);
double buddiness = ((Double)(e.info)).doubleValue();
```

- Edges have three values: **to, from,** and **info** (the weight or label). We extract the weight here.

```
Double2D him = students.yard.getObjectLocation(e.getOtherNode(this));
```

- Since the edge is undirected, we don't know, without looking, if we were in the **to** or **from** position. No big deal: just use the **getOtherNode** method to return our partner on the other side of the edge.

Step 7: Have the Students Use the Network

- What this means.

```
forceVector.setTo((him.x - me.x) * buddiness, (him.y - me.y) * buddiness);
```

- MutableDouble2D can be **setTo** new x and y values. (We could have also said forceVector.x = ... and forceVector.y = ...)

```
if (forceVector.length() > MAX_FORCE)  // I'm far enough away
     forceVector.setLength(MAX_FORCE);
```

- **length** is the sum squared magnitude MutableDouble2D. You can change the length but keep the angle using **setLength** as long as the previous length wasn't 0 or infinity.

Output

- Run again, and press Play.
- Now we've got something! The students are grouping into cliques.
- But it'd sure be nice to see the relationships among them...

StudentsWithUI.java

Step 8: Portray the Network

- It's simple to overlay the network on Continuous region. In **StudentsWithUI,** we start by adding a new import:

```
import sim.portrayal.network.*;
```

- Now we'll add a new instance variable for the Portrayal for our network:

```
NetworkPortrayal2D buddiesPortrayal = new NetworkPortrayal2D();
```

- In the **init** method, attach this portrayal *before* the Yard portrayal. This causes it to be drawn first (so the students are drawn nicely on top of their relationship lines).

```
displayFrame = display.createFrame();
displayFrame.setTitle("Schoolyard Display");
c.registerFrame(displayFrame);
displayFrame.setVisible(true);
display.attach( buddiesPortrayal, "Buddies" );
display.attach( yardPortrayal, "Yard" );
```

StudentsWithUI.java

Step 8: Portray the Network

- Now we just need to attach the Network to the portrayal. To do this we provide a SpatialNetwork2D object has both the node *locations* (students.yard) and the collection of edges (students.buddies)

```
public void setupPortrayals()
    {
    Students students = (Students) state;

    // tell the portrayals what to portray and how to portray them
    yardPortrayal.setField( students.yard );
    yardPortrayal.setPortrayalForAll(new OvalPortrayal2D());
    buddiesPortrayal.setField( new SpatialNetwork2D(
                                            students.yard, students.buddies ) );
    buddiesPortrayal.setPortrayalForAll(new SimpleEdgePortrayal2D());

    // reschedule the displayer
    display.reset();
    display.setBackdrop(Color.white);

    // redraw the display
    display.repaint();
    }
```

Step 8: Portray the Network

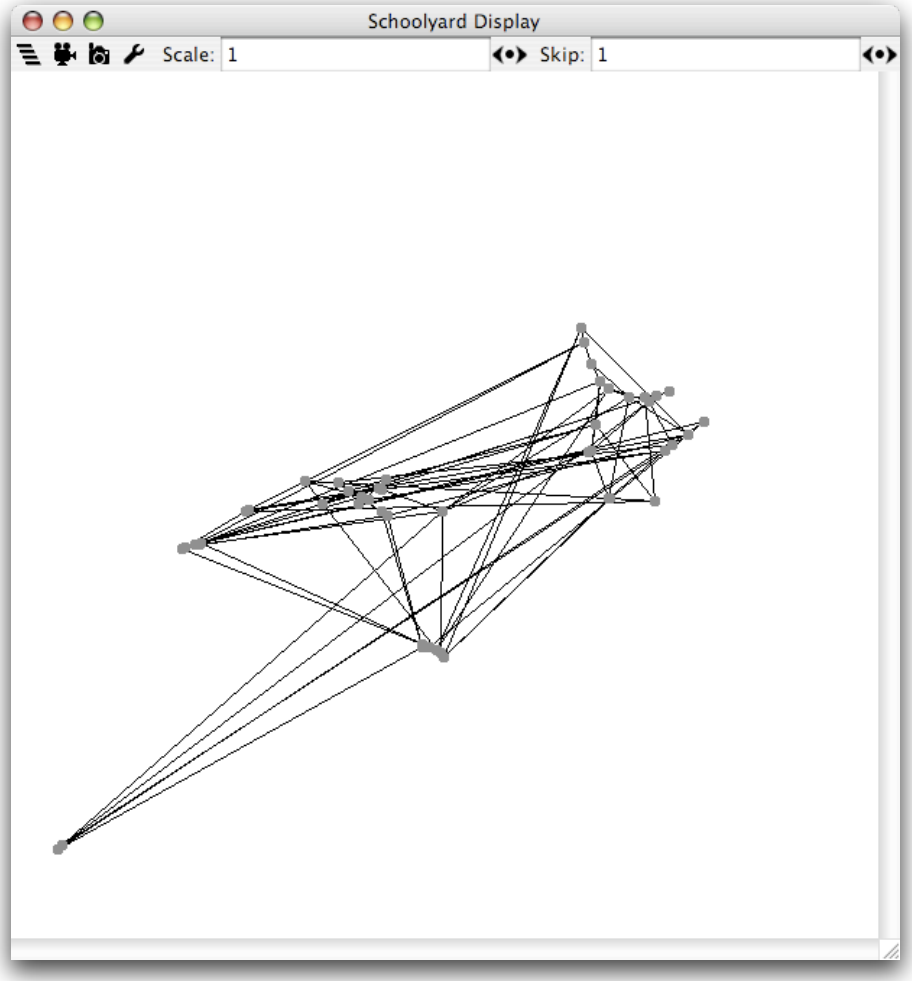
- What this means.

```
buddiesPortrayal.setField( new SpatialNetwork2D(
                                        students.yard, students.buddies ) );
buddiesPortrayal.setPortrayalForAll(new SimpleEdgePortrayal2D());
```

- A **NetworkPortrayal2D** needs *two* things to draw its edges. First, it needs to know where the nodes are located in space (using a **Continuous2D** or discrete **SparseGrid2D** field) so it can determine where to draw the edges. Second, it needs to know the Network so it can extract those edges and nodes. Since the **FieldPortrayal** interface only passed one item into **setField**, we pass in a special "Field" called a **SpatialNetwork2D** which holds onto these two fields for us. The NetworkPortrayal2D is expecting it.
- The **SimpleEdgePortrayal2D** is a trivial edge portrayal which draws edges as lines or as triangles, and can color them and label them in various ways. The default just uses unlabelled black lines.

Output

- Run again, and press Play.
- Woohoo! A network of students.



Student.java

Step 9: Inspect the Agents

- We'll begin by adding some instance variables to **Student.java** which hold statistics for each of our students: **force,** which is the average force exerted, and **friendsClose** and **enemiesCloser**, which way the average weighted distance to friends. These determine the **happiness** of the agent (lower values are more happy). **getHappiness()** and **getForce()** are read-property methods and will be recognized and displayed automatically by MASON.

```
double friendsClose = 0.0;  // initially very close to my friends
double enemiesCloser = 10.0;  // WAY too close to my enemies
public double getHappiness() { return friendsClose + enemiesCloser; }

public double force = 0.0;
public double getForce() { return force; }
```

- Reset these variables each **step**:

```
public void step(SimState state)
    {
    friendsClose = enemiesCloser = force = 0.0;
    Students students = (Students) state;
```

Student.java

Step 9: Inspect the Agents

- Now we collect the statistics when gathering the force vectors in **step**:

```
if (buddiness >= 0)  // the further I am from him the more I want to go to him
    {
    forceVector.setTo((him.x - me.x) * buddiness, (him.y - me.y) * buddiness);
    if (forceVector.length() > MAX_FORCE)  // I'm far enough away
        forceVector.setLength(MAX_FORCE);
    friendsClose += forceVector.length();
    }
else  // the nearer I am to him the more I want to get away from him, up to a limit
    {
    forceVector.setTo((him.x - me.x) * buddiness, (him.y - me.y) * buddiness);
    if (forceVector.length() > MAX_FORCE)  // I'm far enough away
        forceVector.setLength(0.0);
    else if (forceVector.length() > 0)
        forceVector.setLength(MAX_FORCE - forceVector.length());  // invert the distance
    enemiesCloser += forceVector.length();
    }
sumForces.addIn(forceVector);
force += forceVector.length();
}
```

StudentsWithUI.java

Step 9: Inspect the Agents

- Let's have the agents change color when they're not so happy. To do this we'll make a custom Portrayal for them. We start with some imports in **StudentsWithUI.java:**

```
import sim.portrayal.*;
import java.awt.*;
```

- Now we'll replace the OvalPortrayal2D with an **anonymous subclass** to change its paint color based on the underlying object's happiness:

```
yardPortrayal.setPortrayalForAll(new OvalPortrayal2D()
        {
        public void draw(Object object, Graphics2D graphics, DrawInfo2D info)
            {
            Student student = (Student)object;
            int happinessShade = (int) (student.getHappiness() * 255 / 10.0);
            if (happinessShade > 255) happinessShade = 255;
            paint = new Color(happinessShade, 0, 255 - happinessShade);
            super.draw(object, graphics, info);
            }
        });
```

Step 9: Inspect the Agents

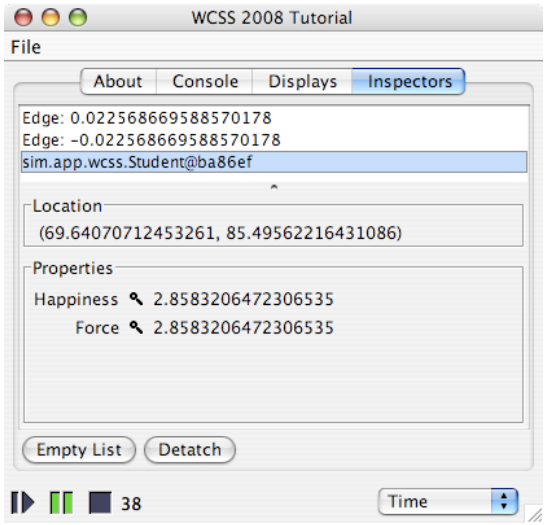
- What this means.

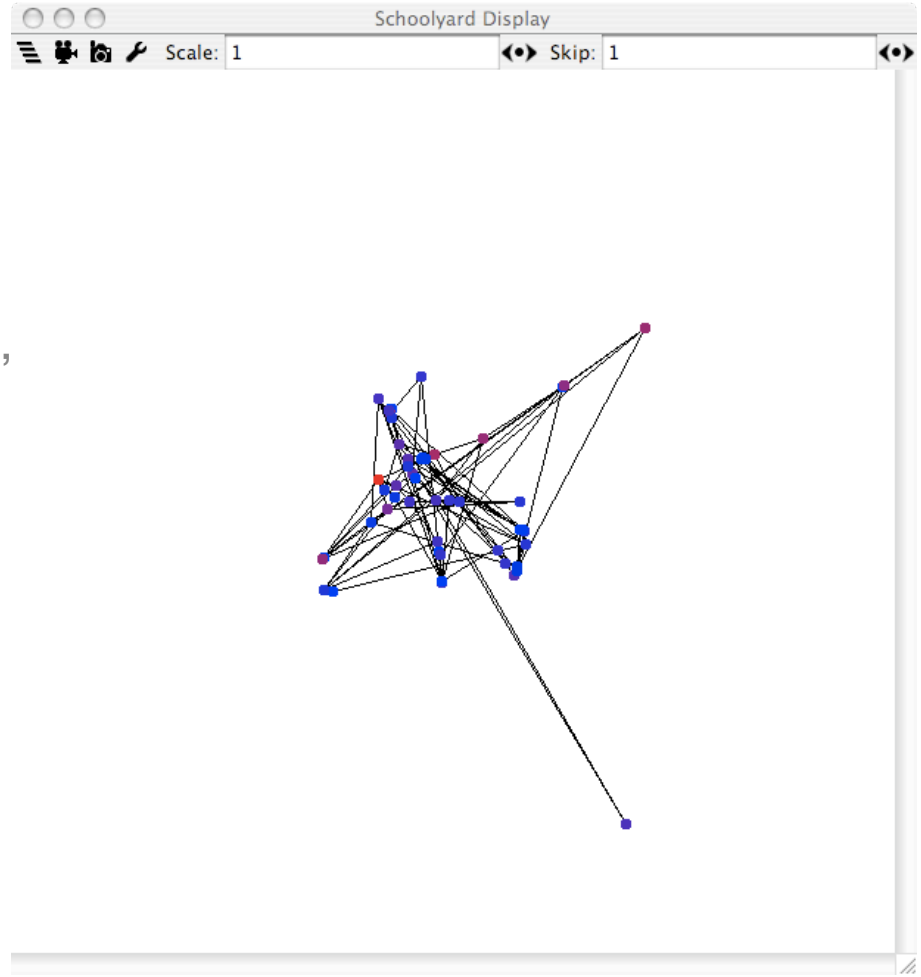
```
yardPortrayal.setPortrayalForAll(new OvalPortrayal2D()
        {
        public void draw(Object object, Graphics2D graphics, DrawInfo2D info)
            {
            Student student = (Student)object;
            int happinessShade = (int) (student.getHappiness() * 255 / 10.0);
            if (happinessShade > 255) happinessShade = 255;
            paint = new Color(happinessShade, 0, 255 - happinessShade);
            super.draw(object, graphics, info);
            }
        });
```

- **OvalPortrayal2D**, like all **Portrayals**, has a **draw** method which takes the object to draw (in this case, a Student), the **Graphics2D** object to draw it with, and a **DrawInfo2D** object which provides the (x,y) location, scale, and clipping rectangle (so we don't bother drawing if we're out of the clip)
- OvalPortrayal2D draws by setting the Graphics2D paint to its **paint** instance and then drawing a filled circle. We can change the color dynamically by changing the paint before drawing. We set it to our shade of happiness.

Output

- Run again, and press Play.
- Now the student are red when unhappy, blue when happy.
- Double-click on a student, and you can select it from the Inspectors list, and see its happiness and force.





StudentsWithUI.java

Step 10: Inspect the Model

- Let's finish up by providing a model inspector and some charts and graphs. We begin by declaring that the global model object to be inspected is the **Students** instance. In **StudentsWithUI.java** add the following method:

```
public Object getSimulationInspectedObject() { return state; }
```

- Ordinarily a model inspector is for knobs we tweak to manipulate model parameters. For that a *non-volatile* (non-constantly-updating) inspector is best. But in this example we'll be adding some properties which are constantly updated as the model progresses. So we need to make the inspector volatile. To do that we just override the **getInspector** method:

```
public Inspector getInspector()
    {
    Inspector i = super.getInspector();
    i.setVolatile(true);
    return i;
    }
```

Students.java

Step 10: Inspect the Model

- Next we need to add some inspectable properties in the **Students.java** file:

```
public int getNumStudents() { return numStudents; }
public void setNumStudents(int val) { if (val > 0) numStudents = val; }



public double getForceToSchoolMultiplier() { return forceToSchoolMultiplier; }
 public void setForceToSchoolMultiplier(double val)
     {
      if (forceToSchoolMultiplier >= 0.0) forceToSchoolMultiplier = val;
     }



 public double getRandomMultiplier() { return randomMultiplier; }
 public void setRandomMultiplier(double val)
     {
     if (randomMultiplier >= 0.0) randomMultiplier = val;
     }
 public Object domRandomMultiplier() { return new sim.util.Interval(0.0, 100.0); }
```

Step 10: Inspect the Model

- What this means.

```
public double getRandomMultiplier() { return randomMultiplier; }
public void setRandomMultiplier(double val)
    {
    if (randomMultiplier >= 0.0) randomMultiplier = val;
    }
public Object domRandomMultiplier() { return new sim.util.Interval(0.0, 100.0); }
```

- These are all read/write properties (they have both get*Foo* and set*Foo*), and MASON automatically recognizes them and displays them as editable.
- The **dom*Foo*** pattern (here **domRandomMultiplier**) is special to MASON: it allows us to specify the **domain** of a read/write property. Here we're stating that the property must be between 0.0 and 100.0. MASON responds by displaying not just an ordinary text field, but one with a slider.
- Fun fact: if you have an integer property, you can return a slider interval if you like; or instead return a domain that's an array of Strings, which MASON will convert to a pop-up menu list. The integer chosen is the index in the array!

Students.java

Step 10: Inspect the Model

- Let's add a read-only property that returns the average force over all students:

```
public double getAverageForce()
    {
    Bag students = buddies.getAllNodes();
    double force = 0.0;

    for(int i = 0; i < students.numObjs; i++)
        {
        force += ((Student)(students.objs[i])).getForce();
        }
    if (students.numObjs > 1) force /= students.numObjs;
    return force;
    }
```

Students.java

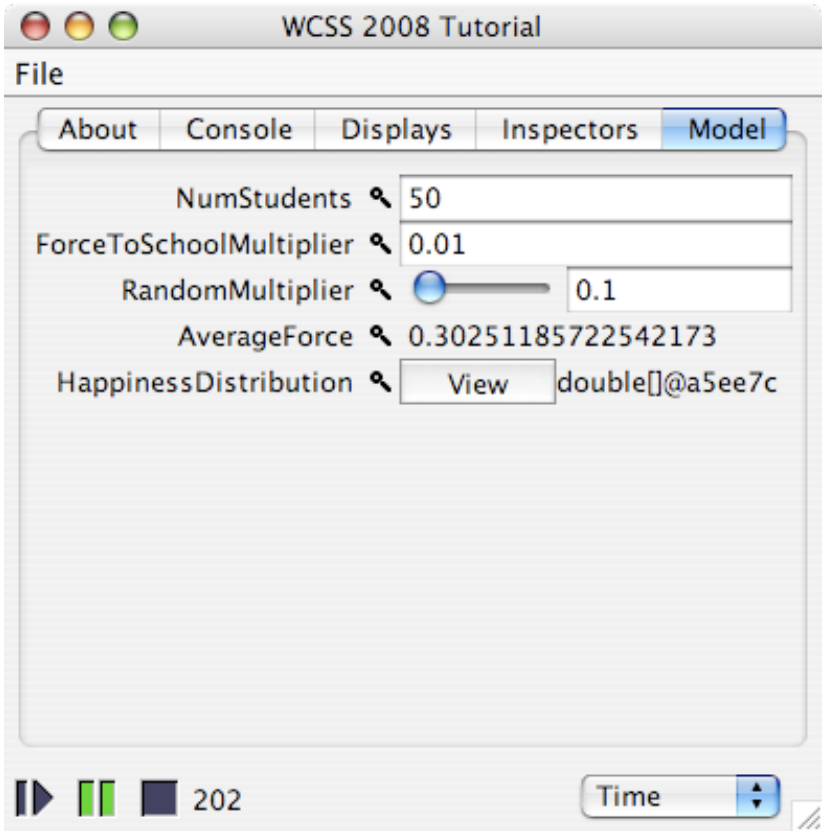
Step 10: Inspect the Model

- Last, let's add an unusual property which returns an *array* of all the happinesses of all the students. We can use this to create a histogram.

```
public double[] getHappinessDistribution()
    {
    Bag students = buddies.getAllNodes();
    double[] distro = new double[students.numObjs];

    for(int i = 0; i < students.numObjs; i++)
        distro[i] = ((Student)(students.objs[i])).getHappiness();
    return distro;
    }
```

Output

- A new tab (“Model”) has appeared in the Console. Click on it to see the various parameters included.
- Try changing the number of students and pressing Play.
- Try changing the RandomMultiplier during play.
- The magnifying glasses offer additional options. Try charting the average force, or making a histogram of the happiness distribution.
- Note that charts/histograms must be set up *after* you the simulation has begun play. You can do this by pressing pause, then setting them up, then unpausing. The charts and histograms presently only reflect the *current* simulation: after stopping, they’re invalid.



WCSS 2008 MASON Tutorial

Sean Luke
George Mason University

http://cs.gmu.edu/~eclab/projects/mason/

MASON is a joint production of
the GMU Department of Computer Science and
the GMU Center for Social Complexity