Multiagent Simulation And the MASON Library

Sean Luke
Department of Computer Science
George Mason University

Zeroth Edition
Online Version 0.1
January, 2011

Where to Obtain MASON
http://cs.gmu.edu/∼eclab/projects/mason/

Thanks to Claudio Cioffi Revilla and Carlotta Domeniconi.

Get the latest version of this document or suggest improvements here:
http://cs.gmu.edu/∼eclab/projects/mason/

Contents

Chapter 1

Introduction

MASON is a multiagent simulation toolkit designed to support large numbers of agents relatively efficiently on a single machine. MASON has no domain-specific features: it is not a robotics simulator like TeamBots or Player/Stage, nor is it a game library. Instead it belongs in the class of domain-independent simulators which might be unfairly described as "dots on a screen" simulators, such as Repast, StarLogo, and NetLogo, and of course, the venerable SWARM.

I call these simulators "ultra-lightweight" multiagent simulation toolkits. They're popular for problems which involve many relatively simple agents and arbitrary problems, and are common in areas like artificial life, population biology, computational social sciences, complexity science, artificial intelligence, and (in my case) robotics.

I use ultra-lightweight multiagent simulation toolkits because of previous experience in domain-specific ones. Among other things, I do research in swarm robotics and multirobotics, but often apply it to other scenarios (vehicular traffic, crowds of people, etc.), and have found that it's easier and *much less buggy* to add domain features to a general toolkit than it is to strip out features from a domain-specific one. The process of stripping out features, or working around them, leads to all sorts of dangling stuff which bites you in the end. I'm sure this isn't an uncommon experience among simulation writers.

Most of these toolkits were developed for small jobs. MASON is distinguished among these simulators because it is meant for big tasks involving a large number of simulation runs, likely on back-end server machines such as a supercomputer. As such it has quite a number of features to help in this scenario.

- MASON models are fully separated from visualization. You can easily run a model without visualization or with various different kinds of visualization, and switch among them.
- MASON models are entirely serializable to disk. This means that you can checkpoint files in case your back-end server goes down, and restart form your latest checkpoint. It also means you can run a MASON model on a back-end Linux machine, pause and serialize it to disk, move it to your Mac, unpause it under visualization, modify it, pause it again, move it to a PC, unpause it under a *different* visualization, pause it again, then put it back out on a back-end Sun server, and it'll continue to run happily as if nothing had happened.
- MASON models are entirely encapsulated. This means you can run multiple MASON models in parallel in the same process, interleaved in a big for-loop, or in separate threads, and they'll not touch each other.
- MASON is written in Java in order to make it easy to run in heterogeneous computer environments. But it also is written in very carefully written Java, with an eye firmly fixed towards efficiency. Notably, a number of poorly-written standard Sun classes are replaced with properly written MASON equivalents. Java has a reputation for being slow, but this is largely due to Sun's bad libraries and design decisions (generics and boxing, iterators, etc.). MASON eschews these internally.

- MASON has a high-quality random number generator (Mersenne Twister) and uses the fastest known implementation of it (I know — I wrote it).
- MASON models are largely *duplicable*,[1] meaning that if you run the simulation with exactly the same parameters (including random number seed) it'll run exactly the same way, even on different machines.[2]
- MASON dovetails nicely with ECJ[3], a very popular evolutionary computation (genetic algorithms, etc.) toolkit I wrote, which is designed (not surprisingly) for everything from single laptops to large jobs involving thousands of machines.
- MASON is unusually modular and consistent. There is a high degree of separation and independence among elements of the system. This allows you to use, and recombine, different parts of the system in unexpected ways. MASON is very highly hackable. Indeed, a number of MASON's classes are used, with our blessing, in MASON's competition.[4]

Many of these features are quite common among good simulation libraries. But not among those in the "ultra-light" multiagent simulation category. In this area, MASON was *completely unique* when it was developed. Since then, MASON's features have influenced the restructuring of other toolkits to include some of the above, but I believe that MASON still remains very much the high-performance, high-hackability leader among systems in this area.

What MASON is Not MASON is *not* a distributed toolkit. Yet. MASON was designed to be efficient when running in a single process, albeit with multiple threads. It requires a single unified memory space, and has no facilities for distributing models over multiple processes or multiple computers. However, there are several projects underway towards developing distributed versions of MASON.

MASON is *not* a easy toolkit for Java beginners. MASON expects significant Java knowledge out of its users. If you are a rank beginner, allow me to recommend NetLogo[5], a good toolkit with an easy-to-learn language.

Finally MASON does *not* have plug-in facilities for Eclipse or NetBeans, though it can be used quite comfortably with them. If you're looking for a richer set of development tools, you might look into Repast,[6]

MASON History In 1998, after using a variety of genetic programming and evolutionary computation toolkits for my thesis work, I decided to develop ECJ, a big evolutionary computation toolkit which was meant to support my own research for the next ten years or so. ECJ turned out pretty well: it's used very widely in the evolutionary computation field and can run on a lot of machines in parallel. ECJ was written in Java.

One common task (for me anyway) for evolutionary computation is the optimization of agent behaviors in large multiagent simulations. ECJ can distribute run many such simulations in parallel across simultaneous machines. But the number of simulations that must be run (often around 100,000) makes it fairly important to run them very efficiently. For this reason I and my students cooked up a plan to develop a multiagent simulation toolkit which could be used for various purposes, but which was fast and had a small and clean model, and so could easily be tied to ECJ to optimize, for example, swarm robotics behaviors. Because ECJ was in Java, we expected our toolkit would be in Java as well.

[1]Not to be confused with *replicable*: where the model can be built again in (say) NetLogo and it'll more or less run the same. That's also true for MASON, but it's in some sense a lower standard.

[2]Well, that's not quite true. MASON doesn't use the strictfp keyword or the java.lang.StrictMath class, so if you move to a different CPU you may get slightly different floating point results. However if you want to guarantee full duplicability, you can add the strictfp keyword and replace java.lang.Math references with java.lang.StrictMath and everything should be perfectly duplicable, albeit slower.

[3]http://cs.gmu.edu/∼eclab/projects/ecj/

[4]Well, not really competition: we're all friends here.

[5]http://ccl.northwestern.edu/netlogo/

[6]http://repast.sourceforge.net/

As we got started, I spoke with Claudio Cioffi-Revilla, who was also interested in developing a toolkit for the computational social sciences. Whereas our needs tended to be more continuous movement in 2D and 3D, many of his needs involved gridworlds and social networks. Claudio suggested we take apart Repast and examine it. We did, but ultimately followed my instinct to reinvent the wheel, and started work on MASON. Ultimately the computer science department (myself and my students) and the Center for Social Complexity (Claudio and his students and faculty) teamed up to develop and fund MASON. When MASON debuted at the Agent 2003 conference, it had model separation and encapsulation, duplicability, checkpointing, and 3D graphics, all of which is passé in simulators, but was new to the ultralight simulation field.

Since then MASON has developed considerably. It now has much more visualization, charting, selection, an improved Schedule, movable and selectable objects, etc. Claudio's influence has also been significant: MASON has extensive gridworld facilities, an external social networks package, and an external GIS facility.

I think MASON has done pretty well: it's fast and clean and has a small, modular, and orthogonal model. It's had a strong influence (I think) on other toolkit development in the field, particularly Repast. And MASON has been used for everything from small gridworld simulations to learning from demonstration on real robots to game development to large (up to millions of agents) simulations of fish, economic agents, swarm robots, and unmanned aerial vehicles.

What MASON Stands For It's a backronym, for which the blame may be placed squarely on the shoulders of John Grefenstette. Notionally it stands for Multi-Agent Simulation Of ... (we can't figure out a good "N"). Claudio likes to add "Neighborhoods and Networks", but that'd be NN!

1.1 Unpacking MASON

When you download and unzip MASON, you'll get a directory consisting of the following things:

- A CHANGES file detailing all changes made to MASON since its inception.
- A LICENSE file describing MASON's open-souce license. Most of MASON uses the Academic Free License (AFL) 3.0. This is a BSD-style license. Those few classes which are not distributed as AFL fall under other BSD-style licenses. AFL is very liberal, but if for some reason you have difficulty with this license, contact me and we can discuss other options.
- A top-level README file describing how to start up MASON and where to go.
- A Makefile. MASON does not use Ant: I'm an old Unix hacker. The Makefile can be used to build the 2D MASON facilities, the 3D facilities, to clean it, to indent all the code using Emacs, or to build a jar file.
- A docs directory, containing all documentation on the system.
- A start directory, containing double-click-on-me scripts for firing up MASON easily on various operating systems.
- A sim directory. This is the top-level package for MASON class files.
- An ec directory. This is the top-level package for the Mersenne Twister random number generator (shared with ECJ, hence the "ec").

Installing Libraries You'll need to install some libraries to make full use of MASON, and they're required if you're going to compile anything (and you're almost certainly going to want to do that!) The libraries may be found on the MASON web page[7], and consist of the following:

[7]http://cs.gmu.edu∼eclab/projects/mason/

- The JFreeChart charting library.
- The iText PDF generation library.
- (For OS X users only, optional) The Quaqua graphical interface extensions.
- The Java Media Framework toolkit for building movies

Additionally you'll want to install the Java3D toolkit if you're doing 3D. On OS X Java3D is already installed. On Windows or Linux you'll have to download and install it. See the MASON web page for instructions.

1.2 Running MASON

MASON is run in two ways: either by running an application directly, or by firing up MASON's console. If you build MASON's jar file, you can double-click on it or otherwise run it, and the console will pop up. Alternatively you can run the console by first adding the mason directory to your CLASSPATH, then running:

```
java sim.display.Console
```

You can run individual demo applications too (they're all in the sim/app directory), for example:

```
java sim.app.heatbugs.HeatBugsWithUI
```

Various scripts in the start directory can do all this for you automatically, depending on your OS. Note that certain Java3D applications suck up a lot of memory and will require more RAM. You can add more memory when running java like this:

```
java -Xmx500M -Xms500M sim.display.Console
```

This example caused MASON to be run with 500 megabytes of available virtual memory.

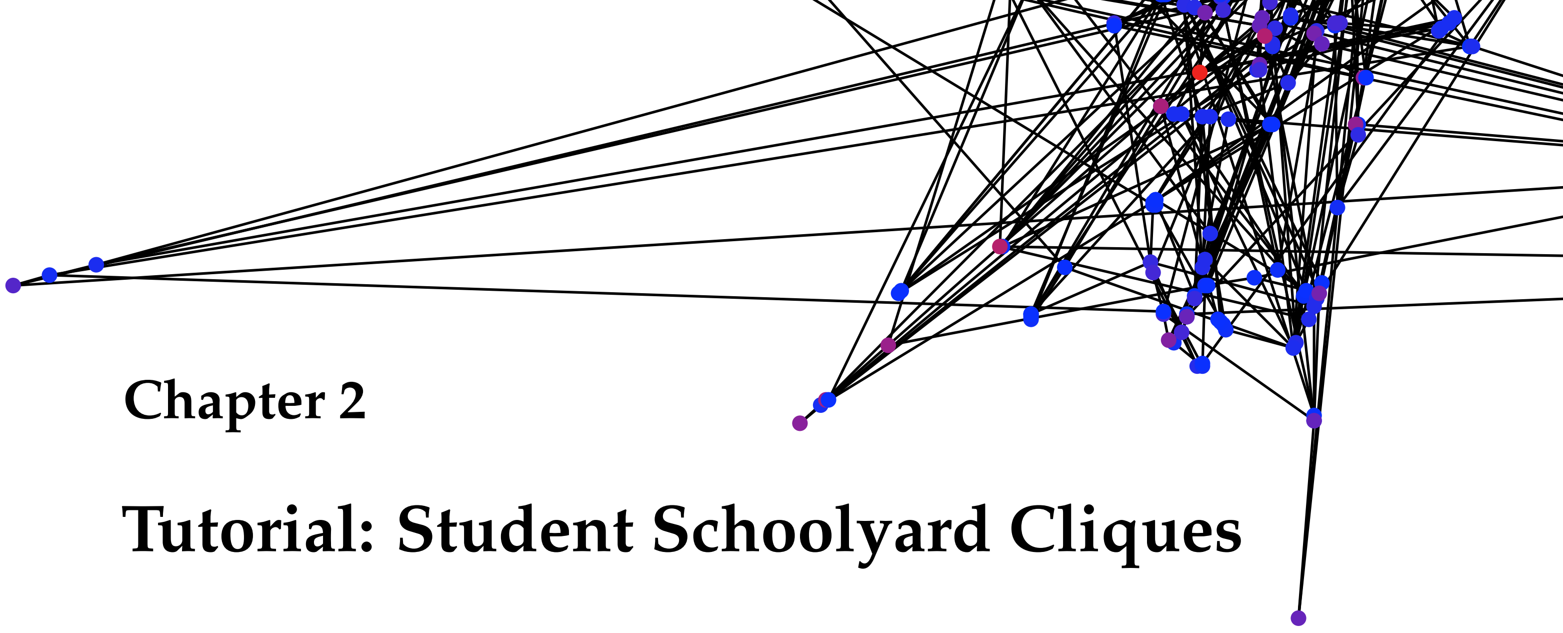
More information on Running MASON can be found in the README file and also in the docs/index.html file. On the website you can also find a tutorials for running an (earlier version of) MASON on NetBeans and Eclipse.

1.3 Additional MASON Modules

MASON has a variety of add-on modules which extend MASON in various ways. They don't come with the main package but you can easily download them from the website. Some public extensions include:

- GeoMason: GIS extensions to MASON, developed by Marc Colletti and Keith Sullivan.
- Rigid-body 2D Physics, developed by Christian Thompson.
- Social Networks. A package developed by Liviu Panait and Gabriel Balan.
- Examples for how to use JUNG (a social networks package) and MASON. By Maciej Latek.

Some of these packages may be found in the "contrib" directory of MASON's SVN repository on Google Code. Various groups have also added stuff to this directory.



Chapter 2

Tutorial: Student Schoolyard Cliques

In this tutorial sequence we will build a simple social network spread throughout the 2-dimensional continuous space of a schoolyard with a school (and notional teacher) at the center.[1]

2.1 Create an Empty Simulation

Let's begin simply, with a MASON application that does absolutely nothing.

Create the file Students.java Place in it the text:

```
import sim.engine.*;

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

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

Wow, it doesn't get a whole lot simpler than that.

When you create a MASON application, you define the simulation model as a subclass of sim.engine.SimState. MASON will create a single instance of this subclass and maintain it as the global state of your entire simulation. SimState contains two important items:

- A **random number generator**, specifically an instance of the class ec.util.MersenneTwisterFast. This generator is far better than java.util.Random. It is also not synchronized, which means that if you access it from multiple threads, you'll need to remember to acquire a lock on it first. We'll get to that later. Anyway, if you're serious about simulation, you should **never use java.util.Random nor java.lang.Math.random().** They're surprisingly poor quality.[2]

[1]This tutorial sequence was originally developed for the World Congress on Social Simulation in 2008.

[2]java.util.Random is highly non-random, and will ruin your experiments. And Math.random() just uses java.util.Random. Don't use them. For a fun example of just how bad java.util.Random is, check out "Sun Renes Randomness": http://alife.co.uk/nonrandom/

The random number generator is seeded with a random number **seed**, a long integer passed into the SimState's constructor as shown in the code above. Note that MersenneTwisterFast only uses the first 32 bits of this seed.

- A **discrete event schedule**, an instance of the class sim.engine.Schedule (or a subclass). You will schedule **agents** onto this schedule to be **stepped**: woken up at various times in the simulation to do their thing. The Schedule is your simulation's representation of time.

When you compile and run this MASON application, the doLoop(...) method is called, passing in the simulation class and the command-line arguments, and then the application terminates with a System.exit(0). The doLoop(...) performs an elaborate simulation loop with a lot of built-in gizmos, but the basic concept is actually a very simple top-level loop:

Why call System.exit(0)? Why not just exit normally?

Because if you forgot to terminate any threads created while running a MASON simulation, and forgot to declare all of your threads to be *daemon threads*, then after main(...) exits, those threads will continue to run and your simulation will not quit. Calling System.exit(0) ensures that uncooperative threads will be killed.

1. Create an instance of your SimState subclass, initializing its random number generator with a seed of some sort (perhaps the current time).
2. Call start() on the instance allow you to initialize your simulation model. (We didn't implement this method because we have nothing to initialize yet).
3. Repeatedly call step(SimState state) on the instance's discrete-event schedule, pulsing it and causing agents stored in the schedule to be stepped.
4. When the schedule is entirely empty of agents (which for us is immediately — we scheduled none), or after some *N* calls to step(...) have been made on the schedule, call finish() on the instance to let it clean up. This method is rarely used.

As mentioned before, the doLoop(...) method has lots of gizmos, but in fact, we could have written a simple version of main(...) like this:

```
public static void main(String[] args)
    {
    SimState state = new Students(System.currentTimeMillis());
    state.start();
    do
        if (!state.schedule.step(state)) break;
    while(state.schedule.getSteps() < 5000);
    state.finish();
    System.exit(0);
    }
```

All doLoop(...) does for us is provide a number of useful ways of controlling MASON from the command line. We can create checkpoints (freeze simulations in time and save them to disk), start up from a previous checkpoint, run for a certain number of steps; until a certain point in "simulation time", run multiple jobs in sequence, etc.

Compile and Run Let's compile and run this sucker. Presuming MASON has been added to your CLASSPATH...

```
javac Students.java
java Students

MASON Version 15.  For further options, try adding ' -help' at end.
Job: 0 Seed: 1293295240209
Starting Students
Exhausted
```

How do I run this on Eclipse or Netbeans?

Stay Tuned...

MASON starts by printing its version, then the job number and the random number seed for the job. It then states it's starting Students, and finally states that it quit because the schedule was **exhausted**, that is, it had no more agents to step.

2.2 Add Some Students

We next create a schoolyard and add some students to it. The students won't do anything yet—- they're not really agents until we add them to the schedule. We'll just put them in random locations. Here's the changes we make to Students.java:

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

public class Students extends SimState
    {
    public Continuous2D yard = new Continuous2D(1.0,100,100);
    public int numStudents = 50;

    public Students(long seed)
        {
        super(seed);
        }

    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));
            }
        }

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

First notice that we have added two variables to the Students class: the yard where the students exist, and the number of students. The yard will be one of our **representations of space** (we'll add the network in a moment). In MASON parlance, representations of space, which are typically displayed in the GUI, are called **fields**. A field is a generic term: you can make your own data structure if you like. But MASON provides a number of built-in fields for your convenience, and this is one of them: the class sim.field.continuous.Continuous2D.

The Continuous2D class defines a 2-dimensional environment of real-valued (continuous) space. The space may be bounded, toroidal, or infinite: here we're going to assume it's infinite, though we'll specify some bounds for drawing purposes later. Our bounds are 100 by 100. Furthermore, the Continuous2D discretizes its space into an underlying grid to make neighborhood lookups easier: we don't use that feature so we just use 1.0 as our discretization (it doesn't make any difference).

We have also added a start() method. Recall that this method is called before the schedule is stepped in order to allow the simulation to set itself up. The first thing we do in this method is call super.start(), which is very important.

We then use this method to set up the yard: we clear it of objects (the students), and then we add 50 students to the yard at random locations.

A Continuous2D can hold objects of any kind and associates each one of them with a 2-dimensional point in space, specified by a sim.util.Double2D object. Notice that we produce the X and Y values of the Double2D by choosing two small random double values from [-0.5...0.5), and adding them to the dead center of the yard. But the yard is 100x100! This places all the students in the range X=[49.5...50.5) and Y=[49.5...50.5), all clustered right around the center. This is, we presume the location of the schoolhouse where the students have just gotten out of class.

Why use Double2D? Why not use Point2D.Double?

Because Point2D.Double is *mutable*, meaning that its X and Y values can be changed once set. As a result, Point2D.Double is not safe as a key in a hashtable. And Continuous2D uses hash tables to store its objects.

You'll see Double2D and its friends (Int2D, Double3D, and Int3D) a lot. They're very simple classes which store *immutable* pairs or triples of doubles or ints. Once you set the values, you cannot change them — you'll need to make a new object — and this makes them safe for use in classes relying on hashtables.

You can easily convert between these classes and similar Java classes, such as Point or Point2D.Double.

Now we need to define what a Student is. We'll start with nothing:

Create the file Student.java Place in it the text:

```
public class Student
    {
    }
```

Compile and Run We compile and run similar to before, and get basically the same thing (since our Students aren't *doing* anything yet).

```
javac Students.java Student.java
java Students

MASON Version 15.  For further options, try adding ' -help' at end.
Job: 0 Seed: 1293298122803
Starting Students
Exhausted
```

Note that the seed has changed. It's based on the current wall clock time. If you'd like to fix the seed to a specific value (say, 4), you can do this (triggering a feature in doLoop(...)):

```
java Students -seed 4

MASON Version 15.  For further options, try adding ' -help' at end.
Job: 0 Seed: 4
Starting Students
Exhausted
```

Very exciting.

2.3 Make the Students Do Something

We next will create two simple forces: a force which causes the students to wander randomly, and a force which draws them to the schoolteacher notionally at the school at the center of the yard. We'll schedule the students on the schedule and have them wander about under the guidance of these forces.

We begin by modifying the Students.java file, adding two new variables for these forces, and also scheduling each student on the Schedule.

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

public class Students extends SimState
    {
    public Continuous2D yard = new Continuous2D(1.0,100,100);
    public int numStudents = 50;
    double forceToSchoolMultiplier = 0.01;
    double randomMultiplier = 0.1;

    public Students(long seed)
        {
        super(seed);
        }

    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));

            schedule.scheduleRepeating(student);
            }
        }

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

There are many methods for scheduling agents on the Schedule. This is one of the simplest: it schedules an agent to be stepped[3], every 1.0 units of time, starting 1.0 units from now. The Schedule starts at time 0.0. Since all the Students are being scheduled like this, each one of them will be stepped at timestep 1.0, then at 2.0, then at 3.0, and so on. Since we have not specified a sorting priority among them, each timestep the Students will be stepped in random order with respect to one another.

Now say what these Students will do when stepped. We define their step(...) methods:

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

public class Student implements Steppable
    {
    public void step(SimState state)
        {
        Students students = (Students) state;
        Continuous2D yard = students.yard;

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

[3]Or "pulsed", or "called", or "made to receive an event", or whatever you like to call it. I say "stepped".

```
        MutableDouble2D sumForces = new MutableDouble2D();

        // add in a vector to the "teacher" -- 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));
        }
    }
```

The first thing to notice is that Student implements the sim.engine.Steppable interface by implementing the method step(...). By being Steppable, the Student can be placed on the Schedule to have its step(...) method called at various times in the future. This graduates the Student from being a mere object in the simulation to being something potentially approximating a real **agent**.

When a Student is stepped, it is passed the SimState. The first thing we do is cast it into the Students class (which it is), then extract our yard (the Continuous2D).

The student needs to know where it's located in the yard. We could have stored the X and Y coordinates in the Student itself (which would be a bit faster) but can always just query the yard itself, which is what we do with getObjectLocation(...).

Seriously, why not use Point2D.Double here?

Mostly for consistency with Double2D and for the purpose of demonstration in this tutorial, that's all. Plus MutableDouble2D has many of the same methods as Double2D, plus various useful methods like addIn. If you want to use Point2D.Double, go right ahead, it'll work fine here.

Now we'd like to move the object with some artificial forces. We start with a sim.util.MutableDouble2D, which is just like a Double2D except that you can change its X and Y values after the fact. MutableDouble2D also has various sometimes-helpful modification methods like addIn. Of course, instead of foo.addIn(new Double2D(4,3)) you could instead write foo.x += 4; foo.y += 3; and be done with it (and it'd be faster too). But we're just demonstrating here!

So.... we add in a force towards the center of the yard which increases with distance to the yard; and also a small constant random force. Finally we the force to our present location, then set our location to the sum. Notice that the location must be a Double2D, not a MutableDouble2D (it must be immutable!).

Compile and Run Here we go...

```
javac Students.java Student.java
java Students

MASON Version 15.  For further options, try adding ' -help' at end.
Job: 0 Seed: 1293316557082
Starting Students
Steps: 50000 Time: 49999 Rate: 37,593.98496
Steps: 100000 Time: 99999 Rate: 54,704.59519
Steps: 150000 Time: 149999 Rate: 53,821.31324
Steps: 200000 Time: 199999 Rate: 54,824.5614
Steps: 250000 Time: 249999 Rate: 55,187.63797
Steps: 300000 Time: 299999 Rate: 54,585.15284
 ...etc...
```

Finally something new! Since the Students are scheduled on the Schedule, they're stepped every timestep. Every so often (roughly every second) the current step is printed out, plus the step rate. the Students never vacate the Schedule (they're scheduled "repeating"), this will go on forever. We could instead run for a limited time like this:

```
java Students -time 200000

MASON Version 15.  For further options, try adding ' -help' at end.
Job: 0 Seed: 1293318062046
Starting Students
Steps: 50000 Time: 49999 Rate: 37,821.4826
Steps: 100000 Time: 99999 Rate: 56,561.08597
Steps: 150000 Time: 149999 Rate: 58,004.64037
Steps: 200000 Time: 199999 Rate: 57,405.28129
Quit
```

This tells us that MASON ran until the specified time, then exited even though there were Steppable agents still scheduled on the Schedule (note the "Quit" as opposed to "Exhausted").

So far we're doing everything on the command line. Time to...

What's the difference between the Schedule's "time" and"steps"?

You can schedule agents to be stepped at any real-valued *time*. Each iteration (or *step*) of the Schedule, it looks for the timestamp of the earliest scheduled Steppable, advances the clock to that time, then extracts all the Steppables scheduled at that time and processes them. It so happens here that agents are all scheduled for timestamps of integers, so the time equals the steps more or less. But that's not necessarily the case: we could have agents scheduled for every 3.923 units of time, say.

2.4 Add a GUI Control

Note that so far everything we've done has been on the command line. Let's move toward a graphical interface by first introducing a **console** which allows us to start, stop, pause, step, and restart the simulation (among other things).

Create the file StudentsWithUI.java Place in it the text:

```
import sim.engine.*;
import sim.display.*;

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

    public StudentsWithUI() { super(new Students(System.currentTimeMillis())); }
    public StudentsWithUI(SimState state) { super(state); }
    public static String getName() { return "Student Schoolyard Cliques"; }
    }
```

MASON is somewhat unusual among multiagent simulation toolkits[4] in that it strictly distinguishes between an entirely self-contained and modular **model**, and the **visualization** or other **controls**. MASON thus largely adheres to the so-called "MVC" model (Model/View/Controller). There's a bright line between the GUI and the model: indeed the model often has no knowledge of or reference to the GUI at all. No GUI events are placed on the Schedule.

This has a number of advantages. First and foremost, we can **separate the model from the GUI** and run it on the command line (on a back-end supercomputer say), then occasionally save it, transfer it to a Windows box, and **re-integrate it with a GUI** to visualize and modify the model. Then save it and put it back out on the supercomputer to continue running. All of this can be done *in the middle of the simulation run*. The model is none the wiser.

Second, we can attach **different kinds of visualization** to the model at different times. We could visualize portions of the model in 2D, then change the visualization to something in 3D, say.

[4]But hardly unusual among big simulation toolkits in general in this respect.

Third, because the model is self-contained, we can run multiple models at the same time, either interleaved in the same thread or on separate threads.

Since MASON separates model and visualization, it has separate primary classes for each:

- The primary model class is a subclass of sim.engine.SimState.
- The class in charge of visualization is a subclass of sim.display.GUIState.

Wait, what? You can run two simulations in parallel?

Sure, why not? The models are self-contained. For example:

```
public static void main(String[] args)
    {
    long time = System.currentTimeMillis();
    SimState state = new Students(time);
    SimState other = new Students(time + 212359234);
    state.start();
    other.start();
    while(true)
         if (!state.schedule.step(state)
           && !other.schedule.step(other)) break;
    state.finish();
    other.finish();
    System.exit(0);
    }
```

In the code above we have created our GUIState subclass. We named it StudentsWithUI. In nearly all the MASON examples, when the SimState (model) is named ***Foo***, then the GUIState is named ***FooWithUI***. You hardly need to keep this tradition — feel free to name your UI class whatever you like — but if you want to be consistent, there you go.

The StudentsWithUI class usually has two constructors: a default (empty) constructor, and one which takes a SimState. The default constructor simply creates an appropriate SimState, typically providing the current wall-clock time as the seed as shown, and passes it to the other constructor.

Subclasses of GUIState also provide a name for the simulation via the method getName(). This name will appear in GUI controls to describe the simulation.

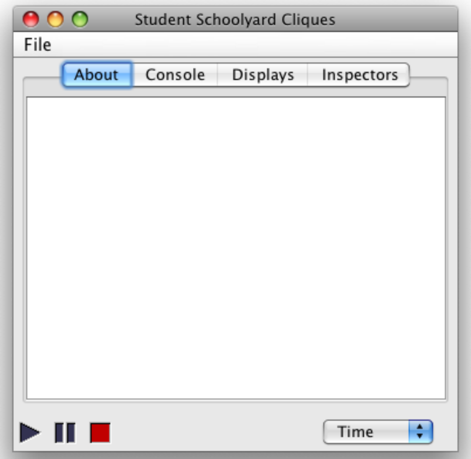
We also provide a new main(...) method. Rather than create a simulation and run it, this method fires up the visualization (which handles creating the simulation on our behalf). Here's what the GUIState's main(...) method must do:

- Create one of our GUIState subclasses.
- Create a Console. This is a GUI control which allows us to start/stop/pause/etc. the simulation.
- Make the Console visible.

...and that's just what our code does.

Compile and Run When we compile and run the code...

```
javac StudentsWithUI.java Students.java Student.java
java StudentsWithUI
```



...we no longer get boring command-line results. Instead, we get the window shown at right. This is the **Console**. It's the command center for your simulation: it controls playing, pausing, and stopping the simulation; plus showing and hiding displays and inspector panels. You can save and load simulations to and from checkpoints and quit using the Console.

An introduction:

▶ The Play button. Press this to start a simulation: the start() method is called on the SimState, followed by repeated calls to step the Schedule.

❚❚ The Pause button. Press this to pause or unpause the simulation. While paused, the Play Button changes to ▮▶ the Step Button, which lets you step through the simulation.

■ The Stop button. Press this stop a running simulation: the finish() will be called on the SimState.

Time When a simulation is playing, either the current simulation time, steps, or rate is shown next to the Stop button. This chooser lets you select which.

About This tab displays descriptive simulation information of your choice as an HTML file (at present there isn't any, but wait).

Console This tab shows various widgets for manipulating the speed of the simulation, automatically pause or stop at a certain time, etc.

Displays This tab shows the list of all visualization **displays**. At present we have none, but we will have one soon!

Inspectors This tab shows the list of all current **inspectors**[5] of objects in the simulation. At present we have none, we'll see them presently.

File The **Console Menu**, which allows to save and load simulations, choose new simulations, or quit.

Start the Simulation Press the Play Button and watch it go. Not very interesting yet: because we don't yet have a **visualization**. That's next.

2.5 Add Visualization

Now we're getting somewhere! The code below is more complex than we've seen so far, but it's not too bad. We're going to add 2-dimensional visualization object called a sim.display.Display2D. This object is capable of displaying several fields at once, layered on top of one another. We only have one field (the schoolyard, a Continuous2D). Such an object, in MASON parlance, is called a **display**. Displays can sprout their own windows (javax.swing.JFrame) to hold them, and we'll do that.

We will also attach to the Display2D a single **field portrayal**. This object is responsible for drawing and allowing inspection of a field (in our case, the schoolyard). In our case, the field portrayal will be an instance of sim.portrayal.continuous.ContinuousPortrayal2D.

The field portrayal will draw and inspect individual objects stored in the Continuous2D by calling forth one or more **simple portrayals** registered with the field portrayal to draw those objects. Our objects are students. For our simple portrayal we'll choose a sim.portrayal.simple.OvalPortrayal2D, which draws objects as gray circles (you can change the color).

First here's the code. Afterwards we'll go through each of the new methods. Notice the three new instance variables in our GUIState:

- The display.
- The JFrame which holds the display.
- The field portrayal responsible for portraying the yard.

```
import sim.portrayal.continuous.*;
import sim.engine.*;
import sim.display.*;
import sim.portrayal.simple.*;
import javax.swing.*;
import java.awt.Color;

public class StudentsWithUI extends GUIState
    {
```

[5]Some other multiagent simulation toolkits, following the Swarm tradition, use the term *probes*. When building MASON I chose the term "inspectors", following the GUI tradition of NeXTSTEP and later OS X.

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

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

public StudentsWithUI() { super(new Students( System.currentTimeMillis())); }
public StudentsWithUI(SimState state) { super(state); }
public static String getName() { return "Student Schoolyard Cliques"; }

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

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

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();
    }

public void init(Controller c)
    {
    super.init(c);
    display = new Display2D(600,600,this);
    display.setClipping(false);

    displayFrame = display.createFrame();
    displayFrame.setTitle("Schoolyard Display");
    c.registerFrame(displayFrame);        // so the frame appears in the "Display" list
    displayFrame.setVisible(true);
    display.attach( yardPortrayal, "Yard" );
    }

public void quit()
    {
    super.quit();
    if (displayFrame!=null) displayFrame.dispose();
    displayFrame = null;
    display = null;
    }
}
```

Okay, so what's going on? We have added four primary methods. Notice than in all four cases, we call super... first. That's important.

- init(...) is called when the GUI is initially created. In this method we construct a new display of size 600x600 pixels, and tell it not to clip the underlying field portrayal to the field's height and width (100x100). This permits the field portrayal to display values far outside the field boundaries, which we want because our students will be permitted to wander as far and wide as they like.

  We then tell the display to sprout a frame, and give it a title. We then register the frame with the Console (the Console is passed into the method as its superclass, a sim.display.Controller), which causes it to appear in the Console's displays list. We make the frame visible and attach the field portrayal to the display (calling it the "Yard").

- start() is called when the Play Button is pressed, and just before the SimState's start() method is called.[6] This method simply calls a method we made up called setupPortrayals(), which will be described below.

- load(), which is called when a simulation is loaded from a checkpoint. Almost always, load(...) will do the same thing as start(), which is why we created the separate method setupPortrayals() (so they could both call it).

- quit() is called when the GUI is about to be destroyed; this gives us a chance to clean up our GUI. Here, we dispose the frame and set it to null if it's not already. This is important because it's possible that quit() may be called more than once — we don't want to dispose the frame *twice*. We also set the display to null to assist in garbage collection, though that's hardly necessary.

Where's finish()?

It's there. Just as SimState had a start() and finish(), so does GUIState. But, as in SimState, it's less used.

The method setupPortrayals(),[7] which was created separately so that both load() and start() can call it, is where we set up the visualization when the user is about to start playing the simulation. Before describing it, it's first important to note certain instance variables stored in the GUIState.

- state The model (SimState) for the current simulation. Set after calling super.start() or super.load(...). In our case, this will be a Students instance.

- controller The controller (usually the Console) for the GUI system. Set after calling super.init().

Armed with these two instance variables (okay, at least the first one), we need to do two major tasks in the setupPortrayals() method: (1) set up various field portrayals and simple portrayals and attach them to fields, and (2) reset and clean up the display. In the case of this tutorial, here's what we'll need to do:

- Tell the field portrayal which field it will be portraying (the yard).

- Tell the field portrayal that all objects in the yard will be portrayed using an OvalPortrayal2D.

- Reset the display. This causes it to re-register itself with the GUIState asking to be repainted by the GUIState after every step of the Schedule.

[6]There's also a finish() method complementary to the SimState's finish() method, but just as is the case for SimState, you'll rarely override it.

[7]setupPortrayals() is not a MASON library method: it's just an arbitrary method name I created here to reduce repeated code in this and other examples. If you look in the application demos you'll see this theme often, but it's not required by any means.

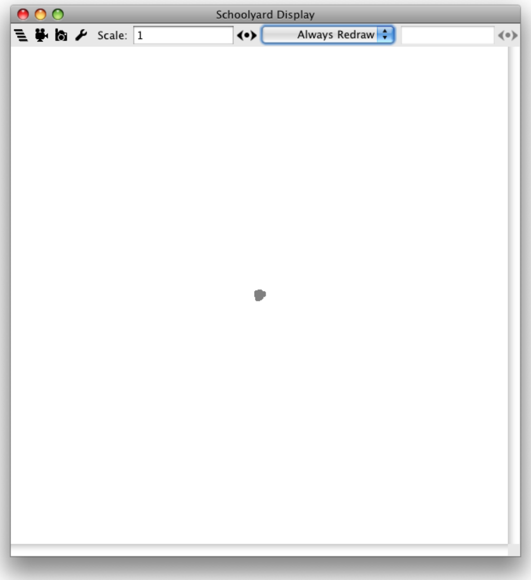
- Set the backdrop (the background color) of the display to white.
- Repaint the display once to display the initial arrangement of the schoolyard prior to running the simulation.

Why not use the standard Java method name setBackground(...)*?*

Because setBackdrop() doesn't set the background color of the Display2D widget, but rather the background color of an inner display which is, among other widgets, part of the Display2D. That's my excuse anyway.

Compile and Run When we compile and run the code...

```
javac StudentsWithUI.java Students.java Student.java
java StudentsWithUI
```



Notice that we now have *two windows*:[8] the Console and a new Display2D window (actually a JFrame holding a Display2D within it). The Display2D is shown at right.

At its default scaling, the Display2D shows the 100x100 region of the schoolyard. You can see the children all clustered around the center of the schoolyard with a white background. Press Play and you'll see them bouncing around in that tight cluster but never leaving (the force of the teacher is too strong). We'll get them doing something more interesting in a moment. But first, let's discuss some of the elements of the Display2D window:

The Layers menu. The Display2D overlays each field portrayal on top of one another. This menu allows you to selectively display or hide various field portrayals (we only have one for now).

The Movie button. Press this to start a movie of the simulation running.

The Camera button. Press this to take a picture of the simulation. You can save the picture out as a bitmap (a PNG file) or as a publication-quality PDF vector image.

The Options button. Press this to bring up various drawing options.

The Scale Field. This widget lets you zoom in and out of the display (change its scale).

The Redraw Field. This widget lets you zoom in and out of the display (change its scale).

2.6 Add a Social Network

Right now the simulation isn't very interesting because the students are all huddled around the teacher. Let's make them separate from one another based on how much they like or dislike one another.

We'll do this by adding a new field called a **network**, defined by the class sim.field.network.Network. This class defines directed and undirected graphs and multigraphs[9] with unlabeled, labeled, or weighted edges.

Network allows any object to be a node. Objects are connected via edges defined by the class sim.field.network.Edge. An Edge stores an info object which labels the edge: this can be anything.

Students will be embedded in an undirected graph, plus some random edges indicating strong like or dislike of one another. If students lack an edge between them, we assume they have no mutual opinion.

Make the following changes to the Students class:

[8]You can have as many displays as you want. Furthermore, you can have multiple displays for the same field; multiple fields shown in the same display; whatever you like.

[9]A *multigraph* is a graph where more than one edge can exist between two nodes.

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

public class Students extends SimState
    {
    public Continuous2D yard = new Continuous2D(1.0,100,100);
    public int numStudents = 50;
    double forceToSchoolMultiplier = 0.01;
    double randomMultiplier = 0.1;
    public Network buddies = new Network(false);

    public Students(long seed)
        {
        super(seed);
        }

    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));

            buddies.addNode(student);
            schedule.scheduleRepeating(student);
            }

        // define like/dislike relationships
        Bag students = buddies.getAllNodes();
        for(int i = 0; i < students.size(); i++)
            {
            Object student = students.get(i);

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

            // who does he dislike?
            do
                studentB = students.get(random.nextInt(students.numObjs));
            while (student == studentB);
            buddiness = random.nextDouble();
            buddies.addEdge(student, studentB, new Double( -buddiness));
            }
        }

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

The first thing that the new code does is define a Network called buddies. This is the student relationships graph. Notice that it's created with the parameter false, indicating that it's an *undirected* graph. Then each student is added to the graph as a node.

Afterwards the code adds random edges to this graph. Each student likes at least one other student (not himself) and dislikes at least one other student (not himself). It's possible for students to both like and dislike one another.

Here's how the code works. We first extract all the students from the graph with the method getAllNodes(). This method returns a sim.util.Bag, a faster replacement for Sun's java.util.ArrayList which is used throughout MASON whenever speed is of importance. Bag more or less has the same methods as ArrayList, so it shouldn't be hard to learn.

The getAllNodes() method returns a Bag used internally in Network, and which you should treat as **read-only**: do not modify it. If you want to modify it, make a copy and modify that.

The code does a loop through this bag of Students and, for each student, adds one random Edge for like and dislike respectively. Each edge goes **from** the student **to** another student, and is **weighted** with the degree of affinity (negative indicates dislike). We pick the degree of affinity using the random number generator.

Why use a Bag? Why not an ArrayList?

Up until Java 1.6, Sun's ArrayList class has been *very slow*. ArrayList's get(...), set(...), and add(...) methods have not been inlinable due to errors on Sun's part. And of course, you can't access the underlying array directly. We created Bag many years ago as a replacement for ArrayList with properly inlinable methods and with direct access to the array (if you're careful). This made Bag *much* faster than ArrayList. Since extensible arrays are so common in simulation, and back-end simulation must be efficient, we felt it was an acceptable compromise to make: using a nonstandard class in return for an almost five-fold speedup. It was a good decision for almost ten years.

Things have finally changed in Java 1.6. Sun's HotSpot VM can now inline even methods like those in ArrayList. This makes ArrayList decently fast; and indeed faster than Bag in certain cases. So we may switch back to ArrayList at some time in the future if present trends continue.

Have the Students Respond to the Network Now that the students have likes and dislikes, let's create a force that causes each Student to go towards the kids he likes and away from those he dislikes.

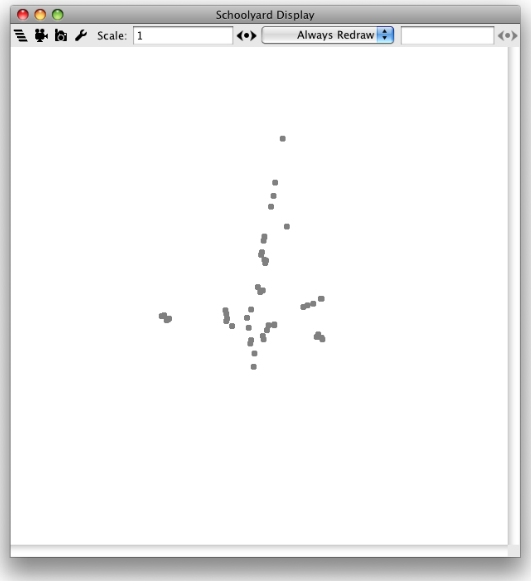
Here's how we'll do it for any given Student. We create a temporary MutableDOuble2D called forceVector, which folds the force from one student to another. We then extract from the network all the edges connecting the Student to his buddies (those whom he likes or dislikes).

For each buddy, we grab the degree of affinity. If the affinity is positive, then we create a force which draws the Student to the buddy. If the affinity if negative, then we store in forceVector a force which pushes the Student away from the buddy. These forces are proportional to how close (or far away) the Student is from the buddy, like springs. Finally, we add the force into the sumForces.

Here's the edited Student.java file:

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

public class Student implements Steppable
    {
    public static final double MAX_FORCE = 3.0;

    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);
        int len = out.size();
        for(int buddy = 0 ; buddy < len; buddy++)
            {
            Edge e = (Edge)(out.get(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));

            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.resize(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.resize(0.0);
                else if (forceVector.length() > 0)
                    forceVector.resize(MAX_FORCE - forceVector.length());  // invert the distance
                }
            sumForces.addIn(forceVector);
            }

        // add in a vector to the "teacher" -- 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));
        }
    }
```

Notice the various MutableDouble2D convenience methods being used. First there's the setTo(...) method: this simply replaces the X and Y values in the MutableDouble2D with the given values. Next, there's the length() method, which returns $\sqrt{X^2 + Y^2}$. And the method resize(...) scales X and Y so that $\sqrt{X^2 + Y^2} = L$ for the desired length L. There are a lot more than that. These methods are just for convenience of course: you can just set the X and Y values yourself.



Compile and Run If you compile and run the simulation at this point...

```
javac StudentsWithUI.java Students.java Student.java
java StudentsWithUI
```

...you'll find that the students now push away from one another, settling into lines of affinity. But we can't see

the network yet, nor visualize the strain it's putting on
the students. We'll do that next.

2.7 Visualize the Social Network

Visualizing a Network is easy, since we're already drawing the nodes (the Students). We merely have to set up a sim.portrayal.network.NetworkPortrayal2D and tell it two things:

- What Network it should be portraying (this tells it which edges to draw).
- What Continuous2D or other spatial field is associating the nodes (in our case, the Students) with locations. This tells it *where* to draw the edges.

A NetworkPortrayal2D, like all field portrayals, is assigned a field to draw via the method setField(...). Since the NetworkPortrayal2D in actuality needs *two* fields, we create a special "field" of sorts which stores both of them and pass that in instead. That object is called a sim.portrayal.network.SpatialNetwork2D.

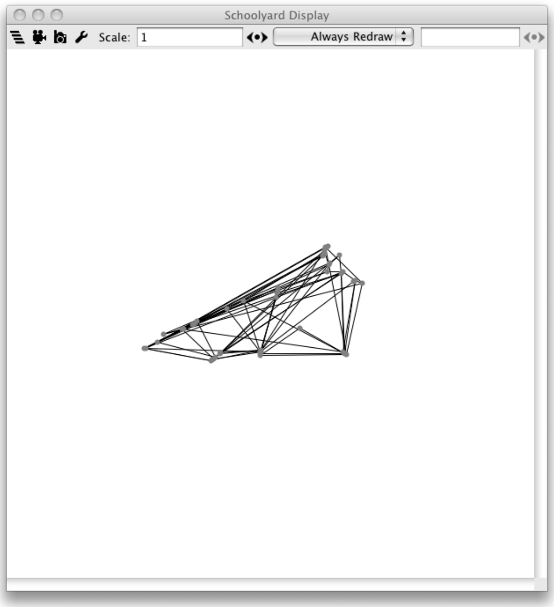
The nodes are already being drawn: we simply need to define how to portray the edges. We'll do this by assigning a sim.portrayal.network.SimpleEdgePortrayal2D, which just draws them as black lines.[10]

We edit the StudentsWithUI.java file. The changes are pretty small:

```
import sim.portrayal.network.*;
import sim.portrayal.continuous.*;
import sim.engine.*;
import sim.display.*;
import sim.portrayal.simple.*;
import javax.swing.*;
import java.awt.Color;


public class StudentsWithUI extends GUIState
    {
    public Display2D display;
    public JFrame displayFrame;

    ContinuousPortrayal2D yardPortrayal = new ContinuousPortrayal2D();
    NetworkPortrayal2D buddiesPortrayal = new NetworkPortrayal2D();

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

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

    public static String getName() { return "Student Schoolyard Cliques";  }

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

    public void load(SimState state)
        {
```

[10]You can change the color if you like. And for directed edges, you can choose the color of the "from" portion of the edge and the color of the "to" portion of the edge to distinguish them. There are other non-line options as well. And yes, you can draw the labels or weights.

```
        super.load(state);
        setupPortrayals();
        }

    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();
        }

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

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

        displayFrame = display.createFrame();
        displayFrame.setTitle("Schoolyard Display");
        c.registerFrame(displayFrame);   // register the frame so it appears in the "Display" list
        displayFrame.setVisible(true);
        display.attach( buddiesPortrayal, "Buddies" );
        display.attach( yardPortrayal, "Yard" );
        }

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

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



Compile and Run And we're off and running!

```
javac StudentsWithUI.java Students.java Student.java
java StudentsWithUI
```

As shown at right, we've now got edges being drawn. Now let's modify how the nodes are drawn to reflect how agitated the students are.

2.8 Inspect and Student Agitation and Customize its Visualizaion

Before we can visualize how agitated the students are, we'll need to actually define what "agitated" means. We'll define it in terms of the affinity forces: a student is happier if he has lower affinity forces on him (he's fine where he is), and more agitated if the forces are high.

Modify the file Student.java:

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

public class Student implements Steppable
    {
    public static final double MAX_FORCE = 3.0;

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

    public void step(SimState state)
        {
        Students students = (Students) state;
        Continuous2D yard = students.yard;

        Double2D me = students.yard.getObjectLocation(this);

        MutableDouble2D sumForces = new MutableDouble2D();

        friendsClose = enemiesCloser = 0.0;

        // 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);
        int len = out.size();
        for(int buddy = 0 ; buddy < len; buddy++)
            {
            Edge e = (Edge)(out.get(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));

            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.resize(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.resize(0.0);
                else if (forceVector.length() > 0)
                    forceVector.resize(MAX_FORCE - forceVector.length());  // invert the distance
                enemiesCloser += forceVector.length();
                }
            sumForces.addIn(forceVector);
           }

        // add in a vector to the "teacher" -- 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));
        }
    }
```

Notice that we've set up this value (agitation) as a read-only Java Bean Property via the method getAgitation(). This will come in handy later.

Notice also that the initial size of the variable enemiesCloser is 10.0, not 0.0. There's a reason for this. When the students exit the schoolyard, they're very close to their enemies; but this isn't reflected in the very first frame of the simulation visualization, because the step(...) method hasn't been called yet. So they all will look initially very happy. We remedy this by making them initially very agitated at step 0!

Change Student Color Now let's customize the color of the student to reflect their degree of agitation. You can portray the students in any way you like: just create your own sim.portrayal.SimplePortrayal2D subclass; or have the Students themselves subclass from SimplePortrayal2D. But you could also take an existing SimplePortrayal2D and modify it: typically change its size or its color. We'll do that.

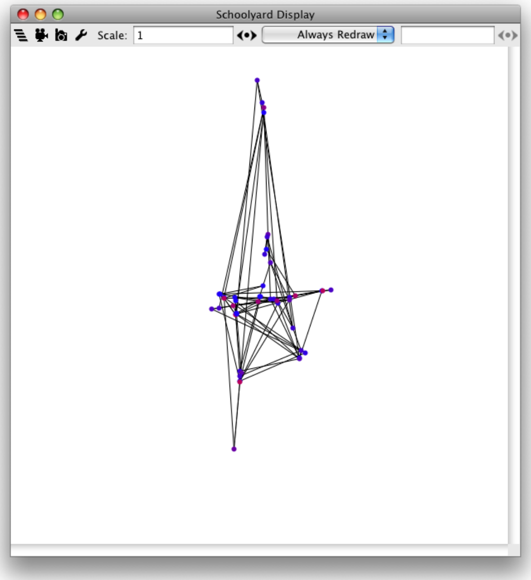
Recall that our students are presently being portrayed as gray circles using the SimplePortrayal2D subclass sim.portrayal.simple.OvalPortrayal2D. This class has three variables you can customize: Paint paint (the color), double scale (the size), and boolean filled. We could just set the color and be done with it, but to have the color dynamically change each step, we'll need to override the method draw(...) to change the color to the proper value, then call super.draw(...) to draw the oval.

You can make a custom subclass in its own file, but it's just simpler to use an anonymous subclass:

```
import sim.portrayal.network.*;
import sim.portrayal.continuous.*;
import sim.engine.*;
import sim.display.*;
import sim.portrayal.simple.*;
import sim.portrayal.*;
import javax.swing.*;
import java.awt.Color;
import java.awt.*;

public class StudentsWithUI extends GUIState
    {
    public Display2D display;
    public JFrame displayFrame;

    ContinuousPortrayal2D yardPortrayal = new ContinuousPortrayal2D();
    NetworkPortrayal2D buddiesPortrayal = new NetworkPortrayal2D();

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

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

    public static String getName() { return "Student Schoolyard Cliques";  }
    public void start()
        {
        super.start();
        setupPortrayals();
        }

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

    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()
            {
            public void draw(Object object, Graphics2D graphics, DrawInfo2D info)
                {
                Student student = (Student)object;

                int agitationShade = (int) (student.getAgitation() * 255 / 10.0);
                if (agitationShade > 255) agitationShade = 255;
                paint = new Color(agitationShade, 0, 255 - agitationShade);
                super.draw(object, graphics, info);
                }
            });

        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();
        }

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

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

        displayFrame = display.createFrame();
        displayFrame.setTitle("Schoolyard Display");
        c.registerFrame(displayFrame);   // register the frame so it appears in the "Display" list
        displayFrame.setVisible(true);
        display.attach( buddiesPortrayal, "Buddies" );
        display.attach( yardPortrayal, "Yard" );
        }

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

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

```
        display = null;
        }
    }
```

Note that the draw(...) method is passed the object to draw (a Student). From this we extract the agitation, then set an appropriate Color (ranging from red to blue). Finally (and crucially) we call super.draw(...) to draw the Student.



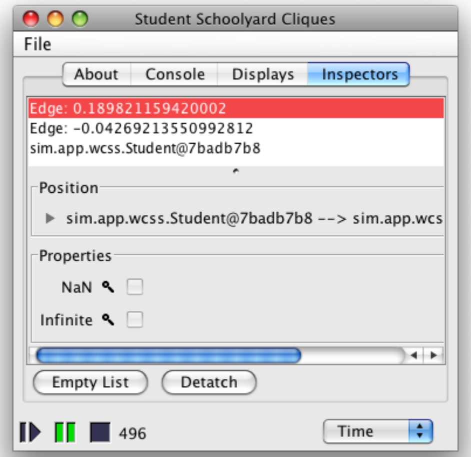
Compile and Run

javac StudentsWithUI.java Students.java Student.java
java StudentsWithUI

As you can see at right, we now have students at various levels of agitation. As students head away from one another, they'll get less agitated, but some students seem to be unhappy no matter where they are.

Inspect Now that our students have certain Java Bean Properties, we can inspect those values in various ways. For example, we can:

- Examine the values (and modify them if the Properties are writable, which isn't the case in this example).
- Track properties on a time series chart or a histogram.
- Write the properties out to a file or to a text window.



Edge: 0.189821159420002
Edge: -0.04269213550992812
sim.app.wcss.Student@7badb7b8
Position
sim.app.wcss.Student@7badb7b8 --> sim.app.wcs
Properties
NaN
Infinite

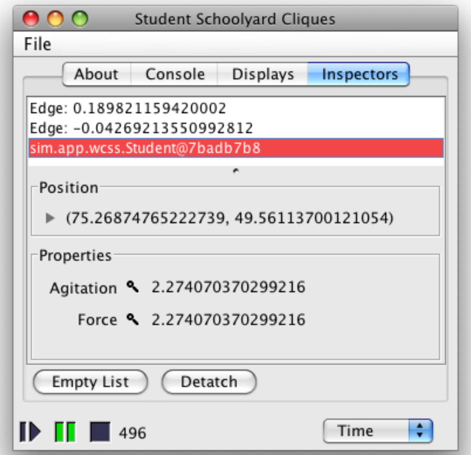
It's easy to to inspect a Student now. Double-click on a student and the Console will shift to showing a list of Inspectors, one for each object you hit with the mouse. If you find it challenging to select a student, try rescaling the simulation a bit via the Scale Field Scale: 1,

The two things inspectable in the simulation, at the moment, are Students and Edges. At right is the Console after I clicked on a particularly simple Student with just two Edges. Often you'll get a bounty of Edges, followed by one or a few Students, depending on what you clicked on.

In the Console at right we see the list of Inspectors, and beneath it the sim.portrayal.SimpleInspector for an Edge with weight 0.18921159420002.[11] An Inspector (usually a SimpleInspector) is produced by the SimplePortrayal2D (in this case, the OvalPortrayal2D) when requested by the Display2D after the user double-clicked on an object. Notice that the Edge has two properties: *NaN* and *Infinite*. The Edge is displaying a SimpleInspector for its info object, and we placed a java.lang.Double in there (that's what's holding the 0.18921159420002 value). Sun's Double class has two unhelpful Java Bean Property methods: isNaN() and isInfinite().

[11] Actually, it's *two* inspectors one nested in the other: the SimpleInspector is displaying the stuff inside the "Properties" box. The ContinuousPortrayal2D wrapped this in an outer Inspector which also includes the position of the object. Also if you look carefully you'll see the Students are called sim.app.wcss.Student rather than just Student: I took these screenshots from the WCSS tutorial. So sue me.

Click on the Student[12] and you'll get an Inspector like the one shown at right. The Student has two properties: *Agitation* and *Force*. These correspond to the two Java Bean Property Methods we defined: getAgitation() and getForce(). If you continue the simulation, you'll find that these properties automatically update.[13]



Edge: 0.189821159420002
Edge: -0.04269213550992812
sim.app.wcss.Student@7badb7b8

Position
(75.26874765222739, 49.56113700121054)

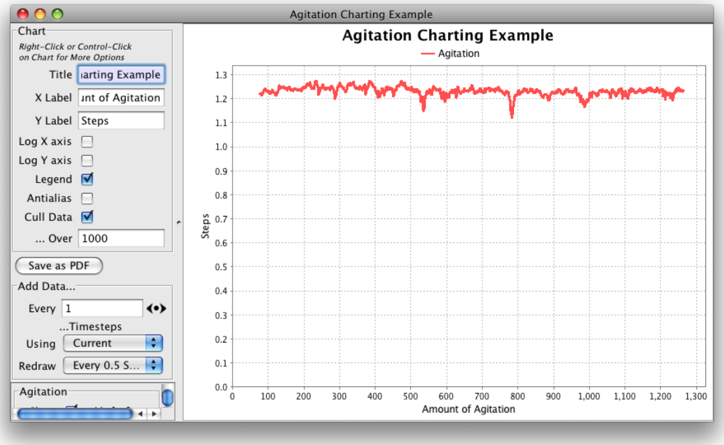
Properties
Agitation 2.274070370299216
Force 2.274070370299216

496

We can go further than this. Let's chart the degree of agitation of a given Student. Stop the simulation and start it paused (click on the Pause button when the console is Stopped). Then step the simulation a few times and double-click on a student and inspect it. Next, click on the magnifying glass button next to the *Agitation* property. Up will pop up a menu which lets you *stream* the property to a file or to a window, or *chart* the property[14] by calling forth further inspectors to do more interesting things.

These inspectors are different from the SimpleInspectors you've seen so far. First off, the SimpleInspector is designed to inspect an *object*. These additional inspectors, subclasses of sim.portrayal.inspector.PropertyInspector, are designed to inspect *the value of a property of an object*. Second, PropertyInspectors can be dynamically plugged-in at run-time, and appear in that pop-up menu.

Select *Chart* from the pop-up menu and you'll get a chart window like the one at right. Make some tweaks of features, unpause the simulation, and watch the chart unspool. If you click on "Save as PDF...", you can generate a publication-quality PDF output of the chart (as opposed to a bitmap screenshot).



Title: ıarting Example
X Label: ınt of Agitation
Y Label: Steps
Log X axis: ☐
Log Y axis: ☐
Legend: ✓
Antialias: ☐
Cull Data: ✓
... Over: 1000
Every: 1 ...Timesteps
Using: Current
Redraw: Every 0.5 S...

[12]Don't like that ugly name for the Student in the Inspector's list? Just override the Student's toString() method, and it'll be used instead.

[13]Also the simulation runs more slowly because there's more stuff to draw on-screen. Click "Empty List" to get rid of the Inspectors and make the simulation run faster. You can make the simulation run even faster by getting rid of the display. Click the display's close box (on OS X, the red button). Last, change the Time Button Time to say "Rate", which is updated less often. With no displays to draw, no inspectors to update, and no steps counter to repaint, the GUI will approach — but not match — the speed you get if you just run on the command line. Don't worry, you can get the display back – go to the Displays tab Displays and double-click on the display.

[14]MASON uses the charting library *JFreeChart* (http://jfreechart.org) to do charting in its inspectors.

2.9 Inspect the Model

Let's add an inspector for the global model parameters. Before we do so, it might be useful to *have* global model parameters! So we'll modify the Students.java file, adding some Java Bean Properties. We add four properties below, and they all have interesting features which need to be explained:

- getNumStudents() is accompanied by setNumStudents(...). This makes it a read-write property, not just a read-only property. As a result you'll be able to change the number of students in the Inspector (via a text field), not just view it.
- Likewise, getForceToSchoolMultiplier() is accompanied by setForceToSchoolMultiplier(...), making it a read-write property.
- getRandomMultiplier() is not only accompanied by setRandomMultiplier(...), making it a read-write property, but it's also accompanied by a special method name custom to MASON: domRandomMultipler(). If a property *Foo* has a method called domFoo(), MASON interprets this as providing the *domain* of the property. There are two domains recognized: either a sim.util.Interval, which defines the legal range of a numerical property; or an array of Strings. In the first place, MASON replaces the standard read-write text field with a slider. In the second case, MASON replaced the text field with a pop-up menu of those Strings. If you choose a String, then the property is set to the index value of the String in the array (starting at 0).
- getAgitationDistribution() is computed at runtime. Furthermore it doesn't return a simple integer or boolean value: it returns an *array of doubles*. Arrays or lists of objects allow you to create historgrams as property inspectors.

Here's the code.

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

public class Students extends SimState
    {
    public Continuous2D yard = new Continuous2D(1.0,100,100);

    public int numStudents = 50;

    double forceToSchoolMultiplier = 0.01;
    double randomMultiplier = 0.1;

    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); }

    public double[] getAgitationDistribution()
        {
        Bag students = buddies.getAllNodes();
        double[] distro = new double[students.numObjs];
        for(int i = 0; i < students.numObjs; i++)
            distro[i] = ((Student)(students.objs[i])).getAgitation();
        return distro;
        }
    public Network buddies = new Network(false);

    public Students(long seed)
        {
        super(seed);
        }

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

        // 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);
            }

        // define like/dislike relationships
        Bag students = buddies.getAllNodes();
        for(int i = 0; i < students.size(); i++)
            {
            Object student = students.get(i);

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

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

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

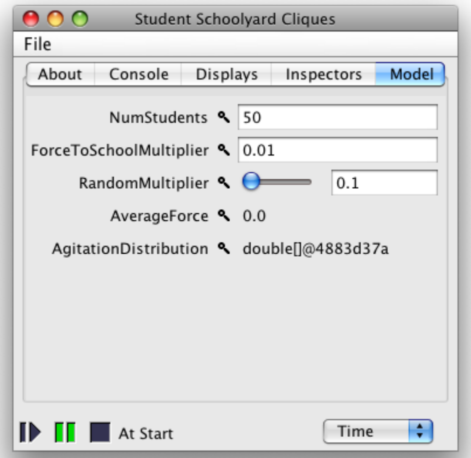
We also need to modify the file StudentsWithUI.java to inform MASON that it should display an Inspector for the model. We need to actually tell it two things:

- The object from which it should extract the Java Bean Properties. Typically this is the model (SimState subclass) itself. To do this we add a single method called getSimulationInspectedObject().
- That the Inspector is *volatile* and thus must be updated every timestep. This is expensive, and not all that common (usually we use model inspectors to set parameters at the beginning of a run). But in this example, our Inspector will be providing data such as histogram information which changes each timestep. So we need to declare it to be volatile. To do this we override the method getInspector() to set the inspector to be volatile before returning it.

Here's the code, it's pretty straightforward:

```
import sim.portrayal.network.*;
import sim.portrayal.continuous.*;
import sim.engine.*;
import sim.display.*;
import sim.portrayal.simple.*;
import sim.portrayal.*;
import javax.swing.*;
import java.awt.Color;
import java.awt.*;

public class StudentsWithUI extends GUIState
    {
    public Display2D display;
    public JFrame displayFrame;

    ContinuousPortrayal2D yardPortrayal = new ContinuousPortrayal2D();
    NetworkPortrayal2D buddiesPortrayal = new NetworkPortrayal2D();

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

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

    public static String getName() { return "Student Schoolyard Cliques";  }

    public Object getSimulationInspectedObject() { return state; }

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

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

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

    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()
            {
            public void draw(Object object, Graphics2D graphics, DrawInfo2D info)
                {
                Student student = (Student)object;

                int agitationShade = (int) (student.getAgitation() * 255 / 10.0);
                if (agitationShade > 255) agitationShade = 255;
                paint = new Color(agitationShade, 0, 255 - agitationShade);
                super.draw(object, graphics, info);
                }
            });

        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();
        }

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

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

        displayFrame = display.createFrame();
        displayFrame.setTitle("Schoolyard Display");
        c.registerFrame(displayFrame);   // register the frame so it appears in the "Display" list
        displayFrame.setVisible(true);
        display.attach( buddiesPortrayal, "Buddies" );
        display.attach( yardPortrayal, "Yard" );
        }

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

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



NumStudents 50
ForceToSchoolMultiplier 0.01
RandomMultiplier 0.1
AverageForce 0.0
AgitationDistribution double[]@4883d37a

Compile and Run

```
javac StudentsWithUI.java Students.java Student.java
java StudentsWithUI
```

When you fire up MASON this time, notice that the Console now has an extra tab at the end: the **Model Tab** Model. This tab reveals the Model's Inspector, as shown at right. Notice the five properties we added, one of which has a slider.

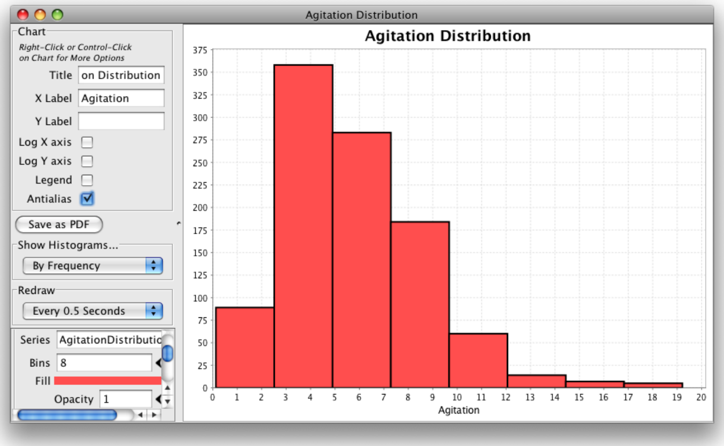


Chart
Right-Click or Control-Click on Chart for More Options
Title: on Distribution
X Label: Agitation
Y Label:
Log X axis ☐
Log Y axis ☐
Legend ☐
Antialias ✓
Show Histograms...: By Frequency
Redraw: Every 0.5 Seconds
Series: AgitationDistributic
Bins: 8
Fill: 🟥
Opacity: 1

Important Note The Model Inspector isn't valid until you start the simulation. So examining or setting values when the Simulation is Stopped ■ may have no effect. Instead, press Pause ❚❚ first, then modify and examine the inspector.

Try tweaking these values. Note that you can only modify the number of students before starting a simulation (the model uses this value at start(...) time and nowhere else) but the two Multiplier properties can be modified whenever you like. Try setting the number of students to 1000 for example. Or change the degree of randomness.

You can create time series charts from the *AverageForce* property. But more interesting, you can now create histograms of from the double array generated by getAgitationDistribution(). Just click on the magnifying glass button 🔍 and select *Histogram*. Up pops a histogram like the one here.

Can you have more than one series on a chart?

Absolutely. Histograms and time series charts both support multiple series. Just ask to chart a second item, and you'll be given the option to put it on its own chart or to add it to an existing chart.

2.10 Select, Label, and Move Students

MASON allows you to *select* objects with the mouse, which signals further state changes in those objects; and to *move* objects by dragging them. Selecting happens automatically, but there's no effect unless you add something which responds to the selection. Moving is not enabled by default at all, so we'll do it here.

Let's start with selection.

MASON has a concept called **wrapper portrayals**, which are special SimplePortrayals which contain nested SimplePortrayals (the primary portrayals) within them. A wrapper portrayal adds additional functionality beyond what the SimplePortrayal provides. You can have as many wrapper portrayals wrapped around your SimplePortrayal as you like. Here are some wrapper portrayals available:

- sim.portrayal.simple.CircledPortrayal2D draws a circle around the object to hilight it. CircledPortrayals can be set to to only draw the circle when the object is selected (or always draw it).
- sim.portrayal.simple.LabelledPortrayal2D adds a text label the object. LabelledPortrayals can be set to to only add the label when the object is selected (or always draw it). LabelledPortrayals can have a fixed label or one based on some value of the object which changes as you like.
- sim.portrayal.simple.FacetedPortrayal2D has more than one subsidiary SimplePortrayal, and changes which SimplePortrayal is in charge based on the current state of the object. Particularly useful for doing simple animations.
- sim.portrayal.simple.MovablePortrayal2D allows you to drag and move the object.
- sim.portrayal.simple.OrientedPortrayal2D adds an orientation marker to the object to demonstrate what "direction" it's "pointing".
- sim.portrayal.simple.TrailedPortrayal2D adds a fading-out trail to the object so you can see the route it's taken.
- sim.portrayal.simple.TransformedPortrayal2D scales, rotates, or translates the object with respect to the underlying portrayal.

Notice the theme? All wrapper portrayals end in "ed".

In the code below we wrap the OvalPortrayal2D in not one, not two, but *three* wrapper portrayals: MovablePortrayal2D, CircledPortrayal2D, and LabelledPortrayal2D. It's easy:

```
import sim.portrayal.network.*;
import sim.portrayal.continuous.*;
import sim.engine.*;
import sim.display.*;
import sim.portrayal.simple.*;
import sim.portrayal.*;
import javax.swing.*;
import java.awt.Color;
import java.awt.*;

public class StudentsWithUI extends GUIState
    {
    public Display2D display;
    public JFrame displayFrame;

    ContinuousPortrayal2D yardPortrayal = new ContinuousPortrayal2D();
    NetworkPortrayal2D buddiesPortrayal = new NetworkPortrayal2D();

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

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

    public static String getName() { return "Student Schoolyard Cliques";  }

    public Object getSimulationInspectedObject() { return state; }

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

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

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

    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 MovablePortrayal2D(
                new CircledPortrayal2D(
                    new LabelledPortrayal2D(
                        new OvalPortrayal2D()
                            {
                            public void draw(Object object, Graphics2D graphics, DrawInfo2D info)
                                {
                                Student student = (Student)object;

                                int agitationShade = (int) (student.getAgitation() * 255 / 10.0);
                                if (agitationShade > 255) agitationShade = 255;
                               paint = new Color(agitationShade, 0, 255 - agitationShade);
                                super.draw(object, graphics, info);
                                }
                            },
                        5.0, null, Color.black, true),
                    0, 5.0, Color.green, true)));

        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();
        }

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

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

        displayFrame = display.createFrame();
        displayFrame.setTitle("Schoolyard Display");
        c.registerFrame(displayFrame);   // register the frame so it appears in the "Display" list
        displayFrame.setVisible(true);
        display.attach( buddiesPortrayal, "Buddies" );
        display.attach( yardPortrayal, "Yard" );
        }

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

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

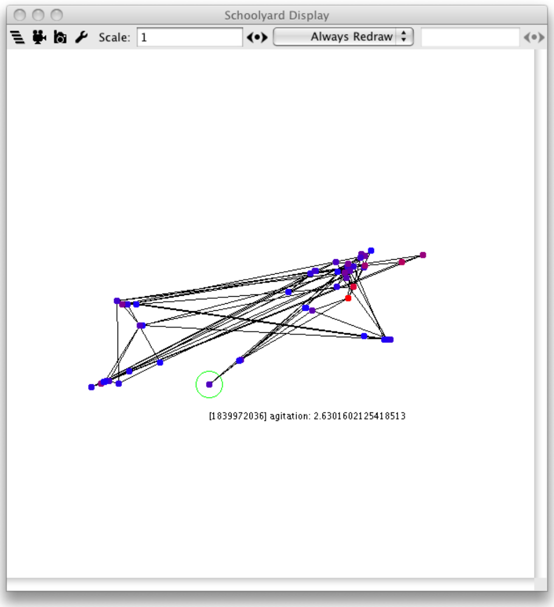
How do I keep certain objects from being moved but allow others? Or constrain movement?

Have the objects implement the sim.portrayal.Fixed2D interface, which gives them control over how they're moved.

Notice how we've inserted the portrayals "wrapped" around the basic OvalPortrayal2D. MovablePortrayal2D is straightforward and needs no explanation.

CircledPortrayal2D and LabelledPortrayal2D are set up to scale out five times the standard size of a SimplePortrayal2D and to only draw when the object is selected. The LabelledPortrayal is drawing the text in black and the CircledPortrayal in green. Importantly, the null value passed into the LabelledPortrayal2D tells it that no label is being provided: instead it must ask the underlying object what label to use (by calling toString()). At the moment, this returns the same ugly default name that's showing up in the Inspector list (such as "Student@5c76458f"). Let's keep the unique number identifier but add information about current agitation:

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

public class Student implements Steppable
    {
    public static final double MAX_FORCE = 3.0;

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

    public String toString() { return "[" + System.identityHashCode(this) + "] agitation: " + getAgitation(); }

    public void step(SimState state)
        {
        Students students = (Students) state;
        Continuous2D yard = students.yard;

        Double2D me = students.yard.getObjectLocation(this);

        MutableDouble2D sumForces = new MutableDouble2D();

        friendsClose = enemiesCloser = 0.0;

        // 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);
        int len = out.size();
        for(int buddy = 0 ; buddy < len; buddy++)
            {
            Edge e = (Edge)(out.get(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));

            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.resize(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.resize(0.0);
                else if (forceVector.length() > 0)
                    forceVector.resize(MAX_FORCE - forceVector.length());  // invert the distance
                enemiesCloser += forceVector.length();
                }
            sumForces.addIn(forceVector);
           }

        // add in a vector to the "teacher" -- 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));
        }
    }
```



[1839972036] agitation: 2.6301602125418513

Compile and Run

```
javac StudentsWithUI.java Students.java Student.java
java StudentsWithUI
```

As can be seen in the Display to the right, when you click (once) on a node, it is circled in green and given a useful label; any other nodes you happened to hit will also be selected. If you click elsewhere, the original node is deselected. And you can now drag objects and move them around the environment. Try it!

Selection and dragging works for continuous environments and certain gridworld scenarios. At present you can't select or drag edges.

Selection is presently somewhat primitive: you can't band-select (create a rectangle and select all the elements in it) nor hold down the shift key and select/deselect objects selectively. And though you can select more than one object at a time, you can't present move more than one object at a time. Perhaps in a later version.

2.11 Add an Anonymous Agent

Let's add an additional agent which, if used, initially sets the randomness to a high value and gradually decreases it. The user can turn the agent off in the model inspector. The point of this exercise is twofold. First, it demonstrates the use of an **anonymous class** to add a single agent to the schedule (a common practice). Second, it introduces the notion of an **explicit ordering** among agents scheduled at the exact same timestep.

MASON uses anonymous classes a lot, so if you're not familiar with them, they're worth explaining. An anonymous class is simply a class that has no name at all.[15] Instead, the anonymous class exists solely to be the subclass of some other class, or implement some interface, and create a single instance, then go away.

The pattern for anonymous classes is:

```
new SuperclassOrInterface()
    {
    methods and instance variables go here
    }
```

This pattern is an *expression* which creates a single expression, and you can assign it to a variable or whatnot. For example, we might create a subclass of javax.swing.JPanel which overrides the method paintComponent(...):

```
JPanel myJPanel = new JPanel()
    {
    public void paintComponent(Graphics g) { g.setColor(Color.black); g.drawLine(0,0,100,100); }
    };
```

[15]Well, technically *all* Java classes have names. But since you didn't state one, the compiler is free to make one up, which it will. javac will probably name it Foo$*number*, for some random number, if the outer class in which it is declared is named Foo.

Crucially, anonymous classes can access both the instance variables of the classes in which they are declared, and more interestingly, any **final local variables** in the environment in which they were created. For example, we could create a method which generates JPanels customized to your desired color:

```
public JPanel makeJPanel(final Color color)
    {
    return new JPanel()
        {
        public void paintComponent(Graphics g) { g.setColor(color); g.drawLine(0,0,100,100); }
        };
    }
```

This creates new JPanel subclasses on the fly with custom paintComponent(...) methods, produces single instances of them, and returns those instances. Nifty.

Notice that color had to be declared **final**. This isn't the case for outer instance variables used by anonymous classes, but it is the case for local variables and method parameters.[16]

An anonymous class will save us the tedium of creating a new file for our class, and various casts or constructors. So we're going with it.[17] We'll schedule the agent differently than the other agents: it'll occur at the same timestep as the others, but it will always be stepped *after* the other agents. We do this by specifying an ordering of 1 for the agent. The default ordering value for agents is 0. When the schedule has extracted all the agents for a given timestep and is ready to step them, it first sorts them by their ordering (lower values first). It then breaks ties with random shuffling. The method we'll use is scheduleRepeating(*time*, *ordering*, *Steppable*).

So enough talking, here's the code. See if you can make sense of it. Note that for all the discussion above, our anonymous agent uses an instance variable of the outer class (tempering) and so doesn't need to have it declared final. We could have also used isTempering() instead of tempering — anonymous classes have access to the methods of their outer instances.

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

public class Students extends SimState
    {
    public Continuous2D yard = new Continuous2D(1.0,100,100);
```

[16]Why is this the case? Beats me. Any decent modern language contains **closures** — non-final outer local variables usable by anonymous classes (or in many other languages, anonymous functions). Closures are very useful, but require cleverness in compilation. Sun's apparently not very clever. The amazing thing is that you can hack the same thing with a work-around: instead of making a local variable final int foo = 4;, you can make the variable final int[] foo = new int[] {4}; Then you can modify foo, from within the inner class, or more correctly, you can modify the value stored inside the array even if you can't change the array. Why Sun didn't just bake this into the compiler, instead of requiring a hack work-around, is utterly beyond me.

[17]Anonymous classes also make it possible to create an agent which is scheduled *twice* on the schedule, with different methods called each time, even though there's only *one* step(...) method. If you have an agent with methods like this:

```
public class MyAgent
  {
  public void doThis(SimState state) { ...  }
  public void doThat(SimState state) { ...  }
  }
```

... you can just say:

```
final MyAgent agent = ...  // notice that the variable is declared final
schedule.scheduleRepeating(new Steppable() { public void step(SimState state) { agent.doThis(state); }});
schedule.scheduleRepeating(new Steppable() { public void step(SimState state) { agent.doThat(state); }});
```

... or whatever.

```
public double TEMPERING_CUT_DOWN = 0.99;
public double TEMPERING_INITIAL_RANDOM_MULTIPLIER = 10.0;
public boolean tempering = true;
public boolean isTempering() { return tempering; }
public void setTempering(boolean val) { tempering = val; }

public int numStudents = 50;

double forceToSchoolMultiplier = 0.01;
double randomMultiplier = 0.1;

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); }

public double[] getAgitationDistribution()
    {
    Bag students = buddies.getAllNodes();
    double[] distro = new double[students.numObjs];
    int len = students.size();
    for(int i = 0; i < len; i++)
        distro[i] = ((Student)(students.get(i))).getAgitation();
    return distro;
    }

public Network buddies = new Network(false);

public Students(long seed)
    {
    super(seed);
    }

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

    // add the tempering agent
    if (tempering)
        {
        randomMultiplier = TEMPERING_INITIAL_RANDOM_MULTIPLIER;
        schedule.scheduleRepeating(new Steppable()
             {
             public void step(SimState state) { if (tempering) randomMultiplier *= TEMPERING_CUT_DOWN; }
             });
        }

    // 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);
        }

    // define like/dislike relationships
    Bag students = buddies.getAllNodes();
    for(int i = 0; i < students.size(); i++)
        {
        Object student = students.get(i);

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

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

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

Compile and Run

```
javac StudentsWithUI.java Students.java Student.java
java StudentsWithUI
```

Notice that now the agents start off jittery but calm down gradually. That's the work of our tempering agent. Also notice that you can turn off the effect in the *Model* tab under the property *Tempering*.

2.12 Checkpoint the Simulation

One of MASON's hallmarks is its ability to do **checkpointing**. By this I mean the ability to save out the state of the simulation, mid-run, to a file on the disk. The simulation can restart from this checkpoint even if it's on a different machine, or under visualization versus running on the command line, or even under a *different* visualization. Let's see how that works.

Generate Some Checkpoints Run MASON in the following way:

```
java Students -docheckpoint 100000

MASON Version 15.  For further options, try adding ' -help' at end.
Job: 0 Seed: 1293662957282
Starting Students
Steps: 25000 Time: 24999 Rate: 18,628.91207
Steps: 50000 Time: 49999 Rate: 25,562.37219
```

```
Steps: 75000 Time: 74999 Rate: 25,536.26149
Steps: 100000 Time: 99999 Rate: 25,510.20408
Checkpointing to file: 100000.0.Students.checkpoint
Steps: 125000 Time: 124999 Rate: 16,869.09582
Steps: 150000 Time: 149999 Rate: 19,888.62371
Steps: 175000 Time: 174999 Rate: 19,669.55153
Steps: 200000 Time: 199999 Rate: 19,888.62371
Checkpointing to file: 200000.0.Students.checkpoint
Steps: 225000 Time: 224999 Rate: 19,215.9877
Steps: 250000 Time: 249999 Rate: 19,700.55162
```

... etc. ...

Notice that MASON writes out a checkpoint every 100000 steps as requested. What use are these? There are a lot of uses.

For example, imagine that you've been running your simulation on a back-end supercomputer server for quite some time, and your job gets killed by the system administrator. You can just go back to the most recent checkpoint and do this:

```
java Students -checkpoint 200000.0.Students.checkpoint -docheckpoint 100000

MASON Version 15.  For further options, try adding ' -help' at end.
Loading from checkpoint 200000.0.Students.checkpoint
Recovered job: 0 Seed: 1293662957282
Steps: 225000 Time: 224999 Rate: 17,730.49645
Steps: 250000 Time: 249999 Rate: 24,154.58937
Steps: 275000 Time: 274999 Rate: 24,826.21648
Steps: 300000 Time: 299999 Rate: 25,100.40161
Checkpointing to file: 300000.0.Students.checkpoint
Steps: 325000 Time: 324999 Rate: 18,968.13354
Steps: 350000 Time: 349999 Rate: 19,685.03937
```

... etc. ...

MASON started up right where it left off as if nothing happened. You can also load the checkpoint in the GUI:

- Execute `java StudentsWithUI`
- Select *Open...* from the *File* menu File
- Select the checkpoint.
- The simulation loads the checkpoint and waits for you to unpause and continue it.

You can also save out a checkpoint from the GUI:

- Select *Save As...* from the *File* menu File
- Save out the checkpoint.

This saved-out checkpoint is just like any other: you could start it up again from the command line on your back-end machine. Or you could load it under a different GUI you've constructed.[18]

[18]Interested in seeing the transfer of a checkpoint between different GUI approaches? Try the examples sim.app.heatbugs.HeatBugsWithUI versus sim.app.heatbugs3d.HeatBugs3DWithUI, both of which use the same exact model.

A Note on Serialization MASON's checkpointing uses Java Serialization to do its magic, and Java Serialization is a fickle thing. To work properly, every object in the model (agents on the schedule, objects stored in fields, etc.) must implement the java.io.Serializable interface. That's one of several reasons why you shouldn't put GUI objects on the Schedule: generally speaking Java's GUI widgets are not serializable.

One gotcha in Serialization is making sure that all inner classes and anonymous classes are Serializable, as well as their outer classes. If your anonymous class is of type Steppable and thus an agent, for example, you're fine (Steppable is Serializable). But make certain.

Another gotcha lies in the generation of so-called *serial version UID values*, essentially hashes of classes which Java uses to determine whether the class declaration in the serialized file is compatible with the classes is using. Different Java implementations produce these hashes differently. If you're using the same Java VM, compiler, and OS, you're fine. But if you move class files from one machine to another (for example), there's no guarantee that the UID values will match. This is particularly problematic for inner classes, anonymous classes, and outer classes of inner classes.

Much of this is because Java's approach to UIDs has proven historically misguided. The solution is easy: rather than let Java generate the UID value, you declare it yourself. I suggest declaring all UID values to 1L (a common practice) to bypass them entirely. You should declare the following instance variable in all inner classes, all anonymous classes, all outer classes of inner classes, and (for good measure) all regular classes that you want to serialize, basically anything that's part of your model:

```
private static final long serialVersionUID = 1L;
```

In our case, this instance variable should be added to the following classes:

- Student
- Students
- The anonymous class we declared as part of Students which decreased the randomness each timestep.

2.13 Add a Description

MASON has a simple built-in HTML viewer which lets you decorate your simulation with some descriptive text. This text will appear in two places:

- Under the **About Tab** About
- In the pop-up simulation-chooser window which appears when you choose *New Simulation...* from the **File Menu** File

Is there any way to go directly to the simulation-chooser window when I start MASON?

Of course. Run MASON like this:

```
java sim.display.Console
```

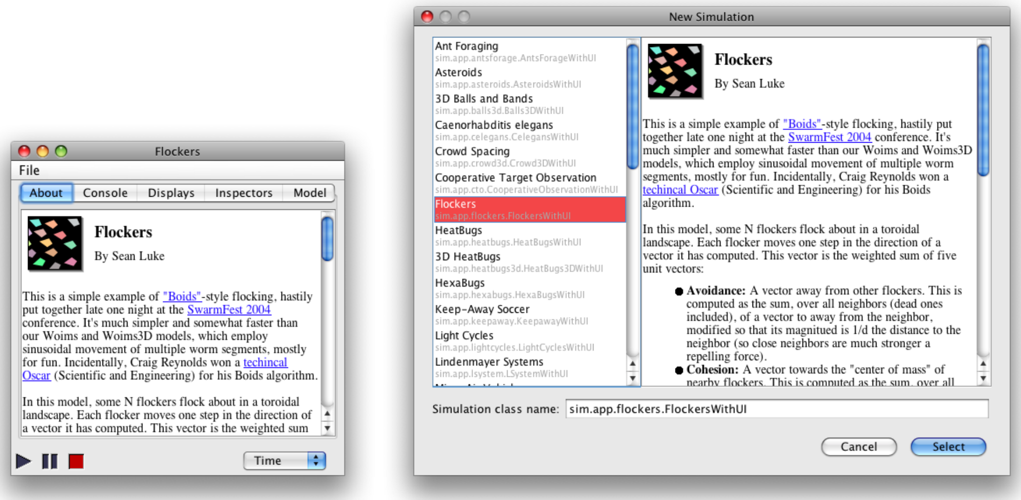
This is also how MASON is started if you use any of the scripts in the mason/start directory.

If you hit the "Cancel" button when firing MASON up this way, the program will simply quit.

These two situations are shown in Figure 2.1. MASON allows you to provide an HTML file, a String description (which may or may not be HTML), or a URL.

It's very easy to add an HTML file. Simply create a file called index.html located right next to the class file of your GUIState subclass. MASON will load the file and display it when your simulation is fired up.

For example, create the following file, named index.html, and save it right next to the file StudentsWithUI.class:



Flockers

By Sean Luke

This is a simple example of "Boids"-style flocking, hastily put together late one night at the SwarmFest 2004 conference. It's much simpler and somewhat faster than our Woims and Woims3D models, which employ sinusoidal movement of multiple worm segments, mostly for fun. Incidentally, Craig Reynolds won a techincal Oscar (Scientific and Engineering) for his Boids algorithm.

In this model, some N flockers flock about in a toroidal landscape. Each flocker moves one step in the direction of a vector it has computed. This vector is the weighted sum of five unit vectors:

- **Avoidance:** A vector away from other flockers. This is computed as the sum, over all neighbors (dead ones included), of a vector to away from the neighbor, modified so that its magnitued is 1/d the distance to the neighbor (so close neighbors are much stronger a repelling force).
- **Cohesion:** A vector towards the "center of mass" of nearby flockers. This is computed as the sum, over all

Ant Foraging
sim.app.antsforage.AntsForageWithUI
Asteroids
sim.app.asteroids.AsteroidsWithUI
3D Balls and Bands
sim.app.balls3d.Balls3DWithUI
Caenorhabditis elegans
sim.app.celegans.CelegansWithUI
Crowd Spacing
sim.app.crowd3d.Crowd3DWithUI
Cooperative Target Observation
sim.app.cto.CooperativeObservationWithUI
Flockers
sim.app.flockers.FlockersWithUI
HeatBugs
sim.app.heatbugs.HeatBugsWithUI
3D HeatBugs
sim.app.heatbugs3d.HeatBugs3DWithUI
HexaBugs
sim.app.hexabugs.HexaBugsWithUI
Keep-Away Soccer
sim.app.keepaway.KeepawayWithUI
Light Cycles
sim.app.lightcycles.LightCyclesWithUI
Lindenmayer Systems
sim.app.lsystem.LSystemWithUI

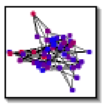
Simulation class name: sim.app.flockers.FlockersWithUI

Figure 2.1 The Console and the Simulation-Chooser Window, both showing the HTML descriptive text of a simulation.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<html>
<head>
</head>
<body>

<table border=0 cellspacing=0 cellpadding=6>
<tr>
<td valign=top>
        <img src="icon.png">
<td valign=top>
        <h2>Student Cliques</h2>
        <b>Tutorial</b>
</table>

<p>
A fictional model of student clique formation on playgrounds.  Students form a social network of varying
degrees of mutual friendship or dislike.  Once let out of school, students try to move towards friends and
away from enemies.  Additional forces tug on students to keep them near near the schoolhouse (the center
of the yard), and to add some randomness.  Students change from red to blue as they become happier with
their situation.

</body>
</html>
```

Notice that the DOCTYPE indicates HTML 3.2, an old and simple version of HTML with limited CSS support. This is as far as Java's GUI facilities go. You do not need to provide the DOCTYPE: but you do need to be aware that 3.2 is all MASON can display.

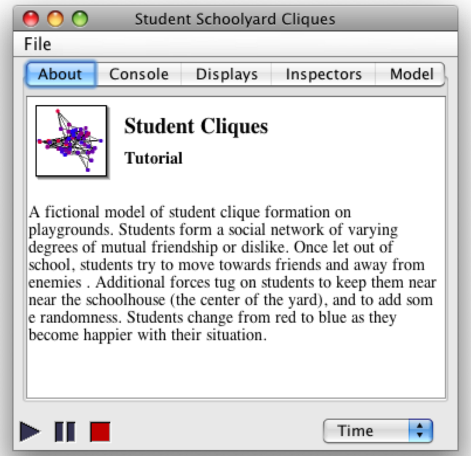
The HTML file above requires an image (icon.png) to provide sort of the "standard look" of MASON's built-in demo applications. The one I use is shown at right.



Run the Simulation Now fire up the simulation again:

```
java StudentsWithUI
```

... and you'll find that the Console now looks like the Figure on the right.



Student Cliques

Tutorial

A fictional model of student clique formation on playgrounds. Students form a social network of varying degrees of mutual friendship or dislike. Once let out of school, students try to move towards friends and away from enemies . Additional forces tug on students to keep them near near the schoolhouse (the center of the yard), and to add som e randomness. Students change from red to blue as they become happier with their situation.

Alternatively you can embed your own String, or a URL to an existing web page, directly in your GUIState subclass. For example, if you didn't provide the index.html file above, you could add some simple text in the method getInfo(). Specifically, add the following method to your StudentsWithUI.java file:

```
public static Object getInfo()
    {
    return "<h2>This is the tutorial</h2>" +
        "<p>Pretty nifty!";
    }
```

With this static method in place, MASON will use its value rather than hunting for an index.html file. But there's a gotcha.

Wait, getInfo() is a static method. You can't override static methods!

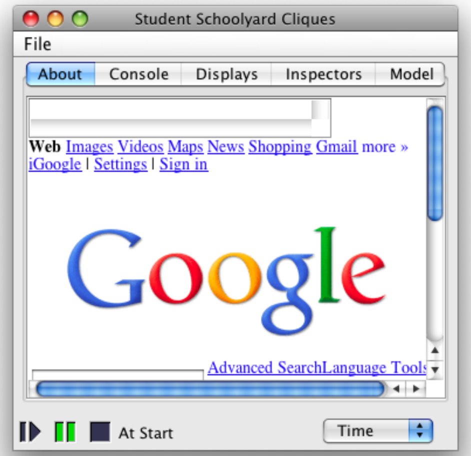
Correct. But MASON actually looks for the method using Java's reflection facility, so it's all good. MASON does this so it can find and display the proper String or URL without having to instantiate a simulation.

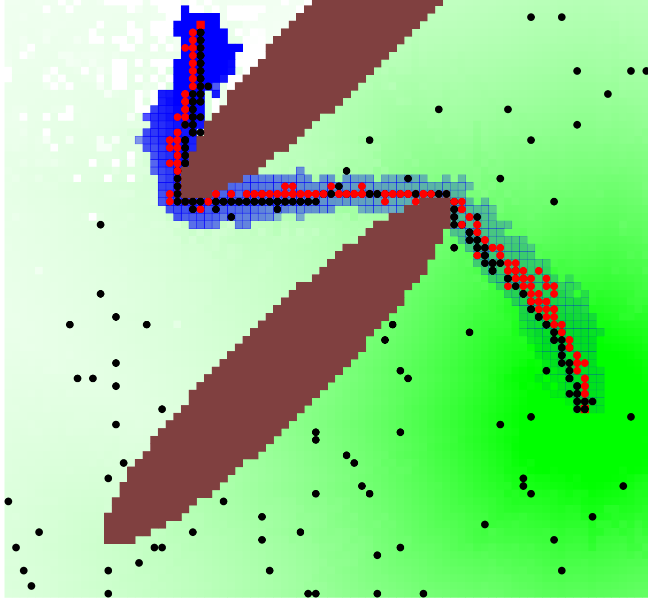
Java looks for image files referenced from an HTML file by looking relative to the HTML file itself. Unfortunately if you override the getInfo() method, Java doesn't know where to look for images. For example, the icon.png image would be broken. So for the time being, you can't embed images or CSS files with relative URLs, or use relative URL links to other HTML files, if you override getInfo() to return a String. In general, I suggest using the index.html file instead, unless you're just trying to display some simple text.

One last approach you can take is to return not a String but a URL. For example, you could override getInfo() like this:

```
public static Object getInfo()
    {
    try { return new java.net.URL("http://google.com/"); }
    catch (java.net.MalformedURLException e) { return "Oops"; }
    }
```

When you do this, the Console window will look like it does at right. Notice that the Google.com page is a bit discombobulated: it's assuming a more recent version of HTML than 3.2. If you try it, you'll also find out that the search buttons don't work, though the links do. I believe form submission in general isn't available in Java's basic HTML facilities (nor is JavaScript).


Student Schoolyard Cliques
File
About
Console
Displays
Inspectors
Model
Web Images Videos Maps News Shopping Gmail more »
iGoogle | Settings | Sign in
Google
Advanced SearchLanguage Tools
At Start
Time



Chapter 3

Basic Utility Classes

MASON has a large number of utility classes which can be used independently of the toolkit. These classes fall in six packages:

- ec.util The Mersenne Twister random number generator
- sim.util Basic utility classes
- sim.util.gui Graphical interface utility classes
- sim.util.media Utility classes for generating various media (movies, pictures)
- sim.util.media.chart Utility classes for generating charts
- sim.util.distribution Utility classes for creating distributions

This chapter concentrates solely on ec.util, sim.util, and sim.util.distribution. Many classes in the core simulation code rely on these packages (well the first two anyway), so it's important to cover them.

3.1 The Random Number Generator and Distributions

The ec.util package contains a single class called ec.util.MersenneTwisterFast. This is an efficient implementation of the MT199937 Mersenne Twister algorithm by Makoto Matsumoto and Takuji Nishimura. The Mersenne Twister is a well-regarded random number generator with an ultra-long period and high quality statistical properties. It's now the standard generator for R, Maple, MATLAB, Python, Ruby, several major implementations of Common Lisp (such as SBCL and CMUCL), and is part of the PHP library. MASON uses Mersenne Twister as its only random number generator; it's possible to substitute another, but you'd need to subclass the MersenneTwisterFast class.

Why not use java.util.Random? Because java.util.Random is *highly non-random*. It is unquestionably inappropriate for use in a high-quality simulator. Never use it, nor should you ever call Math.random() (which also uses it).

Where did ec.util and ec.util.MersenneTwisterFast come from? This class and package came from ECJ, an evolutionary computation toolkit I developed in 1998. There are actually two versions of Mersenne Twister: the class ec.util.MersenneTwister and the class ec.util.MersenneTwisterFast. The former is a drop-in subclass replacement for java.util.Random, and is threadsafe. The latter is not threadsafe, is *not* a subclass of java.util.Random, and has many methods (perhaps nowadays unnecessarily) heavily inlined, and as a result is significantly faster. MersenneTwisterFast is the only class provided with and used by MASON.

Any gotchas? Yes. The standard MT199937 seeding algorithm uses one of Donald Knuth's plain-jane linear congruential generators to fill the Mersenne Twister's arrays. This means that for a short while the algorithm will initially be outputting a (very slightly) lower quality random number stream until it warms up. After about 625 calls to the generator, it'll be warmed up sufficiently. You probably will never notice or care, but if you wanted to be extra extra paranoid, you could call nextInt() 1300 times or so when your model is initially started. Perhaps in the future we'll do that for you.

MersenneTwisterFast has identical methods to java.util.Random, plus one or two more for good measure. They should look familiar to you:

ec.util.MersenneTwisterFast Constructor Methods

public MersenneTwisterFast(long seed)
: Seeds the random number generator. Note that only the first 32 bits of the seed are used.

public MersenneTwisterFast()
: Seeds the random number generator using the current time in milliseconds.

public MersenneTwisterFast(int[] vals)
: Seeds the random number generator using the given array. Only the first 624 integers in the array are used. If the array is shorter than 624, then the integers are repeatedly used in a wrap-around fashion (not recommended). The integers can be anything, but you should avoid too many zeros. MASON does not call this method.

ec.util.MersenneTwisterFast Methods

public void setSeed(long seed)
: Seeds the random number generator. Note that only the first 32 bits of the seed are used.

public void setSeed(int[] vals)
: Seeds the random number generator using the given array. Only the first 624 integers in the array are used. If the array is shorter than 624, then the integers are repeatedly used in a wrap-around fashion (not recommended). The integers can be anything, but you should avoid too many zeros.

public double nextDouble()
: Returns a random double drawn in the half-open interval from [0.0, 1.0). That is, 0.0 may be drawn but 1.0 will never be drawn.

public float nextFloat()
: Returns a random float drawn in the half-open interval from [0.0f, 1.0f). That is, 0.0f may be drawn but 1.0f will never be drawn.

public double nextGaussian()
: Returns a random double drawn from the standard normal Gaussian distribution (that is, a Gaussian distribution with a mean of 0 and a standard deviation of 1).

public long nextLong()
: Returns a random long.

public long nextLong(long n)
: Returns a random long drawn from between 0 to $n - 1$ inclusive.

public int nextInt()
: Returns a random integer.

public int nextInt(int n)
: Returns a random integer drawn from between 0 to $n - 1$ inclusive.

public short nextShort()
: Returns a random short.

public char nextChar()
Returns a random character.

public byte nextByte()
Returns a random byte.

public void nextBytes(byte[] bytes)
Fills the given array with random bytes.

public boolean nextBoolean()
Returns a random boolean.

public boolean nextBoolean(float probability)
Returns a random boolean which is true with the given probability, else false. Note that you must carefully pass in a *float* here, else it'll use the *double* version below (which is twice as slow).

public boolean nextBoolean(double probability)
Returns a random boolean which is true with the given probability, else false.

public Object clone()
Clones the generator.

public boolean stateEquals(Object o)
Returns true if the given Object is a MersenneTwisterFast and if its internal state is identical to this one.

public void writeState(DataOutputStream stream)
Writes the state to a stream.

public void readState(DataInputStream stream)
Reads the state from a stream as written by writeState(...).

public static void main(String[] args)
Performs a test of the code.

The MersenneTwisterFast class, like java.util.Random, provides random numbers from only two floating-point distributions: uniform and Gaussian. What if you need numbers drawn from other distributions? MASON provides a variety of distributions in the sim.util.distribution package. This package is a modified version of the distributions from the COLT/JET library.[1] The modifications remove certain misfeatures of the library which make it difficult to serialize, unify a few utility classes, and most importantly, replace COLT's random number generator data types with Mersenne Twister. You can just plug in your model's MersenneTwisterFast random number generator and pop out random numbers under various distributions.

Because it's separate from the MASON core proper (and a bit new), I don't describe this package in detail, but it should be fairly straightforward. **Warning:** the port of these classes from COLT to MASON is new and has not been tested much.

There are two kinds of distributions: (1) distributions which require their own instances and (2) distributions which just require function calls. The first group each have their own classes, and you must create instances of them. They include:

• Beta	sim.util.distribution.Beta
• Binomial	sim.util.distribution.Binomial
• Breit-Wigner (Lorentz)	sim.util.distribution.BreitWigner
• Mean-Square Breit-Wigner	sim.util.distribution.BreitWignerMeanSquare
• Chi-Square	sim.util.distribution.ChiSquare

[1]http://acs.lbl.gov/software/colt/

• Empirical	sim.util.distribution.Empirical
• Discrete Emperical	sim.util.distribution.EmpiricalWalker
• Exponential	sim.util.distribution.Exponential
• Exponential Power	sim.util.distribution.ExponentialPower
• Gamma	sim.util.distribution.Gamma
• Hyperbolic	sim.util.distribution.Hyperbolic
• Hyper-Geometric	sim.util.distribution.HyperGeometric
• Logarithmic	sim.util.distribution.Logarithmic
• Negative Binomial	sim.util.distribution.NegativeBinomial
• Normal (Gaussian) — not very useful as it's built into Mersenne Twister	sim.util.distribution.Normal
• Poisson (two kinds)	sim.util.distribution.Poisson and sim.util.distribution.PoissonSlow
• Student's T	sim.util.distribution.StudentT
• Uniform — again, not particularly useful	sim.util.distribution.Uniform
• Von Mises	sim.util.distribution.VonMises
• Zeta	sim.util.distribution.Zeta

The second group are just function calls from the sim.util.distribution.Distributions class:

- Burr (various kinds)
- Cauchy
- Erlang
- Geometric
- Lambda
- Laplace,
- Logistic
- Power-Law
- Triangular
- Weibull
- Zipf

3.2 Coordinate Wrapper Classes

MASON has a large number of consistent wrapper classes for 2D and 3D coordinates. Java also has classes for 2D and 3D coordinates: for example, the java.awt.Point class wraps two integers (x and y), and the java.awt.Point2D.Double class wraps two doubles. However Java's classes have severe deficiencies. The most serious problem with them is that they are broken when used as keys in hash tables. Sun in its infinite wisdom made a serious error in how it handles hashcode generation and equality testing in these classes, and so if you use them in hash tables, you will regret it.[2]

To fix this, MASON has its own coordinate classes, in two forms, **immutable** and **mutable**. Immutable instances may not have their values changed once set during instantiation. The immutable classes work well as keys in hash tables. Mutable instances can have their values changed freely. MASON's mutable classes have the same problem as Sun's classes, but they are at least consistent, code-wise with the immutable classes. The mutable classes also have many more mathematical operations available.

The classes are:

- sim.util.Int2D and sim.util.MutableInt2D
- sim.util.Double2D and sim.util.MutableDouble2D
- sim.util.Int3D and sim.util.MutableInt3D
- sim.util.Double3D and sim.util.MutableDouble3D

Though the classes have a zillion utility methods, in fact they are **very simple wrappers over just two or three variables.** Each of these classes has the following variables, which you can read and write directly (or at least read: in the immutable classes they're final):

```
int (or double) x;
int (or double) y;
int (or double) z; // only in the 3D classes
```

You should access these variables with abandon: they're designed for it. Additionally the classes have many accessor methods. We will show methods of the 2D versions of the classes below: the 3D classes are nearly identical (minus a few inappropriate methods here and there). First up is Int2D:

sim.util.Int2D Constructor Methods

public Int2D()
Creates an Int2D with x=0 and y=0.

public Int2D(int x, int y)
Creates an Int2D with the given x and y values.

public Int2D(java.awt.Point p)
Creates an Int2D with x and y values of the given Point.

public Int2D(MutableInt2D p)
Creates an Int2D with x and y values of the given MutableInt2D.

sim.util.Int2D Methods

public int getX()
Returns the x value.

[2]Specifically: the classes hash by value rather than by pointer, yet they can have their values changed. So if you hash an object keyed with a Point2D.Double, then change the values of the Point2D.Double, the object is lost in the hash table.

public int getY()
: Returns the y value.

public java.awt.geom.Point2D.Double toPoint2D()
: Builds a Point2D.Double with the current x and y values.

public java.awt.Point toPoint()
: Builds a Point with the current x and y values.

public String toString()
: Returns a String version of the Int2D.

public String toCoordinates()
: Returns a String version of the x and y values as coordinates in the form (x, y).

public int hashCode()
: Builds a hash code from the Int2D.

public boolean equals(Object obj)
: Returns true if the Int2D is equal to the other object in value. Int2D can be compared against other Int2D, MutableIn2D, Double2D, and MutableDouble2D objects.

public double distance(double x, double y)
: Returns the distance from the Int2D to the given coordinates.

public double distance(Int2D p)
: Returns the distance from the Int2D to the given coordinates.

public double distance(MutableInt2D p)
: Returns the distance from the Int2D to the given coordinates.

public double distance(Double2D p)
: Returns the distance from the Int2D to the given coordinates.

public double distance(MutableDouble2D p)
: Returns the distance from the Int2D to the given coordinates.

public double distance(java.awt.geom.Point2D p)
: Returns the distance from the Int2D to the given coordinates.

public double distanceSq(double x, double y)
: Returns the squared distance from the Int2D to the given coordinates. This is faster than computing the distance (it doesn't require a square root).

public double distanceSq(Int2D p)
: Returns the squared distance from the Int2D to the given coordinates. This is faster than computing the distance (it doesn't require a square root).

public double distanceSq(MutableInt2D p)
: Returns the squared distance from the Int2D to the given coordinates. This is faster than computing the distance (it doesn't require a square root).

public double distanceSq(Double2D p)
: Returns the squared distance from the Int2D to the given coordinates. This is faster than computing the distance (it doesn't require a square root).

public double distanceSq(MutableDouble2D p)
: Returns the squared distance from the Int2D to the given coordinates. This is faster than computing the distance (it doesn't require a square root).

public double distanceSq(java.awt.geom.Point2D p)
: Returns the squared distance from the Int2D to the given coordinates. This is faster than computing the distance (it doesn't require a square root).

public long manhattanDistance(int x, int y)
  Returns the manhattan distance from the Int2D to the given coordinates.

public long manhattanDistance(Int2D p)
  Returns the manhattan distance from the Int2D to the given coordinates.

public long manhattanDistance(MutableInt2D p)
  Returns the manhattan distance from the Int2D to the given coordinates.

The squared distance between two points $\langle x_1, y_1\rangle$ and $\langle x_2, y_2\rangle$ is simply $(x_1 - x_2)^2 + (y_1 - y_2)^2$. The distance is defined as $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. The manhattan distance is defined as $|x_1 - x_2| + |y_1 - y_2|$

MutableInt2D has all these methods and constructors, plus a few more:

Additional sim.util.MutableInt2D Constructor Methods

public MutableInt2D(Int2D)
  Creates a MutableInt2D with x and y values of the given Int2D.

Additional sim.util.MutableInt2D Methods

public void setX(int val)
  Sets the x value.

public void setY(int val)
  Sets the y value.

public void setTo(int x, int y)
  Sets the x and y values.

public void setTo(java.awt.Point p)
  Sets the x and y values to the given Point.

public void setTo(Int2D p)
  Sets the x and y values to the given Int2D.

public void setTo(MutableInt2D p)
  Sets the x and y values to the given MutableInt2D.

public Object clone()
  Returns a clone of the MutableInt2D.

Double2D is similar to Int2D, though you'll find a few more constructors and useful mathematics functions:

sim.util.Double2D Constructor Methods

public Double2D()
  Creates an Double2D with x=0 and y=0.

public Double2D(double x, double y)
  Creates an Double2D with the given x and y values.

public Double2D(java.awt.Point p)
  Creates an Double2D with x and y values of the given Point.

public Double2D(java.awt.geom.Point2D.Double p)
  Creates an Double2D with x and y values of the given Point2D.Double.

public Double2D(java.awt.geom.Point2D.Float p)
: Creates an Double2D with x and y values of the given Point2D.Float.

public Double2D(MutableDouble2D p)
: Creates an Double2D with x and y values of the given MutableDouble2D.

public Double2D(Int2D p)
: Creates an Double2D with x and y values of the given Int2D.

public Double2D(MutableInt2D p)
: Creates an Double2D with x and y values of the given MutableInt2D.

---

sim.util.Double2D Methods

public double getX()
: Returns the x value.

public double getY()
: Returns the y value.

public java.awt.geom.Point2D.Double toPoint2D()
: Builds a Point2D.Double with the current x and y values.

public String toString()
: Returns a String version of the Double2D.

public String toCoordinates()
: Returns a String version of the x and y values as coordinates in the form (x, y).

public int hashCode()
: Builds a hash code from the Double2D.

public boolean equals(Object obj)
: Returns true if the Double2D is equal to the other object in value. Double2D can be compared against Int2D, MutableIn2D, Double2D, and MutableDouble2D objects.

public double distance(double x, double y)
: Returns the distance from the Double2D to the given coordinates.

public double distance(Int2D p)
: Returns the distance from the Double2D to the given coordinates.

public double distance(MutableInt2D p)
: Returns the distance from the Double2D to the given coordinates.

public double distance(Double2D p)
: Returns the distance from the Double2D to the given coordinates.

public double distance(MutableDouble2D p)
: Returns the distance from the Double2D to the given coordinates.

public double distance(java.awt.geom.Point2D p)
: Returns the distance from the Double2D to the given coordinates.

public double distanceSq(double x, double y)
: Returns the squared distance from the Double2D to the given coordinates. This is faster than computing the distance (it doesn't require a square root).

public double distanceSq(Int2D p)
: Returns the squared distance from the Double2D to the given coordinates. This is faster than computing the distance (it doesn't require a square root).

public double distanceSq(MutableInt2D p)
: Returns the squared distance from the Double2D to the given coordinates. This is faster than computing the distance (it doesn't require a square root).

public double distanceSq(Double2D p)
: Returns the squared distance from the Double2D to the given coordinates. This is faster than computing the distance (it doesn't require a square root).

public double distanceSq(MutableDouble2D p)
: Returns the squared distance from the Double2D to the given coordinates. This is faster than computing the distance (it doesn't require a square root).

public double distanceSq(java.awt.geom.Point2D p)
: Returns the squared distance from the Double2D to the given coordinates. This is faster than computing the distance (it doesn't require a square root).

public double manhattanDistance(double x, double y)
: Returns the manhattan distance from the Double2D to the given coordinates.

public double manhattanDistance(Double2D p)
: Returns the manhattan distance from the Double2D to the given coordinates.

public double manhattanDistance(MutableDouble2D p)
: Returns the manhattan distance from the Double2D to the given coordinates.

public double manhattanDistance(Int2D p)
: Returns the manhattan distance from the Double2D to the given coordinates.

public double manhattanDistance(MutableInt2D p)
: Returns the manhattan distance from the Double2D to the given coordinates.

public double manhattanDistance(java.awt.geom.Point2D p)
: Returns the manhattan distance from the Double2D to the given coordinates.

public double angle()
: Returns the angle of the Double2D.

public double length()
: Returns the length of the Double2D.

public double lengthSq()
: Returns the squared length of the Double2D. This is less expensive than calling length(), as it doesn't involve a square root.

public double dot(Double2D other)
: Takes the dot product of this and the other Double2D.

public double perpDot(Double2D other)
: Takes the "perp dot product" (the 2D equivalent of the cross product) of this and the other Double2D.

public Double2D negate()
: Returns the negation of this Double2D.

public Double2D add(Double2D other)
: Adds the other Double2D and returns a new Double2D holding the result.

public Double2D subtract(Double2D other)
: Subtracts the other Double2D and returns a new Double2D holding the result.

public Double2D multiply(double scalar)
: Multiplies the Double2D against the scalar and returns a new Double2D holding the result.

public Double2D resize(double length)
Scales the Double2D to be the given length and returns a new Double2D holding the result.

public Double2D normalize()
Normalizes the Double2D and returns a new Double2D holding the result. If the Double2D is zero in length, an error is thrown.

public Double2D rotate(double theta)
Rotates the Double2D by the given radians and returns a new Double2D holding the result.

Example Usage Double2D is designed so that math operations produce new Double2D classes at each step. For example, let's say you wanted to cause an agent $\vec{a}$ to move away from enemies. If an enemy $e^{(i)}$ is close, it exerts a much higher force on the agent than if an enemy is far away. We could have the agent add up all the forces, then move in a constant speed in the other direction. Something along the lines of:

$$\vec{a} = \vec{a} + \delta \times \text{normalize}\left(\sum_i \frac{-1}{|\vec{e^{(i)}}|}\vec{e^{(i)}}\right)$$

We'd do it like this:

```
double delta = ...
Double2D agent = ...
Double2D[] enemies = ...
Double2D force = new Double2D(); // <0,0>
for(int i = 0; i < enemies.length; i++)
        force = force.add(enemies[i].multiply(-1.0 / enemies[i].length()));
agent = agent.add(force.normalize().multiply(delta)); // alternatively force.resize(delta)
```

Notice the chaining of operations in force.normalize().multiply(delta). Keep in mind that at every stage in these operations new Double2Ds are getting allocated, so this isn't amazingly efficient. If you're doing a lot of vector manipulation, you may instead wish to use a *mutable* version instead, which allows you to change certain vectors in-place:

Mutable Vectors The MutableDouble2D contains similar methods except for the add(), subtract(), multiply(), resize(), normalize(), and rotate() methods. Instead MutableDouble2D methods tend to modify the MutableDouble2D itself instead of returning a brand new one (that's the point of MutableDouble2D after all). Here are some of the different methods and constructors:

Additional sim.util.MutableDouble2D Constructor Methods

public MutableDouble2D(Double2D)
Creates a MutableDouble2D with x and y values of the given Double2D.

Additional sim.util.MutableDouble2D Methods

public void setX(double val)
Sets the x value.

public void setY(double val)
Sets the y value.

public void setTo(double x, double y)
Sets the x and y values.

public void setTo(java.awt.Point p)
: Sets the x and y values to the given Point.

public void setTo(Int2D p)
: Sets the x and y values to the given Int2D.

public void setTo(MutableInt2D p)
: Sets the x and y values to the given MutableInt2D.

public void setTo(Double2D p)
: Sets the x and y values to the given Double2D.

public void setTo(MutableDouble2D p)
: Sets the x and y values to the given MutableDouble2D.

public Object clone()
: Returns a clone of the MutableDouble2D.

public MutableDouble2D addIn(double x, double y)
: Modifies the MutableDouble2D to reflect adding in the other values. Returns the modified MutableDouble2D.

public MutableDouble2D addIn(Double2D other)
: Modifies the MutableDouble2D to reflect adding in the other values. Returns the modified MutableDouble2D.

public MutableDouble2D addIn(MutableDouble2D other)
: Modifies the MutableDouble2D to reflect adding in the other values. Returns the modified MutableDouble2D.

public MutableDouble2D add(MutableDouble2D other1, MutableDouble2D other2)
: Adds other1 and other2, setting the Mutable2D to their sum and returning it.

public MutableDouble2D add(Double2D other1, MutableDouble2D other2)
: Adds other1 and other2, setting the Mutable2D to their sum and returning it.

public MutableDouble2D add(MutableDouble2D other1, Double2D other2)
: Adds other1 and other2, setting the Mutable2D to their sum and returning it.

public MutableDouble2D subtractIn(Double2D other)
: Modifies the MutableDouble2D to reflect subtracting the other values from it. Returns the modified MutableDouble2D.

public MutableDouble2D subtractIn(MutableDouble2D other)
: Modifies the MutableDouble2D to reflect subtracting the other values from it. Returns the modified MutableDouble2D.

public MutableDouble2D subtract(MutableDouble2D other1, MutableDouble2D other2)
: Subtracts other2 from other1 setting the Mutable2D to their difference and returning it.

public MutableDouble2D subtract(Double2D other1, MutableDouble2D other2)
: Subtracts other2 from other1 setting the Mutable2D to their difference and returning it.

public MutableDouble2D subtract(MutableDouble2D other1, Double2D other2)
: Subtracts other2 from other1 setting the Mutable2D to their difference and returning it.

public MutableDouble2D multiplyIn(double scalar)
: Modifies the MutableDouble2D to reflect multiplying it by the scalar. Returns the modified MutableDouble2D.

public MutableDouble2D multiply(MutableDouble2D other, double scalar)
: Multiplies the other MutableDouble2D by the given scalar, setting this MutableDouble2D to the result. Returns the modified MutableDouble2D.

public MutableDouble2D multiply(Double2D other, double scalar)
: Multiplies the Double2D by the given scalar, setting this MutableDouble2D to the result. Returns the modified MutableDouble2D.

public MutableDouble2D resize(double length)
Scales the MutableDouble2D to be the given length, modifying it, and returns it.

public MutableDouble2D normalize()
Normalizes the MutableDouble2D , modifying it, and returns it. If the MutableDouble2D is zero in length, an error is thrown.

public MutableDouble2D rotate(double theta)
Rotates the MutableDouble2D by the given radians, modifying it, and returns it.

public MutableDouble2D negate()
Negates the MutableDouble2D, modifying it, and returns it.

public MutableDouble2D setToMinus(MutableDouble2D other)
Sets the MutableDouble2D to the negation of the other, and returns it.

public MutableDouble2D zero()
Sets the MutableDouble2D to $\langle 0, 0 \rangle$ and returns it.

public MutableDouble2D dup()
Clones the MutableDouble2D.

Whew! As you can see, the general philosophy in the immutable classes is to create new immutable instances as a result of mathematical operations; whereas the mutable classes tend to modify themselves to reflect the results.

The 3D versions of these are very similar to the 2D, minus certain math operations (like perpDot or rotate) which make no sense in 3D.

Example Usage Returning to our previous example:

$$\vec{a} = \vec{a} + \delta \times \text{normalize}\left(\sum_i \frac{-1}{|\vec{e^{(i)}}|}\vec{e^{(i)}}\right)$$

Using MutableDouble2D we could do it a bit more efficiently like this:

```
double delta = ...
Double2D agent = ...
Double2D[] enemies = ...
MutableDouble2D force = new MutableDouble2D(); // <0,0>
MutableDouble2D temp = new MutableDouble2D(); // <0,0>
for(int i = 0; i < enemies.length; i++)
        force.addIn(temp.multiply(enemies[i], -1.0 / enemies[i].length()));
agent = agent.add(force.normalize().multiply(delta)); // alternatively force.resize(delta)
```

Here instead of creating new Double2Ds for each math step, we store the results in a temporary variable, and ultimately modify the force variable. At the end we dump the results into a Double2D again (Double2D is used for hashing in Continuous Fields (see Section 6) and Mutable2D is not).

With modern garbage collection methods, allocating lots of Double2Ds is less of a concern, but it's still more than worthwhile to use MutableDouble2D in many situations.

3.3 Collections

MASON has four collections classes special to it:

- sim.util.Bag is an extensible array, quite similar to java.util.ArrayList.

- sim.util.IntBag is like a Bag, but it holds ints rather than Objects.
- sim.util.DoubleBag is like a Bag, but it holds doubles rather than Objects.
- sim.util.Heap is a binary heap, similar to java.util.PriorityQueue. Heap is discussed later in Section 4.3.2.1.

Why create a new ArrayList? sim.util.Bag was created because at the time most ArrayList operations were quite slow. For years ArrayList's get(), set(), and add() operations were not inlinable! In fact now they're inlinable only because of advances in Hotspot technology and not because of simple bug-fixes that could have been made in the class years ago (the bugs are still there).

Bag is an extensible array like ArrayList but with four major differences:

- It's a java.util.Collection but not a java.util.List most because implementing the List interface is a pain and not particularly useful.
- It's has a special version of the remove() method which is $O(1)$ where ArrayList's is $O(n)$. The method modifies the order of the elements in the Bag however.
- Bag does not (at present) support generics.
- You can access the elements in the underlying array directly (if you're careful).

This last item used to be a big deal: it enabled Bag to be up to five times faster than ArrayList. But no longer. As of Java 1.6, ArrayList has improved significantly in efficiency, bugs and all. So at some point it will make sense to shift from Bag to ArrayList simply to be less obtuse, if anything. But as MASON makes quite extensive use of Bag, for now we're sticking with it.

Bag's basic variables are public:

```
public Object[] objs;
public int numObjs;
```

objs is an array holding the objects in the Bag. The objects fill the positions objs[0] through objs[numObjs-1]. The objs array can be larger than numObjs in size: the remaining slots are not defined and should be ignored. At any time the objs array may be replaced with another one of larger or smaller size.

You can scan rapidly through all the elements in the Bag's array like this:

```
Object[] o = bag.objs;
int n = bag.numObjs;
for(int i = 0; i < n; i++) System.out.println(o[i]);
```

However nowadays you might as well do it in a more traditional fashion, if only to make it easier to migrate if and when we switch from Bag to ArrayList:

```
int n = bag.size();
for(int i = 0; i < n; i++) System.out.println(bag.get(i));
```

IntBag and DoubleBag These classes are very close to identical to Bag, except that instead of Objects, they store ints or doubles respectively. They are missing methods involving Collections, Comparators, and Iterators, but otherwise follow the identical template as Bag. We won't write them out here — too redundant. Refer to the Bag methods below.

Bag Bag has all the standard constructors and methods you'd expect in an extensible array:

sim.util.Bag Constructor Methods

public Bag()
  Creates an empty Bag.

public Bag(int capacity)
  Creates an empty Bag with an initial capacity.

public Bag(Bag other)
  Creates a Bag containing the same elements as another Bag (and in the same initial order).

public Bag(Object[] other)
  Creates a Bag containing the same elements as an array (and in the same initial order).

public Bag(Collection other)
  Creates a Bag containing the same elements as a Collection.

sim.util.Bag Methods

public Object get(int index)
  Returns the object located at *index* in the Bag.

public Object getValue(int index)
  Returns the object located at *index* in the Bag (identical to get(...)).

public Object set(int index, Object element)
  Sets the slot *index* in the Bag to hold *element*. Returns the old element in that slot.

public Object setValue(int index, Object element)
  Sets the slot *index* in the Bag to hold *element*. Returns the old element in that slot. (identical to set(...).

public boolean add(Object obj)
  Appends the given object to the end of the Bag (same as push(...)). Always returns true.

public boolean push(Object obj)
  Appends the given object to the end of the Bag. Always returns true.

public Object pop()
  Removes and returns the last element in the Bag, else null if there is none.

public Object top()
  Returns the last element in the Bag, else null if there is none.

public int size()
  Returns the number of elements in the Bag.

public boolean isEmpty()
  Returns true if the number of elements in the Bag is zero.

public boolean addAll(Bag other)
  Adds all the other elements to the Bag. Returns true if any elements were added, and false if none were added.

public boolean addAll(int index, Bag other)
  Adds all the other elements to the Bag, inserting them at the given index. Returns true if any elements were added, and false if none were added.

public boolean addAll(Collection other)
  Adds all the other elements to the Bag. Returns true if any elements were added, and false if none were added.

public boolean addAll(int index, Collection other)
: Adds all the other elements to the Bag, inserting them at the given index. Returns true if any elements were added, and false if none were added.

public boolean addAll(Object[] other)
: Adds all the other elements to the Bag. Returns true if any elements were added, and false if none were added.

public boolean addAll(int index, Object[] other)
: Adds all the other elements to the Bag, inserting them at the given index. Returns true if any elements were added, and false if none were added.

public void clone()
: Clones the Bag.

public void resize(int toAtLeast)
: Potentially resizes the Bag to accommodate at least the given number of elements.

public void shrink(int desiredLength)
: Shrinks the bag to the larger of the desired length and the current bag size, unless both are greater than or equal to the current capacity of the Bag. This is an $O(n)$ operation, so be sparing.

public boolean contains(Object obj)
: Returns true if the object exists in the Bag.

public boolean containsAll(Collection c)
: Returns true if all the objects in the Collection exist in the Bag.

public Object remove(Object obj)
: Removes the first instance of the object from the Bag and returns it. Moves the last object in the Bag to fill the position vacated by the removed object. This is an $O(1)$ operation.

public Object removeNondestructively(Object obj)
: Removes the first instance of the object from the Bag and returns it. Slides all higher-indexed elements in the Bag down by one. This is what ArrayList's remove(...) operation does, and is $O(n)$.

public boolean removeMultiply(Object obj)
: Removes all instances of the object from the Bag, collapsing all empty slots. Returns true if the object existed in the Bag.

public boolean removeAll(Collection c)
: Removes from the Bag all elements from the given collection. If there were no such elements in the Bag, returns false, else returns true.

public boolean retainAll(Collection c)
: Removes from the Bag all elements *except* those from the given collection. If there were no such elements in the Bag, returns false, else returns true.

public void clear()
: Empties the Bag.

public Object[] toArray()
: Copies all elements in the Bag to an array and returns it.

public Object[] toArray(Object[] o)
: Copies all elements in the Bag to an array of the same type as the one provided. If the passed-in array is sufficiently large, it is used, else a new one is used. Returns the resulting array.

public Iterator iterator()
: Returns an iterator over the Bag. This iterator is NOT fail-fast.

public void sort(Comparator c)
: Sorts the bag using the given comparator.

public void sort()
: Sorts the bag under the assumption that all stored objects are java.lang.Comparable.

public void fill(Object o)
: Replaces each element in the Bag with the provided object.

public void shuffle(Random random)
: Shuffles the bag uniformly using the provided random number generator.

public void shuffle(MersenneTwisterFast random)
: Shuffles the bag uniformly using the provided random number generator.

public void reverse()
: Reverses the order of the elements in the Bag.

Indexed Classes Though Bag, DoubleBag, and IntBag are not Lists, they do adhere to a simpler interface which permits random access, called sim.util.Indexed. The primary benefit of this interface is to make them easily usable in MASON's properties facility, described next in Section 3.4. The Indexed interface is fairly self-explanatory:

sim.util.Indexed Methods

public Class componentType()
: Returns the type of objects returned by this Indexed collection. Bags return java.lang.Object, while IntBags return java.lang.Integer.TYPE and DoubleBags return java.lang.Double.TYPE.

public int size()
: Returns the number of elements in the Indexed collection.

public void setValue(int index, Object value)
: Sets a slot in the collection. For IntBags, the value must be a java.lang.Integer, while for DoubleBags, the value must be a java.lang.Double. Returns the old value.

public Object getValue(int index)
: Returns a slot in the collection. For IntBags, the value will be a java.lang.Integer, while for DoubleBags, the value must be a java.lang.Double.

Note that the setValue(...) and getValue(...) methods aren't quite the same as set(...) and get(...). This is because in IntBags (for example) the former work with java.lang.Integer and the latter work with int. In plain-old Bags, they're the same.

3.4 Properties

Many MASON inspectors and other GUI classes must be able to extract the Java Bean Properties from objects. MASON has some utility classes which make this easy. MASON's property-extraction library enables you to programmatically query or set any of the following:

- The Java Bean Properties of arbitrary objects
- Slots in arrays (each slot is considered a property)
- Slots in Lists, Maps, Collections, or sim.util.Indexed (Section 3.3) classes
- Objects which provide their properties dynamically via a special method call

- Objects which provide Java Bean Properties on behalf of other objects

There are several classes and interfaces in the Properties system:

- sim.util.Properties is the top-level abstract class, and also the factory class for property objects.
- sim.util.SimpleProperties handles properties of Objects.
- sim.util.CollectionProperties handles properties of arrays, lists, maps, collections, etc.
- sim.util.Propertied defines Objects which provide their own dynamic Properties class.
- sim.util.Proxiable defines Objects which provide *other* Objects that stand in for them in providing properties.

3.4.1 Java Bean Property Methods and Extensions

The Standard Java Definition Java has a convention called **Java Bean Properties** where Objects define features by which they may be manipulated, typically by a graphical user interface widget. MASON uses Java Bean Properties, and certain MASON-only extensions, to allow the user to inspect and manipulate objects.

Every public non-static non-void method of the form getFoo() defines a *readable Java Bean Property* called Foo. If the return type of the property is boolean, the method can be either called getFoo() or isFoo(). You can access readable Java Bean Properties to get their current values. For example, here are some readable Java Bean Properties:

```
public String getName();
public int getAge();
public RockBand getCurrentRockBand();
public boolean isMale();
public boolean getQualified();
```

If there also exists a public non-static void method of the form setFoo(val), where *val* is the same type as the return type to getFoo(), then we have a *read-write Java Bean Property*. Such Java Bean Properties can be both queried and set. For example, here is a read-write Java Bean Property:

```
public String[] getNames();
public void setNames(String[] names);
```

Here's another:

```
public boolean isDominated();
public void setDominated(boolean val);
```

Note that the *set* method doesn't have to set the value — if it's a bad value, it can simply refuse to do so.

The rules for properties are rigid. The methods must be named properly, must not have additional arguments, and read-write properties must have properly matching types. Here are some invalid properties:

```
public int GetAge();
public Object getObjectAtIndex(int index);
protected String getAppropriateness();
public static String getLatestJoke();
public void getItAll();

public double getTemperature();
public void setTemperature(float val);
```

If you name methods with the proper rules, MASON will automatically recognize them as Java Bean Properties: you need to nothing more.[3]

MASON Extensions to Java Bean Properties Certain classes have been given special dispensation to permit other methods to act as read-only properties because they lack useful get...() methods. Namely: String and StringBuffer have the toString() as a property, integer Numbers have longValue(), non-integer Numbers have doubleValue(), and Booleans have booleanValue(). In all cases the property is named "Value".

In some cases you may wish to hide a property Foo so MASON GUI widgets don't display it to the user. This is easy to do with MASON's special hideFoo() extension. If this method returns true, then MASON will ignore that property. For example:

```
public int getSecretValue() { return secretValue; }
public boolean hideSecretValue() { return true; }
```

Many Java widgets which interpret Java Bean Properties take the form of text fields (for read/write properties), labels (for read-only properties), or perhaps checkboxes (for read/write boolean properties). But what about numerical properties? Couldn't they be sliders or pop-up menus? Unfortunately no: these usually just show up as text fields because Java doesn't know what the **domain** of the numerical property is.

MASON has an extension to Java Bean Properties which allows you to set the domain of a numerical property. If your property type is a *double*, you can specify minimum and maximum values of the property. For example, if you had the following *Angle* property which could only be between 0 and 2π, as below:

```
double angle = 0;
public double getAngle() { return angle; }
public void setAngle(double val) // convert to between 0 ...  2 Pi
        { val = ((val % (2 * Math.PI) ) + (2 * Math.PI) ) % (2 * Math.PI); }
```

... you can let MASON know that 0...2π is the range of the property like this:

```
public Object domAngle()  return new sim.util.Interval(0.0 , 2 * Math.PI);
```

MASON GUI widgets will respond by displaying your property not as a text field but as a **slider** from the minimum to maximum values.

The domFoo() mechanism is special to MASON, and it must always be typed to return an Object. In this example, we have used sim.util.Interval to define a fully closed numerical interval for the domain. Interval is a simple class:

sim.util.Interval Constructor Methods

public Interval(long min, long max)
    Defines a closed interval from *min* to *max*. Be sure to cast the elements as longs; else they may be interpreted as doubles.

public Interval(double min, double max)
    Defines a closed interval from *min* to *max*. Be sure to cast the elements as doubles; else they may be interpreted as longs.

sim.util.Interval Methods

public Number getMin()
    Returns the minimum interval value.

[3] Java by convention has since used Java Bean Properties as the "standard" pattern for getters and setters. You will notice that a number of MASON objects violate this convention in one fashion or another. This is by design: usually because MASON explicitly doesn't those methods to show up as properties in user interfaces by default.

public Number getMax()
Returns the maximum interval value.

public boolean isDouble()
Returns whether the interval is defined in terms of doubles (as opposed to longs).

But that's not all! *Integer* properties (those with a type of int only), you can also get MASON GUI widgets to display your property as a **pop-up menu**. It works like this. The valid domain of your properties is assumed to be from 0...i inclusive, for example:

```
public static int DEMOCRAT = 0;
public static int REPUBLICAN = 1;
public static int INDEPENDENT = 2;
public static int OTHER = 3;
public static int NONE = 4;
public int party = OTHER;
public int getParty() { return party; }
public void setParty(int val) { if (val <= NONE && val > DEMOCRAT) party = val; }
```

Create an array of $i + 1$ Strings and return them in your domain declaration method, like this:

```
public Object domParty()
   { return new String[] { "Democrat", "Republican", "Independent", "Other", "None" }; }
```

What MASON GUI widgets will do is present the user with a pop-up menu from which he may choose a party. Each String value in the domain will be shown as a separate menu item. When the user chooses an item, the Property will be set to the index value in the array. For example, if the user chooses "Independent", the party property value will be set to 2.

3.4.2 Object Properties

To extract the Java Bean Properties from an object, you merely need to pass it into one of the following methods:

sim.util.Properties Factory Methods

public static Properties getProperties(Object object)
Creates a Properties object for the given object with default values. The same as getProperties(object, true, true, false, true);

public static Properties getProperties(Object object, boolean expandCollections, boolean includeSuperclasses, boolean includeGetClass, boolean includeExtensions)
Creates a Properties object for the given object. If the object is an array, or if *expandCollections* is true and the object is a List, Indexed, Collection, or Map, then a Properties object will be constructed treating each of the slots in the object as properties. If *includeSuperclasses* is true, then properties from superclasses will be included. If *includeGetClass* is true as well as *includeSuperclasses*, then the Class property (derived from java.lang.Object.getClass()) will be included. If *includeExtensions* is true, then if the Object has MASON-style property extension methods (like domFoo() or hideFoo() they will be respected, else they will be ingored.

If you pass in an ordinary Object (not an array, List, Indexed, Collection, or Map), then you will receive back a sim.util.SimpleProperties. This subclass of Properties implements all of the following basic Properties methods which enable you to query and set property values on the object programmatically:

sim.util.Properties Methods

public Object getObject()
: Returns the object on which the Properties was constructed.

public boolean isVolatile()
: Returns true if the number or order of properties could change at any time. For ordinary objects and arrays, the answer is FALSE, but for Lists, Indexed, Collections, or Maps, the answer could be TRUE if the user modifies (for example) the List.

public int numProperties()
: Returns the current number of properties in the object.

public Object getValue(int index)
: Returns the value of property number *index*.

public Object setValue(int index, Object value)
: Sets the value of the property to *value*, returning the old value.

public Object getDomain(int index)
: Returns the domain of property number *index*, that is, the value returned by the domFoo() method.

public boolean isReadWrite(int index)
: Returns whether or not the property is a read-write property versus a read-only property.

public boolean isComposite(int index)
: Returns true if the value returned by the property is *not* a primitive type (double, int, etc.) nor a String.

public boolean isHidden(int index)
: Returns true if the object requested that this property be hidden via the hideFoo() method.

public String getName(int index)
: Returns the name of the property.

public Class getType(int index)
: Returns the data type of the property. Primitive types are described not by classes but by their type signifiers: for example, double is signified by Double.TYPE.

public String betterToString(Object obj)
: Returns a prettier toString() value for a given object than is provided by default.

When you point a Properties factory method at an ordinary Object which happens to be sim.util.Proxiable, the Properties factory method will not build a Properties object based on the Object but rather query the Object for *another* Object , the **properties proxy**, from which to extract Properties on its behalf. This querying is not transitive, even if the proxy object is itself Proxiable.

This procedure, like MASON's hideFoo() Java Beans Property extension, is meant to allow Objects to control which Properties are actually shown or to create different ones. To make your Object Proxiable, you need to implement the following method:

sim.util.Proxiable Methods

public Object propertiesProxy()
: Returns the proxy for the Object.

3.4.3 Collection Properties

Java Bean Properties, or things that look rather like them, can also be extracted from Lists, Maps, Collections, Indexed, and arrays, using the same factory methods above (with default settings). If you call the factory method, you will receive a sim.util.CollectionProperties subclass of Properties which implements all of the methods above.

Arrays An array of length *n* has exactly *n* properties, and they are all read-write. The name for property indexed *i* is simply "*i*". Arrays cannot change in length, so their properties are NON-volatile.

Lists, Collections, and Indexed These kinds of objects have *n* properties if they are holding *n* elements. Because the number of elements can change, the number of properties can change; further in Collections the order of the properties themselves can change. Thus these kinds of objects are volatile. In a List, Collection, or Indexed object, a property indexed *i* is, once again, simply "*i*". Properties objects do not at present allow you to add or delete elements, only to view and change existing ones.

Maps A Map that holds *n* elements has *n* properties. Like Collections, Maps also are volatile because the number and order of the properties can change at any time. Maps hold elements in *key*⟶*value* pairs. The name of a property corresponding to a Map element is "*key*", and the value of the property is *value*. Properties objects do not at present allow you to add or delete elements, only to view and change existing ones.

3.4.4 Dynamic Properties

In very rare cases an Object may wish to exact complete control over the properties object used for it. In this case the object itself can provide a dynamic Properties object of its own construction. For example, if you are implementing a MASON library for some other programming language, you may wish to enable MASON's GUI Inspectors to access certain language features: the easiest way to do this is to create an object which provides dynamic Properties object: when it is queried for properties, it turns around and simply asks the language what it should provide the queries.

The way this is done is to create an Object which is sim.util.Propertied:

sim.util.Propertied Methods

public Properties properties()
: Returns the proxy for the Object.

You will be required to implement *at least* the following methods:

sim.util.Properties Abstract Methods

public boolean isVolatile()
: Returns true if the number or order of properties could change at any time. For ordinary objects and arrays, the answer is FALSE, but for Lists, Indexed, Collections, or Maps, the answer could be TRUE if the user modifies (for example) the List.

public int numProperties()
: Returns the current number of properties in the object.

public Object getValue(int index)
: Returns the value of property number *index*.

public boolean isReadWrite(int index)
: Returns whether or not the property is a read-write property versus a read-only property.

public String getName(int index)
: Returns the name of the property.

public Class getType(int index)
: Returns the data type of the property. Primitive types are described not by classes but by their type signifiers: for example, double is signified by Double.TYPE.

You'll probably want to also implement the setValue(...) method for read-write properties.

3.5 Other Classes

MASON's basic utility classes also contain a few random classes which fit in no other place. They're defined here:

Valuable Objects Some MASON GUI and visualization facilities require Objects to provide numeric values. For classes like java.lang.Double this is easy. Other objects in this situation will be required to implement the sim.util.Valuable interface:

sim.util.Valuable Methods

public double doubleValue()
  Returns the current value of the object.

Notice that this is *not* getDoubleValue() — it's expressly not a Java Bean Property in case you wanted to hide this information from GUI widgets. Of course, you could *make* a getDoubleValue() method which returned the same value...

Mutable Doubles Java has no mutable wrapper for doubles, only the immutable class java.lang.Double. In some cases MASON needs to store a double as an object in a mutable fashion, and it does so using sim.util.MutableDouble. This class extends java.lang.Number, and is sim.util.Valuable, java.lang.Cloneable, and java.io.Serializable.

sim.util.MutableDouble Constructor Methods

public MutableDouble()
  Creates a MutableDouble holding 0.

public MutableDouble(double val)
  Creates a MutableDouble holding the given value.

public MutableDouble(MutableDouble other)
  Creates a MutableDouble holding the other MutableDouble's value.

---

sim.util.MutableDouble Methods

public Double toDouble()
  Returns a Double with the current value.

public double doubleValue()
  Returns the current value.

public float floatValue()
  Returns the current value cast into a float.

public int intValue()
  Returns the current value cast into an int.

public long longValue()
  Returns the current value cast into a long.

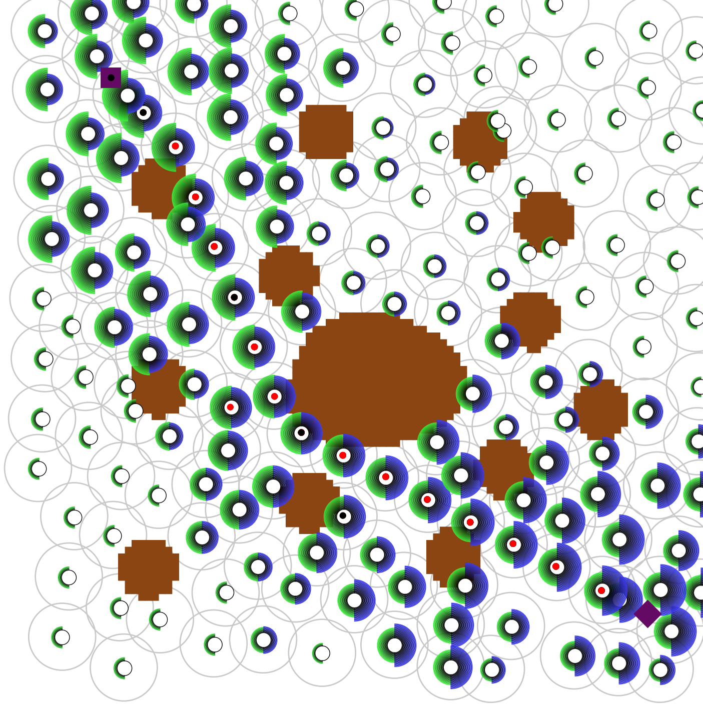
public Object clone()
  Clones the MutableDouble.

public String toString()
  Returns the MutableDouble as a String, in the same format as Double.toString().

public boolean isNaN()
    Returns true if the MutableDouble is holding NaN.

public boolean isInfinite()
    Returns true if the MutableDouble is holding a positive or negative infinite value.

The Heap MASON has a basic binary heap, found in sim.util.Heap. Since it is primarily used by the Schedule, description of the Heap may be found in Section 4.3.2.1, coming up.



Chapter 4

The Simulation Core

MASON's core simulation code is found in the package sim.engine. This package mostly contains the following elements:

- sim.engine.SimState: the global object responsible for holding your simulation **model**.
- sim.engine.Schedule: a discrete-event schedule.
- Various utility classes for making the schedule more useful.

The SimState is responsible for your simulation as a whole, and the Schedule is responsible for your simulation's notion of time. Who's responsible for your simulation's notion of space (or agent relationships)? These are called **fields**, and MASON's default fields are not found in sim.engine proper, but rather in other packages:

- sim.field.grid holds 2-dimensional and 3-dimensional grid data structures. These are discussed in Section 5.
- sim.field.continuous holds 2-dimensional and 3-dimensional continuous data structures. These are discussed in Section 6.
- sim.field.network holds graphs and networks. These are discussed in Section 7.

You'll make a subclass of SimState and create an instance of it. This instance will hold your Schedule (the representation of time) plus whatever fields (the representation of space), and some other stuff which collectively make up the model proper.

Let's start with SimState. Then we'll move to the top-level simulation loop and the Schedule. In later Sections we'll talk about various fields.

4.1 The Model

Your simulation model will be entirely encapsulated within a single class: a subclass of sim.engine.SimState which you design. MASON will create a single instance of your subclass to hold your model. The purpose of this class is to give you a place to store any and all elements you deem necessary for your simulation. For example, you will probably have some number of **fields** (representations of space, discussed later in Sections 5, 6, and 7) either of your design or provided by MASON. These will likely appear as instance variables in your class. You might have certain objects—collections of agents or whatnot—stored away for some

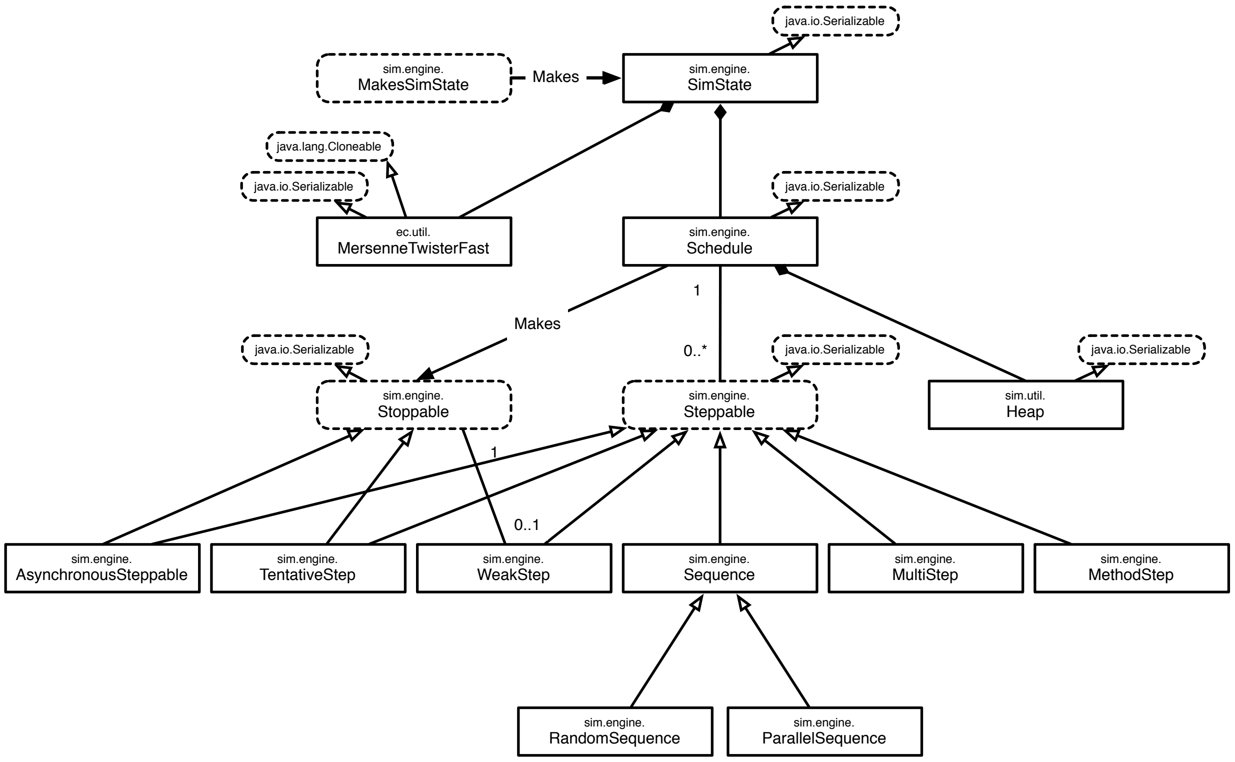


Figure 4.1 UML diagram of MASON's core simulation and discrete-event schedule facility.

purpose. And almost certainly you'll have a bunch of **model parameters** stored as instance variables as well. SimState is your catch-all global object friend.

Because everything in the model ultimately hangs off of a single instance of a subclass of SimState, MASON can **serialize** the entire model to a **checkpoint file**, meaning it can freeze the simulation in time and save it in its entirety to the file. You can restart the model from this file, even on a different computer or operating system, or under a visualization facility, and the simulation will continue right where it left off without thinking twice about it. In order to make this possible, all MASON model objects are java.io.Serializable, meaning they can be (largely automatically) written to or read from a stream.

SimState already contains two instances ready for you to use:

```
public MersenneTwisterFast random;
public Schedule schedule;
```

We'll get to the schedule in Section 4.3. random is an ec.util.MersenneTwisterFast, a random number generator which (true to its name) is a fast-operating implementation of the Mersenne Twister algorithm. Unlike java.util.Random, this is a high-quality random number generator, and as discussed in Section 3.1, its methods are essentially identical to java.util.Random. **Do not use java.util.Random or java.lang.Math.random() in your simulation. You should always use random.**

ec.util.MersenneTwisterFast is not thread synchronized Ordinarily your MASON simulation will involve a single thread, so there's no issue. But if you break your simulation into multiple threads, you'll need to make sure that when you access the random

Why do you lock on the schedule instead of the random number generator?

Because MASON needs to guarantee thread-safeness for other tasks than just random number generation. For example, access to fields or global model parameters. We use the schedule because it's synchronized already, and so makes a good central lock. Always use the schedule as your lock point for threadsafe access in MASON simulations.

number generator you do so in a threadsafe manner. This is done by locking on schedule. For example, imagine if you need a random double. You should do something like this:

```
SimState mySimState = ...
double val = 0;
synchronized(state.schedule) { val = random.nextDouble(); }
```

You can add anything you need to your custom subclass of SimState: fields, global parameters, etc.

Imagine that each timestep your simulation model is, for some reason, updating an array of Strings, and also a 2D grid of integers using MASON's sim.field.grid.IntGrid2D class (we discuss this class later in Section 5. You might create a subclass like this which stores them for the duration of the simulation:

```
import sim.engine.*;
import sim.field.grid.*;

public class MySimulation extends SimState
    {
    public String[] strings = new String[] { "Initial String 1", "Initial String 2", "Initial String 3" };
    public IntGrid2D grid = new IntGrid2D(100, 100);
    }
```

MASON will make one instance of your subclass. Everything scheduled on the Schedule will be given access to the instance of your SimState subclass when its their time to do their thing, so you can consider this instance as essentially a global repository for your simulation.

4.2 The Big Loop

MASON does simulations by setting up a SimState, allowing it to load the Schedule with things to step, then repeatedly stepping the Schedule until it is exhausted (there's nothing left to step), then cleaning up. SimState plays a critical role in a loop of this type. Often MASON's simulation loop looks something like this:

```
1. Create an instance of a SimState subclass called state.
2. Call state.nameThread(...); to label the thread with a name you'll recognize in debuggers.
3. Loop some jobs times:
4.       Call state.setJob(...);
5.       Call state.start();
6.       Loop:
7.             Call boolean result = state.schedule.step(state);
8.             If result == false or if too much time has passed, break from Loop
9.       Call state.finish();
10. Call System.exit(0); for good measure (to kill any wayward threads you may have accidentally created).
```

You can easily write this loop yourself:

```
public static void main(String[] args)
    {
    int jobs = 100;  // let's do 100 runs
    SimState state = new MyModel(System.currentTimeMillis()); // MyModel is our SimState subclass
    state.nameThread();
    for(int job = 0; job < jobs; job++)
        {
        state.setJob(job);
        state.start();
        do
            if (!state.schedule.step(state)) break;
        while(state.schedule.getSteps() < 5000);
        state.finish();
        }
    System.exit(0);
    }
```

This is a slight extension of the version shown in Section 2.1 of the Schoolyard Cliques tutorial, which was missing lines 2, 3, and 4 (it had only one job iteration, which was automatically job number 0, and the thread label is unnecessary).

This example relies on several vital methods, including start(), finish(), and schedule.step(...). We'll get to those presently. Let's cover the other elements first. To begin with, your SimState has a single constructor:

sim.engine.SimState Constructor Methods

public SimState(long seed)
: Produces a new SimState with an empty Schedule and a random number generator seeded with the given seed. If you override this, be sure to call super(seed);

Next various utility methods:

sim.engine.SimState Methods

public void setSeed(long seed)
: Sets the job's current random number seed, replaces the random number generator with a new one using the seed. This is typically only called from the GUI.

public long seed()
: Returns the job's current random number seed. This is typically only called from the GUI.

public void setJob(long job)
: Sets the job number.

public long job()
: Returns the job number.

public static double version()
: Returns MASON's version number.

public void nameThread()
: Stamps the current thread with a name easily recognized by debuggers, such as "`MASON Model:  Students`"

Notice that a number of these methods aren't in standard Java Beans format (that is, they're called foo() rather than getFoo(). This is on purpose: the SimState is often inspected in the GUI, and we don't want these methods being displayed as properties. If you'd prefer otherwise, simply make a cover method (like getSeed()

4.2.1 Checkpointing

MASON's models are, crucially, **serializable** to checkpoints: you can in essence freeze-dry and save them to a file to be thawed out and started up later. The models never know and just pick up where they had left off.

Most of this is Java serialization magic and you need know nothing about it. But there are some situations where things *won't* be exactly the same when your model is unfrozen. Most commonly, you'll need to open files or sockets up again which had long since been closed since your model went into its coma. For example, your simulation may need to re-open some statistics logs it produces. Thus MASON provides three SimState hooks for you to be informed when your simulation is checkpointed or awoken from a checkpoint.

Additionally, MASON has four methods for actually performing checkpointing. These methods are almost universally called from your top-level loop: we recommend against calling them from *within* your simulation. Here we go:

sim.engine.SimState Methods

public void preCheckpoint()
A hook called immediately before checkpointing occurs. If you override this, be sure to call super.preCheckpoint();

public void postCheckpoint()
A hook called immediately after checkpointing has occurred. If you override this, be sure to call super.postCheckpoint();

public void awakeFromCheckpoint()
A hook called immediately after waking up from a checkpoint. If you override this, be sure to call super.awakeFromCheckpoint();

public void writeToCheckpoint(Output stream) throws IOException
Checkpoints the SimState out to the given stream.

public SimState writeToCheckpoint(File file)
Checkpoints the SimState out to the given file and returns it. If an error occurred, returns null.

public static SimState readFromCheckpoint(InputStream stream) throws IOException, ClassNotFoundException, OptionalDataException, ClassCastException
Reads a SimState from the given stream, constructs it, and returns it.

public static SimState readFromCheckpoint(File file)
Reads a SimState from the given file, constructs it, and returns it. If an error occurred, returns null.

Adding checkpointing complicates the top-level loop, particularly if you mix it with a job facility. Here's one possibility, which does 100 jobs, each of 5000 steps, checkpointing out at 2500 each time.

```
public static void main(String[] args)
    {
    int jobs = 100;  // let's do 100 runs
    int job = 0;

    SimState state = null;
    if (args.length > 0)
        {
        state = SimState.readFromCheckpoint(new File(args[0]));
        if (state == null) return;  // uh oh
        job = state.job();
        }
    else
        {
        state = new MyModel(System.currentTimeMillis());
        state.start();
        }

    for( ; job < jobs; job++)
```

```
            {
            state.setJob(job);
            do
                {
                if (!state.schedule.step(state)) break;
                if (state.schedule.getSteps() == 2500)
                    state.writeToCheckpoint(new File("out." + job + ".checkpoint"));
                }
            while(state.schedule.getSteps() < 5000);
            state.finish();
            if (job < jobs - 1)  // we're not done yet
                state.start();  // notice we put it here so as not to start when reading from a checkpoint
            }
        System.exit(0);
        }
```

4.2.2 The doLoop() Method

Ugh, that's getting ugly. Though you can always fire up a MASON model as described above, if you're running the model from the command line (without visualization), there's a far easier approach: use SimState's doLoop method:

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

The advantage of doLoop is that it has provides lots of command-line gizmos for free:

- Job handling. You can run the simulation some *R* times.
- Checkpointing. You can easily start up the simulation from a checkpoint file and/or save checkpoints every *D* simulation steps.
- Specifying the random number seed.
- Running the simulation for some number of simulation steps, or until a given simulation timestep has passed.
- Automatically print out timestamps as you go so you know how far the model has gotten.

If you run the MyModel program from the command line with the given parameter:

```
java MyModel -help
```

... the doLoop will give you back a full description of its capabilities:

```
Format:          java MyModel \
                      [-help] [-repeat R] [-seed S] [-until U] \
                      [-for F] [-time T] [-docheckpoint D] [-checkpoint C]

-help            Shows this message and exits.

-repeat R        Long value > 0: Runs the job R times.  Unless overridden by a
                  checkpoint recovery (see -checkpoint), the random seed for
                  each job is the provided -seed plus the job# (starting at 0).
                  Default: runs once only: job number is 0.

-seed S          Long value not 0: the random number generator seed, unless
                  overridden by a checkpoint recovery (see -checkpoint).
                  Default: the system time in milliseconds.
```

```
-until U          Double value >= 0: the simulation must stop when the
                   simulation time U has been reached or exceeded.
                   Default: don't stop.

-for N            Long value >= 0: the simulation must stop when N
                   simulation steps have transpired.
                   Default: don't stop.

-time T           Long value >= 0: print a timestamp every T simulation steps.
                   If 0, nothing is printed.
                   Default: auto-chooses number of steps based on how many
                   appear to fit in one second of wall clock time.  Rounds to
                   one of 1, 2, 5, 10, 25, 50, 100, 250, 500, 1000, 2500, etc.

-docheckpoint D   Long value > 0: checkpoint every D simulation steps.
                   Default: never.
                   Checkpoint files named
                   <steps>.<job#>.MyModel.checkpoint

-checkpoint C     String: loads the simulation from file C, recovering the job
                   number and the seed.  If the checkpointed simulation was begun
                   on the command line but was passed through the GUI for a while
                   (even multiply restarted in the GUI) and then recheckpointed,
                   then the seed and job numbers will be the same as when they
                   were last on the command line.  If the checkpointed simulation
                   was begun on the GUI, then the seed will not be recovered and
                   job will be set to 0. Further jobs and seeds are incremented
                   from the recovered job and seed.
                   Default: starts a new simulation rather than loading one, at
                   job 0 and with the seed given in -seed.
```

The doLoop method actually comes in two forms:

sim.engine.SimState Methods

public static void doLoop(MakesSimState generator, String[] args)
args are the command-line arguments. The *generator* is called to produce an instance of the desired SimState subclass. The method start() is called on this instance. Then the top-level simulation loop is entered, calling step() on the schedule each time. Finally, stop() is called on the instance.

public static void doLoop(Class c, String[] args)
args are the command-line arguments. An instance of the given class is produced: this class must subclass from SimState. The method start() is called on this instance. Then the top-level simulation loop is entered, calling step() on the schedule each time. Finally, stop() is called on the instance.

In fact, the second of these methods simply calls the first one with a little wrapper.

So what's a sim.engine.MakesSimState? This is a simple interface for objects which produce instances of SimState subclasses, and that's all. It's a mechanism which allows you some flexibility in how your simulation is created (though it's rarely used). The MakesSimState class has a single method:

sim.engine.MakesSimState Methods

public SimState newInstance(long seed, String[] args)
args are the command-line arguments, and *seed* is the seed for the random number generator. This method produces an instance of a subclass of sim.engine.SimState, populating it with a Schedule and a seeded random number generator, then returns it.

It'll be pretty rare that you'd need this.

4.2.3 Starting and Finishing

When a simulation is begun from the top-level, the first method called on it (other than the constructor) is start(). The final method called at the end of the simulation run is finish(). They're defined as follows:

sim.engine.SimState Methods

public void start()
: Called immediately before the schedule is iterated. You will probably want to override this method to set up the simulation (or clean it up and get it ready again). Be sure to call super.start() first.

public void finish()
: Called immediately after the schedule is exhausted. You may wish to override this method to clean up afterwards: for example, closing various streams. Be sure to call super.finish() first.

The most common method to override, by far, is start() first. Almost certainly, you'll override this do these tasks:

- Reset the global variables (parameters, fields, etc.) of your simulation to be used again
- Reload the schedule it with initial agents (see Section 4.3 below). The schedule will have already been cleared for you in super.start()

4.3 Agents and the Schedule

sim.engine.Schedule is a discrete-event schedule: MASON's representation of time. It is the most central element of MASON, and so it's important to understand how it works. A schedule is a data structure on which you can (of course) *schedule* objects to be *stepped*[1] at some point in the future. Here's how the general loop works:

1. During start() you put initial items on the schedule, associating each with a time.
2. When the top-level loop calls schedule.step(...):
   (a) The schedule advances its internal time stamp to that of the minimally-scheduled item.
   (b) The schedule extracts all the items scheduled for that time, then sorts them and steps them in that order.
   (c) When items are stepped, they may in turn put themselves back on the schedule, or add new items to the schedule.
3. When there are no items left on the schedule, the top-level loop calls finish(), then ends the simulation.

Everything that can be posted on the Schedule must adhere to the interface sim.engine.Steppable (which is in turn automatically java.io.Serializable). The Steppable interface defines a single, and fairly obvious, method:

sim.engine.Steppable Methods

public void step(SimState state)
: Called when the Steppable is being pulsed. The Steppable should respond by performing some action.

[1]Or fired, or pulsed, or called, or whatever.

Most Steppable objects are more or less MASON's concept of **agents**: computational entities which manipulate their environment in response to feedback they have gathered about it. An agents' computational process could be implemented in all sorts of ways: as a separate thread for example. But MASON's basic approach is to assume that agents do their jobs in small increments which can be triggered by **events** posted on the Schedule. Thus MASON is, fundamentally, a simple **discrete event simulation**.

For example, suppose you have a robot as an agent, and it has been posted on the schedule. Each time the schedule steps it, the robot advances by some epsilon. Or perhaps you have a thousand fish scheduled on the schedule. Each time a fish is stepped, it makes a decision to swim a little bit in some new direction. Or maybe you have a social network. Each time a person on the network is stepped, he changes his internal state a bit based on the current opinions of his neighbors on the network.

4.3.1 Scheduling

MASON's Schedule is little more than a binary queue with a current real-valued **simulation time** and a current number of **steps** (the count of how often schedule.step(...) has been called by the top-level loop). The Schedule defines certain important constants:

```
public static final double EPOCH = 0.0;
public static final double BEFORE_SIMULATION;
public static final double AFTER_SIMULATION;
public static final double EPOCH_PLUS_EPSILON;
public static final double MAXIMUM_INTEGER;
```

When the schedule is first made, its initial time is set to the the value BEFORE_SIMULATION. You can't schedule agents for this time: the earliest you can possibly schedule an agent is at the EPOCH (the "dawn of time" so to speak). The earliest possible timestep that's still after the epoch is EPOCH_PLUS_EPSILON (you won't use this much). When the simulation has concluded, the schedule is set to AFTER_SIMULATION. Last but very important is MAXIMUM_INTEGER. Many simulations schedule agents with integer timestamps. There are not a finite number of these: at some (very high) value, the integer timestamp becomes so large that if you added 1 to it the resulting double value would actually jump to much larger than the timestamp + 1. You should be aware of this point.

Here are some methods for accessing the current time and steps:

sim.engine.Schedule Methods

public long getSteps()
  Returns how many times the Schedule has had its step(...) method called.

public double getTime()
  Returns the Schedule's current simulation time.

public String getTimestamp(String beforeSimulationString, String afterSimulationString)
  Returns the Schedule's current simulation time as a String value. If the time is BEFORE_SIMULATION, then beforeSimulationString is returned. If the time is AFTER_SIMULATION, then afterSimulationString is returned.

public String getTimestamp(double time, String beforeSimulationString, String afterSimulationString)
  Returns the provided time as a String value. If the provided time is BEFORE_SIMULATION, then beforeSimulation-String is returned. If the provided time is AFTER_SIMULATION, then afterSimulationString is returned.

Agents are scheduled by associating them with a **time** (a double value) an an **ordering** (an integer). The time indicates exactly when (in simulation time, not wall-clock time) the agent's step(...) method is to be called. You must schedule an agent for a value higher than the current simulation time: if you schedule an agent for exactly the current simulation time, an epsilon (a small positive value) will be added.

Agents scheduled for exactly the same time will have their step(...) methods called in the order specified by their relative ordering values (lower orderings first). Agents with exactly the same time and exactly the same ordering will have their step(...) methods called in random order with respect to one another.

You can schedule agents on the Schedule in two basic ways: in **one-shot** or **repeating**. An agent scheduled one-shot will have its step(...) method called once at some agreed on point in time in the future, and then the agent will be dropped from the Schedule. An agent scheduled repeating will stay on the schedule indefinitely, repeatedly having its step(...) method called every so often.

There are lots of methods available for scheduling agents as one-shot or as repeating. Don't let this daunting list get to you. There are really just three basic methods: scheduleOnce(...), which schedules an agent one time only at an absolute time, scheduleOnceIn(...), which schedules an agent once at a time relative to the current time, and scheduleRepeating(...), which schedules an agent repeating. All the other methods are just simplifications of those methods with common default values.

sim.engine.Schedule Methods

public boolean scheduleOnce(Steppable agent)
: Schedules the given agent at the current time + 1.0, with an ordering of 0, and returns true. If the agent cannot be scheduled at that time (or the simulation is over), or the agent is null, returns false.

public boolean scheduleOnce(Steppable agent, int ordering)
: Schedules the given agent at the current time + 1.0, with the given ordering, and returns true. If the agent cannot be scheduled at that time (or the simulation is over), or the agent is null, returns false.

public boolean scheduleOnce(double time, Steppable agent)
: Schedules the given agent for the given time, with an ordering of 0, and returns true. If the given time is exactly equal to the current time, an epsilon (a minimally small positive value) is added to the given time. If the agent cannot be scheduled at that time (or the simulation is over), or the agent is null, returns false.

public boolean scheduleOnce(double time, Steppable agent, int ordering)
: Schedules the given agent for the given time and ordering, and returns true. If the given time is exactly equal to the current time, an epsilon (a minimally small positive value) is added to the given time. If the agent cannot be scheduled at that time (or the simulation is over), or the agent is null, returns false.

public boolean scheduleOnceIn(double delta, Steppable agent)
: Schedules the given agent at the current time + delta, with an ordering of 0, and returns true. If the delta is zero, it is replaced by an epsilon (a minimally small positive value). If the agent cannot be scheduled at that time (or the simulation is over), or the agent is null, returns false.

public boolean scheduleOnceIn(double delta, Steppable agent, int ordering)
: Schedules the given agent at the current time + delta, and at the given ordering, and returns true. If the delta is zero, it is replaced by an epsilon (a minimally small positive value). If the agent cannot be scheduled at that time (or the simulation is over), or the agent is null, returns false.

public Stoppable scheduleRepeating(Steppable agent)
: Schedules the given agent at the current time + 1.0, with an ordering of 0. Once the agent has been stepped, it will be automatically re-scheduled for the current time + 1.0 more, again with an ordering of 0. This will continue indefinitely. Returns a Stoppable which can be used to stop further re-scheduling. If the agent cannot be scheduled at the initial time (or the simulation is over), or the agent is null, returns null.

public Stoppable scheduleRepeating(Steppable agent, double interval)
: Schedules the given agent at the current time + *interval*, with an ordering of 0. Once the agent has been stepped, it will be automatically re-scheduled for the current time + *interval* more, again with an ordering of 0. This will continue indefinitely. *interval* must be greater than 0. Returns a Stoppable which can be used to stop further re-scheduling. If the agent cannot be scheduled at the initial time (or the simulation is over), or the agent is null, returns null.

public Stoppable scheduleRepeating(Steppable agent, int ordering, double interval)
: Schedules the given agent at the current time + *interval*, with the given ordering. Once the agent has been stepped, it will be automatically re-scheduled for the current time + *interval* more, again with the given ordering. This

will continue indefinitely. *interval* must be greater than 0. Returns a Stoppable which can be used to stop further re-scheduling. If the agent cannot be scheduled at the initial time (or the simulation is over), or the agent is null, returns null.

public Stoppable scheduleRepeating(double time, Steppable agent)
Schedules the given agent at provided time and with an ordering of 0. If the given time is exactly equal to the current time, an epsilon (a minimally small positive value) is added to the given time. Once the agent has been stepped, it will be automatically re-scheduled for the current time + 1.0 more, again with an ordering of 0. This will continue indefinitely. *interval* must be greater than 0. Returns a Stoppable which can be used to stop further re-scheduling. If the agent cannot be scheduled at the initial time (or the simulation is over), or the agent is null, returns null.

public Stoppable scheduleRepeating(double time, Steppable agent, double interval)
Schedules the given agent at provided time and with an ordering of 0. If the given time is exactly equal to the current time, an epsilon (a minimally small positive value) is added to the given time. Once the agent has been stepped, it will be automatically re-scheduled for the current time + *interval* more, again with an ordering of 0. This will continue indefinitely. *interval* must be greater than 0. Returns a Stoppable which can be used to stop further re-scheduling. If the agent cannot be scheduled at the initial time (or the simulation is over), or the agent is null, returns null.

public Stoppable scheduleRepeating(double time, int ordering, Steppable agent)
Schedules the given agent at provided time and ordering. If the given time is exactly equal to the current time, an epsilon (a minimally small positive value) is added to the given time. Once the agent has been stepped, it will be automatically re-scheduled for the current time + 1.0 more, again with the given ordering. This will continue indefinitely. *interval* must be greater than 0. Returns a Stoppable which can be used to stop further re-scheduling. If the agent cannot be scheduled at the initial time (or the simulation is over), or the agent is null, returns null.

public Stoppable scheduleRepeating(double time, int ordering, Steppable agent, double interval)
Schedules the given agent at provided time and ordering. If the given time is exactly equal to the current time, an epsilon (a minimally small positive value) is added to the given time. Once the agent has been stepped, it will be automatically re-scheduled for the current time + *interval* more, again with the given ordering. This will continue indefinitely. *interval* must be greater than 0. Returns a Stoppable which can be used to stop further re-scheduling. If the agent cannot be scheduled at the initial time (or the simulation is over), or the agent is null, returns null.

It's important to note that **all of these methods are threadsafe**. Any thread can schedule agents on the schedule at any time.

When an agent is scheduled repeating, you are returned a sim.engine.Stoppable. This is a simple interface which allows you to stop the re-scheduling of the agent:

sim.engine.Stoppable Methods

public void stop()
Stops whatever the Stoppable is designed to stop.

4.3.2 Iterating and Stopping the Schedule

In the model's start() method we add agents to the Schedule. The top-level loop then repeatedly pulses the Schedule until there are no more agents in it, at which time finish() is called.

Here's how the pulsing works. The top-level loop calls the following method on the Schedule:

sim.engine.Schedule Methods

public boolean step(SimState state)
Steps the schedule forward one iteration. Returns false if there were no agents left, and the Schedule has thus has advanced its time to AFTER_SIMULATION. Else returns true.

Note that this method returns a boolean: it's not the same method as step(...) in Steppable objects.[2] The Schedule's step(...) method works as follows:

1. If the time is AFTER_SIMULATION or if there are no more agents, set the time to AFTER_SIMULATION and return false immediately.
2. Identify the timestamp of the earliest-scheduled agent on the Schedule.
3. Advance the Schedule's current time to that timestamp.
4. Remove from the Schedule all agents scheduled for that time.
5. Sort the removed agents with lower-ordering agents first. Agents with identical ordering are shuffled randomly with regard to one another.[3]
6. Call the step(...) method on each of the agents in their sorted order. At this stage agents are free to add themselves or other agents back into the Schedule.
7. Each time the step(...) is called on an agent, increment the Schedules steps counter by one.
8. Return true.

Some things to note. First, though the Schedule is threadsafe, it's perfectly fine for an agent to call any of the scheduling methods from *within* its step(...) method call. In fact, that's what a great many agents will do. The Schedule's step(...) procedure isn't holding onto the synchronized lock.

Second, let's say the Schedule has removed various agents and is in the process of stepping each of them. An agent then tries to reinsert itself at the current timestamp. It cannot. Thus the Schedule does not allow agents to cause queue starvation (hogging it by constantly reinserting themselves at the current time).

If you like you can clear out the entire Schedule. There are two methods to do this (clear() and reset()), but in *both cases* any agents which have been removed to be processed will still have their step(...) methods called.[4] Methods for querying and manipulating the Schedule:

sim.engine.Schedule Methods

public void clear()
: Clears out the schedule. If this method is called while agents are being stepped, all agents meant to be stepped this timestamp will still be stepped.

public void reset()
: Clears out the schedule, as in clear(), resets the step count to 0, and sets the time to BEFORE_SIMULATION.

public boolean scheduleComplete()
: Returns true if the Schedule is empty.

If your goal is to end the simulation prematurely, don't use the clear() or reset() methods, as they don't clear out AsynchronousSteppables (see below). Instead there is a special SimState method which does the task for you:

sim.engine.SimState Methods

[2]In retrospect, it should have probably been named something else to avoid confusion.

[3]We used to have an option to turn shuffling off because users wanted to be able to execute agents in exactly the order in which they were added to the Schedule. But that's mistaken: the Schedule is backed by a binary heap, and the dynamics of a heap are such that agents aren't removed in the order they're inserted even if they have the exact same key value. So we removed that ability to avoid confusion.

[4]We used to have a feature in the Schedule which would prevent those agents from being stepped as well, but the complexity was just too great to be worthwhile. So it's gone.

public void kill()
  Clears the schedule, pushes the Schedule's time forward to AFTER_SIMULATION, and clears out all AsynchronousSteppables.

Note that this method should NOT be called from within the step() methods of AsynchronousSteppables, ParallelSequences, or other agents with non-primary threads, as it will cause a deadlock. Instead, these kinds of agents can kill the simulation by scheduling an agent for the immediate next timestep which itself calls kill(). For example:

```
schedule.scheduleOnceIn(0,
        new Steppable() { public void step(SimState state) { state.kill(); } } );
```

4.3.2.1 Under the Hood

> *Why doesn't Schedule use java.util.PriorityQueue?*
>
> sim.util.Heap is a little bit faster, but that's not the reason. PriorityQueue didn't exist when MASON was developed.

The Schedule employs a binary heap: specifically the MASON utility class sim.util.Heap. A binary heap is a data structure in which objects can be inserted and associated with *keys* which specify their ordering with respect to one another. Inserting or removing an object is $O(\lg n)$, and finding the minimum object is $O(\lg 1)$. The class is pretty simple:

sim.util.Heap Constructor Methods

public Heap()
  Creates an empty binary heap.

public Heap(Comparable[] keys, Object[] objects)
  Creates a binary heap holding the given objects and their corresponding keys. The number of objects and keys must match.

---

sim.util.Heap Methods

public void add(Object object, Comparable key)
  Adds an object with its associated key to the heap.

public boolean isEmpty()
  Returns true if the heap is empty.

public void clear()
  Empties the heap.

public Comparable getMinKey()
  Returns the minimum key in the heap.

public Object extractMin()
  Removes the minimum object and its key from the heap, and returns the object. If the heap is empty, null is returned.

public Bag extractMin(Bag putInHere)
  Removes all objects (and their keys) whose keys are equal to that of the minimum object. Places the objects in the provided Bag and returns it (not clearing it first). If the Bag is null, a new Bag is created. If the heap is empty, the Bag will be empty.

MASON uses a binary heap because we don't know beforehand how you'll be using the Schedule, and so need something general-purpose. Binary heaps are fast enough for most purposes, but if you have a lot of agents and know something about how they'll be scheduled, you may be able to do significantly better, with a smarter **calendar queue** tuned for your application. All you have to do is write a subclass of Heap which overrides above methods. Then override Schedule's createHeap() method to return your special kind of calendar queue instead of the standard Heap. Then in SimState's constructor, set the Schedule to your special subclass to use it instead of the standard Schedule.

sim.util.Schedule Methods

protected Heap createHeap()
Creates and returns a new Heap for the Schedule to use.

4.3.3 Utility Agent Classes

MASON provides a variety of utility subclasses of Steppable which you may find useful to expand on the basic capabilities of the Schedule. Most of these subclasses are designed to hold other Steppables witin them, and when stepped, step those Steppables in various ways:

- sim.engine.Sequence holds an array of Steppables and when stepped it steps each of them in turn.
- sim.engine.RandomSequence is like Sequence, but when stepped it first shuffles the order of the Steppables in its array.
- sim.engine.ParallelSequence is like Sequence, but when stepped it steps each of its Steppables in parallel in separate threads.
- sim.engine.TentativeStep is both Steppable and Stoppable. It holds a subsidiary Steppable, and when stepped, it steps its Steppable, unless you have first called stop() on the TentativeStep.
- sim.engine.WeakStep holds a subsidiary Steppable weakly (meaning it can be garbage collected at any time), and when stepped, it steps its Steppable only if it still exists.
- sim.engine.MultiStep holds a subsidiary Steppable, and when stepped, it either steps its subsidiary N times in a row, or once every N steps.
- sim.engine.MethodStep holds a subsidiary Object (not a Steppable), and when stepped, calls a method of your choice on that Object.
- sim.engine.AsynchronousSteppable is an abstract Steppable class. When stepped, it forks off an asynchronous thread to do some task in the background. In the future you can stop the thread and rejoin it; or pause it; or resume it.

We next go through each of these in turn.

sim.engine.Sequence This class holds an array of agents, and when it is stepped, it calls step(...) on each of the agents in turn.

sim.engine.Sequence Constructor Methods

public Sequence(Steppable[] agents)
Builds a Sequence for the given array of agents (there should be no null values, and the array ought not be modified later).

sim.engine.RandomSequence This is a subclass of Sequence: the difference is that, when stepped, it first uniformly shuffles the order of the agents in the array, then calls calls step(...) on each of them in turn. In order to shuffle the agents, RandomSequence uses the simulation's random number generator. As a result, you have the option of the RandomSequence first locking on the Schedule prior to using the random number generator. If your simulation is single-threaded when the RandomSequence is fired, then this is unnecessary and (slightly) slower. Since most simulations are single-threaded, the default is to not lock. On the other hand, if you've got multiple threads going on — notably, if you're using the RandomSequence from within a ParallelSequence (see below), you should turn locking on.

sim.engine.RandomSequence Constructor Methods

public RandomSequence(Steppable[] agents)
: Builds a RandomSequence for the given array of agents (there should be no null values, and the array ought not be modified later). Prior to stepping the agents, the RandomSequence will first shuffle the order of the array using the model's random number generator. The sequence will not synchronize on the random number generator prior to calling it.

public RandomSequence(Steppable[] agents, boolean shouldSynchronize)
: Builds a RandomSequence for the given array of agents (there should be no null values, and the array ought not be modified later). Prior to stepping the agents, the RandomSequence will first shuffle the order of the array using the model's random number generator. The sequence will synchronize on the random number generator prior to calling it if *shouldSynchronize* is true.

---

sim.engine.ParallelSequence This is a subclass of Sequence which, instead of stepping each agent in turn, instead steps all of them simultaneously in separate threads. Be careful when using ParallelSequence: as you have multiple threads, it's up to you to make sure that you don't create race conditions as they access various parts of your model simultaneously. Notably if your agents use the random number generator, they should lock on the Schedule first, for example like this:

```
double val = 0;
synchronized(state.schedule) { val = state.random.nextDouble(); }
// now use val here....
```

Notably the RandomSequence described above needs to be used carefully when in combination with ParallelSequence: make certain that it is set to synchronize if used inside one of the ParallelSequence's threads.

sim.engine.ParallelSequence Constructor Methods

public ParallelSequence(Steppable[] agents)
: Builds a ParallelSequence for the given array of agents (there should be no null values, and the array ought not be modified later). When its step(...) method is called ParallelSequence will step each of these agents in parallel in separate threads, then wait for them to complete before returning.

public ParallelSequence(Steppable[] agents, int numThreads)
: Builds a ParallelSequence for the given array of agents (there should be no null values, and the array ought not be modified later). When its step(...) method is called, ParallelSequence will generate *numThreads* number of threads, then divide the agents roughly evenly among the various threads. Each thread will then step its assigned agents in some order. ParallelSequence will wait for all threads to complete this task before returning. If for *numThreads* you pass in ParallelSequence.CPUS, then ParallelSequence will query Java for the number of CPUs or cores on the system and spawn exactly that number of threads.

Allocating and starting threads is very expensive. To be efficient, ParallelSequence allocates the threads when constructed, then holds them ready in reserve in a semaphore until the step(...) method is called. It then sets them going, and when they are done, they once again are held waiting in the semaphore. The threads are set to be daemon threads so they die if MASON exits. This is far faster than creating and destroying them every time.

There's a downside however: if the Schedule gets rid of the ParallelSequence, and it is garbage collected, the threads will still be hanging around in memory, producing a serious memory leak and consuming precious resources. ParallelSequence.finalize() is set to destroy the threads, but there's no guarantee that method will be called. What to do?

You have three options:

- Set the ParallelSequence to destroy its threads each and every time its step(...) method finishes. This is expensive but requires no further thought.
- In your SimState's finish() method, manually call cleanup() on each of your ParallelSequences to destroy their threads after the simulation has concluded.
- Sometime mid-simulation, after a ParallelSequence has outlived its usefulness, schedule a special Steppable which, when its step(...) method is called, will call cleanup() on the ParallelSequence. You can your own Steppable or use a convenience method provided by ParallelSequence below.

The ParallelSequence makes any of these options easy with the following methods:

sim.engine.ParallelSequence Methods

public boolean getDestroysThreads()
    Returns true if the ParallelSequence is set to destroy its threads each and every time its step(...) method completes.

public void setDestroysThreads(boolean val)
    Sets the ParallelSequence to destroy its threads (or not) each and every time its step(...) method completes.

public void cleanup()
    Destroys all the threads presently held in reserve. Note that if the ParallelSequence again has its step(...) method called, the threads will be recreated (necessitating further calls to cleanup().

public Steppable getCleaner()
    Returns a Steppable which, when stepped, will call cleanup() on the ParallelSequence. This can be scheduled at some point in the future to clean up threads after the ParallelSequence has been terminated.

See the sim.app.heatbugs.ThreadedDiffuser class for a good example of how to use ParallelSequence to dramatically speed up a simulation.

sim.engine.TentativeStep Let's suppose you want to schedule an agent one-shot, but it's possible that in the future you'd need to prevent it from having its step(...) method called. TentativeStep to the rescue! Create a TentativeStep wrapped around your agent and schedule the TentativeStep instead. When the TentativeStep is stepped, it'll call step(...) on the underlying agent. But TentativeStep is sim.engine.Stoppable, so you can call stop() on it at any time to prevent this from happening.

sim.engine.TentativeStep Constructor Methods

public TentativeStep(Steppable agent)
    Builds a TentativeStep for a given agent.

---

sim.engine.TentativeStep Methods

public void stop()
    Prevents the TentativeStep from ever calling step(...) on its underlying agent.

TentativeStep's step(...) and stop() methods are synchronized: different methods can call them safely.

sim.engine.WeakStep In some rare cases you may need to schedule an agent *weakly*, meaning that if Java is running low on memory, it should feel free to garbage collect the agent before its step(...) method is called. WeakStep enables this: create the WeakStep, passing in the agent you wish to be held weakly. Then schedule the WeakStep on the schedule. If the agent is garbage collected early, when it comes time to step it, the WeakStep will simply do nothing. Note that even if the agent is garbage collected, *the WeakStep is still on the schedule until removed.* Thus this is really useful for agents which consume significant memory and are worthwhile garbage collecting if necessary.

sim.engine.WeakStep Constructor Methods

public WeakStep(Steppable agent)
Builds a WeakStep for a given agent. The agent will be held as a weak reference.

WeakStep can also be scheduled repeating: and if the agent is garbage collected, the WeakStep can be set up to automatically stop itself from being rescheduled further. To do this the WeakStep needs to know the appropriate Stoppable, as in:

```
WeakStep weak = new WeakStep(myAgent);
Stoppable stop = mySchedule.scheduleRepeating(weak);
weak.setStoppable(stop);
```

sim.engine.WeakStep Methods

public void setStoppable(Stoppable stop)
Sets the optional stoppable for the WeakStep.

---

sim.engine.MultiStep When its step(...) method is called, this class does one of the following:

- Steps a subsidiary agent *N* times.
- Steps the subsidiary agent only every *N*th time.

The action taken depends on what you provide to the constructor:

sim.engine.MultiStep Constructor Methods

public MultiStep(Steppable agent, int n, boolean countdown)
If *countdown* is false, creates a MultiStep which steps its subsidiary agent *n* times whenever the step(...) method is called. If *countdown* is true, creates a MultiStep which steps its subsidiary agent only once every *n* times the step(...) method is called.

If you choose the second option, MultiStep sets a countdown timer to *n*. Each time step(...) is called, *n* is decreased by one. When it reaches zero, it's reset to *n* and the subsidiary agent is stepped. You can reset the timer to *n* at any time by calling:

sim.engine.MultiStep Methods

public void resetCountdown()
Resets the internal countdown timer for MultiStep.

This method is threadsafe.

sim.engine.MethodStep This convenience class allows you to call specific methods on your agent rather than calling its step(...) method. Before we get into this, let's talk first about the issues raised, some history, and alternative solutions.

The seminal multiagent simulation toolkit was SWARM, written in a combination of Objective-C and TCL/TK. Objective-C has a procedure which enables you to call methods on objects by just specifying the method name as a string.[5] SWARM's scheduler took advantage of this: when an agent was scheduled, SWARM would store not just the agent and the time to step the agent, but also the method to call when the agent was to be stepped.

Repast's scheduler, following the SWARM tradition, does things the same way: when you schedule an agent, you specify the agent, the timestamp, and the method name to call (as a string). In Objective-C this makes sense, but Repast is written in Java and this operation (1) breaks all sorts of Java contracts and (2) is quite slow.

MASON's approach is more rudimentary and much more Java-like: each agent has a *single* dedicated method, called step(...), which can be called. But what if you want to schedule an agent to do different things at different times? For example, what if your agent has a method foo and another bar, and foo must be called on even times, and bar on odd times? You can't name both of them step(...), of course,... right?

Actually you can. Just create an anonymous wrapper Steppable and schedule it instead, like this:

```
final Object myObject = ...
Steppable even = new Steppable() {public void step(SimState state) { myObject.foo(); }};
Steppable odd = new Steppable() {public void step(SimState state) { myObject.bar(); }};
schedule.scheduleRepeating(0.0, even, 2.0);
schedule.scheduleRepeating(1.0, odd, 2.0);
```

Piece of cake. (Note that myObject is final.)

But maybe you don't like this approach and would prefer something more like Repast's, even if it proves slower and a bit funky Java-wise. You can do that with MethodStep. When MethodStep is stepped, it calls a method (which you specify as a string) on a target object. So you could write the above like this:

```
Object myObject = ...
Steppable even = new MethodStep(myObject, "foo");
Steppable odd = new MethodStep(myObject, "bar");
schedule.scheduleRepeating(0.0, even, 2.0);
schedule.scheduleRepeating(1.0, odd, 2.0);
```

Note that foo() and bar(), in this example, must take no arguments. Alternatively MethodStep can pass the SimState as an argument to its method. So if we instead had foo(SimState) and bar(SimState), we could call them like this:

```
Object myObject = ...
Steppable even = new MethodStep(myObject, "foo", true);
Steppable odd = new MethodStep(myObject, "bar", true);
schedule.scheduleRepeating(0.0, even, 2.0);
schedule.scheduleRepeating(1.0, odd, 2.0);
```

I recommend you use the anonymous Steppable instead of MethodStep. It's likely faster and more Java-like. But MethodStep is available for you if you prefer.

[5] Objective-C differs from Java in an important respect for purposes here: whereas in Java you *call* methods on other objects, in Objective-C, you *send* messages to them. Normally these messages correspond to methods in those objects but they don't have to. One effect of this is that in Objective-C you can specify what method you want to call dynamically, using Objective-C's SEL data type:

```
char* fooName = "foo";
SEL callFoo = NSSelectorFromString([NSString stringWithUTF8String:fooName]);
```

Now callFoo contains a message corresponding to a method with the name "foo". You can then send any object this message. In Objective-C anyway.

sim.engine.MethodStep Constructor Methods

public MethodStep(Object target, String methodName)
: Constructs the MethodStep such that when step(...) is called, the method with the name *methodName* is called on the *target*, passing in zero arguments.

public MethodStep(Object target, String methodName, boolean passInSimState)
: Constructs the MethodStep such that when step(...) is called, the method with the name *methodName* is called on the *target*, passing in zero arguments or, if *passInSimState* is true, passing in the SimState as a single argument.

---

sim.engine.AsynchronousSteppable During its step(...) method, this Stoppable Steppable forks off a single thread, which runs *asynchronously from the simulation thread*, and optionally rejoins with the simulation thread at some time in the future. The class is *abstract*: you have to subclass it and override certain methods which will be called when this thread is started or stopped. Here they are:

sim.engine.AsynchronousSteppable Methods

public abstract void run(boolean resuming)
: The entry point to the thread. Called when the thread has just begun, or is being resumed after having been paused. This method is called in the asynchronous thread.

public abstract void halt(boolean pausing)
: Called when the thread is being paused, or is being permanently killed off. This method should notify the thread that it should terminate — that is, that the run(...) method should exit.

The methods above aren't actually called directly by the model: instead the model calls other methods already defined in AsynchronousSteppable to pause, resume, or kill the thread. These methods perform various internal threading magic, then call the methods you've overridden:

sim.engine.AsynchronousSteppable Methods

public void stop()
: Kills the asynchronous thread permanently. This method calls halt(false) to ask the thread to exit, then joins with the asynchronous thread.

public void pause()
: Kills the asynchronous thread temporarily. This method calls halt(true) to ask the thread to exit, then joins with the asynchronous thread.

public void resume()
: Resumes a paused thread. This method calls run(true) to restart the thread.

It shouldn't surprise you that these methods are threadsafe.

So who calls run(false)? The step(...) method does.

After an AsynchronousSteppable has been started via its step(...) method, you *could* call any of the three methods as appropriate. For example, occasionally, you might wish to stop the AsynchronousSteppable at some agreed-upon time in the future. You could do this by posting a Steppable which does this task:

```
final AsynchronousSteppable s = ...
Steppable stopper = new Steppable() { public void step(SimState state) { s.stop(); } }
schedule.scheduleOnce(s....);
schedule.scheduleOnce(stopper....);
```

But usually you just let MASON do it: AsynchronousSteppables register themselves with your SimState and are automatically stopped when super.finish() is called, and likewise paused or resumed when checkpoints are saved out or restarted from. That's the primary reason for having pausing, resuming, and killing in the first place.

Use of AsynchronousSteppable is quite rare. Here most common scenarios in which you might need it, plus some template code to help you:

- The AsynchronousSteppable fires off a task of some sorts which will run for a little bit, on its own CPU, and then die on its own. The time is so short that it's not a big deal if MASON for some reason must wait for it to complete. For example, you might need to write something big out to a file in the background but wish to continue the simulation while it's writing. The typical template for writing code like this would be:

```
AsynchronousSteppable s = new AsynchronousSteppable()
    {
    protected void run(boolean resuming)
        {
        if (!resuming)
            {
            // do your stuff here
            }
        }

    protected void halt(boolean pausing) { } // this stays empty
    };
```

- The AsynchronousSteppable fires off a task which runs forever until killed. The task can be paused or resumed but it doesn't need to distinguish between pausing and killing (or resuming and starting — it's all the same, just starting and stopping). For example, you might create a task which, as long as it's running, periodically says "boo" through the computer loudspeakers. The typical template for writing this code would be:

```
AsynchronousSteppable s = new AsynchronousSteppable()
    {
    boolean shouldQuit = false;
    Object[] lock = new Object[0]; // an array is a unique, serializable object

    protected void run(boolean resuming)
        {
        boolean quit = false;
        while(!quit)
            {
            // do your stuff here -- assuming it doesn't block...

            synchronized(lock) { quit = shouldQuit; shouldQuit = false; }
            }
        // we're quitting -- do cleanup here if you need to
        }

    protected void halt(boolean pausing) { synchronized(lock) { shouldQuit = val; } }
    };
```

- The AsynchronousSteppable fires off a task which runs forever until killed. The task can be paused or resumed. It needs to be able to distinguish between resuming and starting fresh, but doesn't need to distinguish between pausing and killing. For example, you might create a task which writes to a file: when it is resumed, it must reopen the file in appending mode rather than overwrite the original contents. The typical template for writing this code would be:

```
AsynchronousSteppable s = new AsynchronousSteppable()
    {
    boolean shouldQuit = false;
    Object[] lock = new Object[0]; // an array is a unique, serializable object

    protected void run(boolean resuming)
        {
        boolean quit = false;

        if (!resuming)
            {
            // we're starting fresh -- set up here if you have to
            }
        else // (resuming)
            {
            // we're resuming from a pause -- re-set up here if you have to
            }

        while(!quit)
            {
            // do your stuff here -- assuming it doesn't block...

            synchronized(lock) { quit = shouldQuit; shouldQuit = false; }
            }
        // we're quitting -- do cleanup here if you need to
        }

    protected void halt(boolean pausing) { synchronized(lock) { shouldQuit = val; } }
    };
```

- The AsynchronousSteppable fires off a task which runs forever until killed. It must know if it's being paused, resumed, killed, or restarted from pause. For example, you might create a task which writes to a file: when it is resumed, it must reopen the file in appending mode rather than overwrite the original contents. If the task is being permanently killed, it also wants to write a final footer to the file. The typical template for writing this code would be:

```
AsynchronousSteppable s = new AsynchronousSteppable()
    {
    boolean shouldQuit = false;
    boolean shouldPause = false;
    Object[] lock = new Object[0]; // an array is a unique, serializable object

    protected void run(boolean resuming)
        {
        boolean quit = false;
        boolean pause = false;

        if (!resuming)
            {
            // we're starting fresh -- set up here if you have to
            }
        else // (resuming)
            {
            // we're resuming from a pause -- re-set up here if you have to
            }

        while(!quit && !pause)
            {
            // do your stuff here -- assuming it doesn't block...

            synchronized(lock)
                {
                quit = shouldQuit;
                shouldQuit = false;
                pause = shouldPause;
                shouldPause = false;
                }
            }

        if (quit)
            {
            // we’re quitting -- do cleanup here if you need to
            }
        else // if (pause)
            {
            // we’re pausing -- do cleanup here if you need to
            }
        }

    protected void halt(boolean pausing)
        {
        synchronized(lock)
            {
            if (pausing) shouldPause = val;
            else shouldQuit = val;
            }
        }
    };
```

4.3.3.1 Under the Hood

AsynchronousSteppable adds itself to a **registry** maintained by the SimState. When the SimState is saved to checkpoint, all registered AsynchronousSteppables are paused, then resumed after saving is complete. When restoring from a checkpoint, AsynchronousSteppables are again resumed. Finally, when the simulation is over, in the finish() method all AsynchronousSteppables are killed off and the registry is cleaned out.

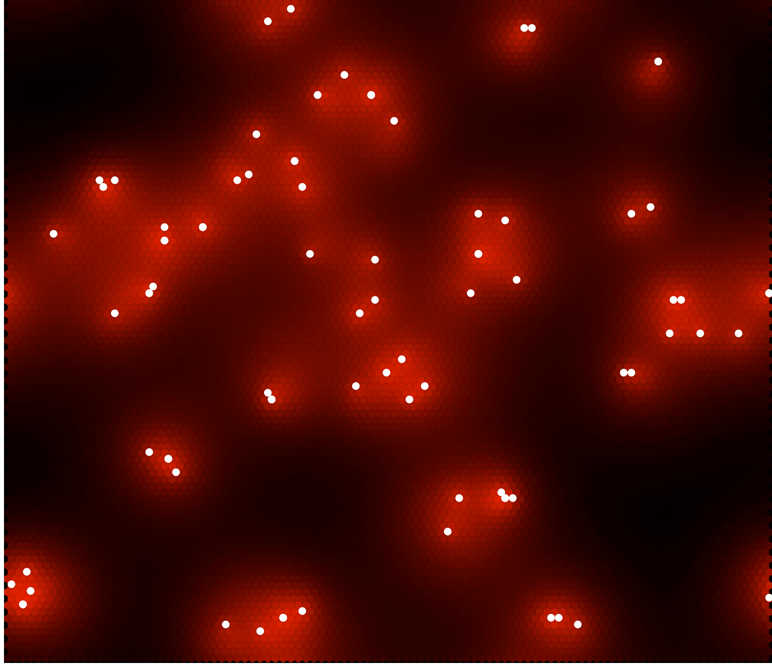
Though you’ll never call them, there are three methods you should be aware of which maintain the registry:

sim.engine.SimState Methods

public boolean addToAsynchronousRegistry(AsynchronousSteppable step)
    Adds the AsynchronousSteppable to the registry. AsynchronousSteppables cannot be added multiple times. Returns true if added, else false if it’s already there or the simulation is over or in the process of running finish().

public void removeFromAsynchronousRegistry(AsynchronousSteppable step)
    Removes the AsynchronousSteppable from the registry, unless the registry is already being cleaned out.

public AsynchronousSteppable[] asynchronousRegistry()
    Returns the registered AsynchronousSteppables as an array.



Chapter 5

Grids

A **grid** is MASON's name for objects arrange in a 2-dimensional or 3-dimensional array or equivalent. MASON supports a wide range of grid environments as fields (representations of space). This includes most any combination of the following:

- 2-dimensional and 3-dimensional grids.
- Rectangular grids, hexagonal grids, and triangular grids.
- Bounded grids, toroidal grids, and (in one case) unbounded grids.
- Grids of doubles, grids of integers, grids of Objects, and sparse grids of Objects (implemented internally using hash tables rather than arrays).

There's nothing magic about many of these representations: but they're helpful for two reasons. First, they present a consistent interface with a variety of helpful functions, and second, MASON has field portrayals already written for them, so you don't have to write one.

5.1 General Topics

Grids are found in the MASON package sim.field.grid; a few minor abstract classes are further found in sim.field. As shown in Figure 5.1, MASON has four basic grid classes, available in both 2-dimensional and 3-dimensional formats:

- sim.field.grid.IntGrid2D and sim.field.grid.IntGrid3D are little more than covers for 2-d and 3-d arrays of ints.
- sim.field.grid.DoubleGrid2D and sim.field.grid.DoubleGrid3D are little more than covers for 2-d and 3-d arrays of doubles.
- sim.field.grid.ObjectGrid2D and sim.field.grid.ObjectGrid3D are little more than covers for 2-d and 3-d arrays of Objects.
- sim.field.grid.SparseGrid2D and sim.field.grid.SparseGrid3D are representations of 2-d and 3-d grids using hashtables, which permit objects to be located at any positive or negative integer location and multiple objects to be stored at the same location.

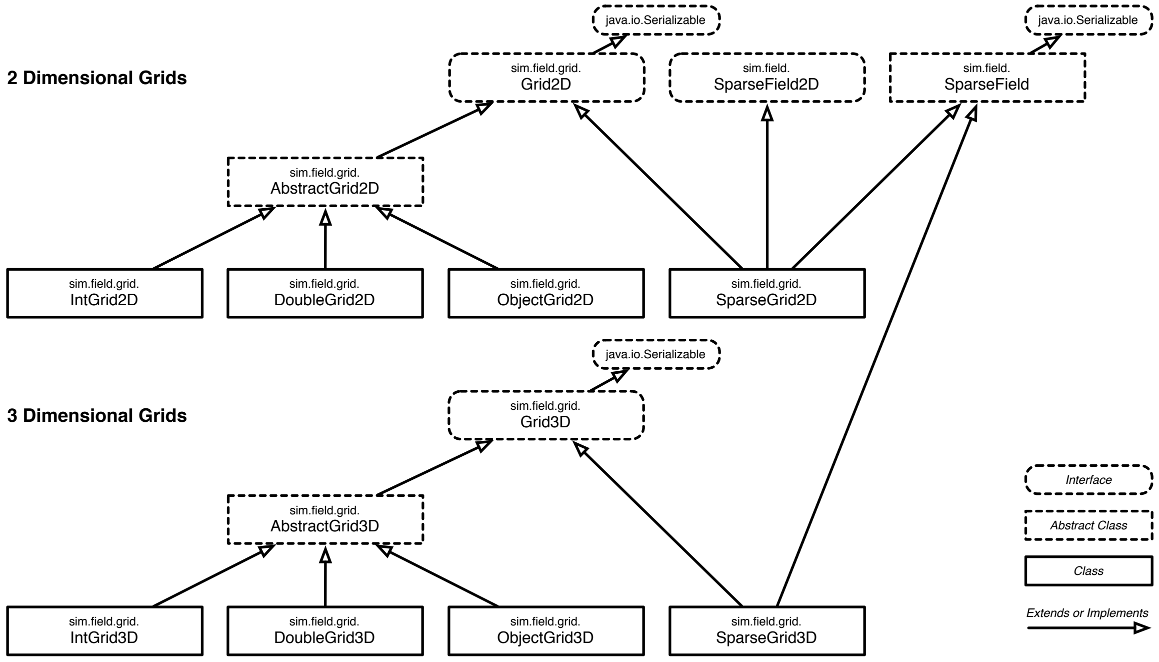

2 Dimensional Grids
java.io.Serializable
sim.field.grid. Grid2D
sim.field. SparseField2D
java.io.Serializable
sim.field. SparseField
sim.field.grid. AbstractGrid2D
sim.field.grid. IntGrid2D
sim.field.grid. DoubleGrid2D
sim.field.grid. ObjectGrid2D
sim.field.grid. SparseGrid2D
3 Dimensional Grids
java.io.Serializable
sim.field.grid. Grid3D
sim.field.grid. AbstractGrid3D
sim.field.grid. IntGrid3D
sim.field.grid. DoubleGrid3D
sim.field.grid. ObjectGrid3D
sim.field.grid. SparseGrid3D
Interface
Abstract Class
Class
Extends or Implements

Figure 5.1 UML diagram of MASON's 2-dimensional and 3-dimensional grid classes.

- sim.field.grid.DenseGrid2D is a representation of 2-d grids using double arrays of Bags, which permits multiple objects to be located at the same location.

These classes extend certain abstract classes and interfaces. All 2-dimensional grids implement the interface sim.field.grid.Grid2D, which provides various utility methods discussed in a moment (similarly all 3-dimensional grids implement sim.field.grid.Grid3D). The bounded 2-dimensional grids (IntGrid2D, DoubleGrid2D, and ObjectGrid) share a common abstract implementation of a number of these methods by subclassing the abstract class sim.field.grid.AbstractGrid2D (and likewise the bounded 3-dimensional grids implement sim.field.grid.AbstractGrid3D).

sim.field.grid.SparseGrid2D (and sim.field.grid.SparseGrid3D) is a bit different. It stores Objects at X, Y (or X, Y, Z) locations using a hash table. Unlike an array, this allows multiple Objects to be stored at the same location, and also allows arbitrarily large grids, indeed unbounded, grids of sparse numbers of objects without the memory overhead incurred by arrays. And you can look up all Objects within a certain distance, and find the location of an object quickly. It's also got a lot more overhead than a simple array.

SparseGrid2D and SparseGrid3D are both manifestations of the same abstract superclass sim.field.SparseField, which handles the general hashtable lookup issues. SparseField has been set up to make it easy for you to implement your own sparse fields if you care to. SparseGrid2D also implements two methods, found in the interface sim.field.SparseField2D which make it more useful to 2-dimensional Portrayals:[1]

sim.field.SparseField2D Utility Methods

public Double2D getDimensions()
    Returns the width and height of the field.

public Double2D getObjectLocationAsDouble2D(Object obect)
    Returns the location, as a Double2D, of a given Object stored in the field.

[1]We've not needed an equivalent for 3-dimensional portrayals yet: but may eventually.

5.1.1 Extents and Neighborhood Lookup

All grids, even unbounded ones, have a width, height, and (for 3-dimensional grids) length which define an extent for the grid. For finite grids, this extent is the actual extent of the grid. For unbounded grids, the width and height allow the grid to be used as a finite or toroidal grid, and also act as a convenience to let MASON know the appropriate default region to display on-screen. The width and height do not prevent you from setting values outside the extent: but they are used in gathering neighbors when toroidal facilities are turned on.

sim.field.grid.Grid2D and sim.field.grid.Grid3D Utility Methods

public int getWidth()
    Returns the width of the field (X dimension).

public int getHeight()
    Returns the width of the field (Y dimension).

Additional sim.field.grid.Grid3D Utility Methods

public int getLength()
    Returns the width of the field (Z dimension).

5.1.2 Rectangular Grids

Unless they are unbounded grids (notably sim.field.SparseField2D and sim.field.SparseField3D, which can have negative values), all rectangular grids start at coordinate 0 and proceed in a positive direction. It's also probably best to think of grids in traditional matrix format, that is, as having their origin in the top let corner, with the positive Y axis pointing down. This is because of how MASON displays them with its portrayals, which in turn is because Java's graphics coordinate system is flipped in the Y axis and has $\langle 0, 0\rangle$ at the top left corner.

All rectangular grids have neighborhood lookup facilities which gather all the locations (not their values) in the grid lying within some region a distance away from a given point. MASON provides two such methods:

sim.field.grid.Grid2D Utility Methods

public void getNeighborsMaxDistance(int x, int y, int dist, boolean toroidal, IntBag xPos, IntBag yPos)
    Computes the neighboring locations lying within the $(2\,\mathsf{dist}+1) \times (2\,\mathsf{dist}+1)$ square centered at x, y. That is, all neighbors $\langle X_i, Y_i\rangle$ of a location that satisfy $\max(|(\mathsf{x}-X_i)|, |(\mathsf{y}-Y_i)|) \leq \mathsf{dist}$. If $\mathsf{dist}=1$, this is equivalent to the center location itself and its eight neighbors. If toroidal is true, then the environment is toroidal and the square is permitted to wrap around. For each such neighboring location $\langle X_i, Y_i\rangle$, the values X_i and Y_i are added to xPos and yPos respectively, clearing them first.

public void getNeighborsHamiltonianDistance(int x, int y, int dist, boolean toroidal, IntBag xPos, IntBag yPos)
    Computes the neighboring locations lying within the $(2\,\mathsf{dist}+1) \times (2\,\mathsf{dist}+1)$ diamond centered at x, y. That is, all neighbors $\langle X_i, Y_i\rangle$ of a location that satisfy $|(\mathsf{x}-X_i)| + |(\mathsf{y}-Y_i)| \leq \mathsf{dist}$. If $\mathsf{dist}=1$, this is equivalent to the center location itself and its "Von-Neuman Neighborhood" (the four neighbors above, below, and to the left and right). If toroidal is true, then the environment is toroidal and the diamond is permitted to wrap around. For each such neighboring location $\langle X_i, Y_i\rangle$, the values X_i and Y_i are added to xPos and yPos respectively, clearing them first.

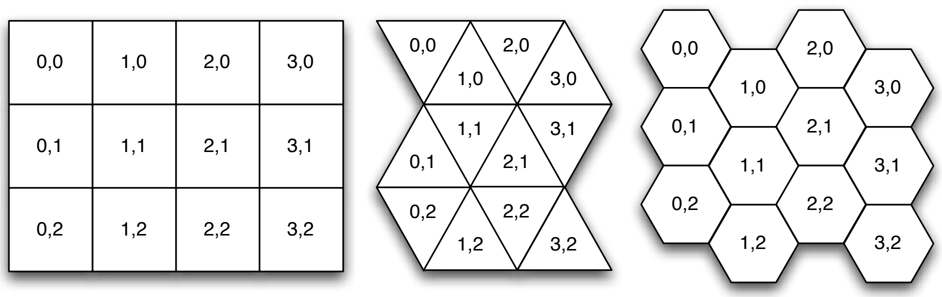


Figure 5.2 Rectangular, triangular, and hexagonal grids topologies with coordinate equivalencies.

public void getNeighborsHexDistance(int x, int y, int dist, boolean toroidal, IntBag xPos, IntBag yPos)

Computes the neighboring locations located within the hexagon centered at x, y $2 \times$ variabledist $+ 1$ cells from point to opposite point inclusive. If dist $= 1$, this is equivalent to the six neighbors immediately surrounding x, y, plus x, y itself. If toroidal is true, then the environment is toroidal and the diamond is permitted to wrap around. For each such neighboring location $\langle X_i, Y_i \rangle$, the values X_i and Y_i are added to xPos and yPos respectively, clearing them first.

---

sim.field.grid.Grid3D Utility Methods

public void getNeighborsMaxDistance(int x, int y, int z, int dist, boolean toroidal, IntBag xPos, IntBag yPos, IntBag zPos)

Computes the neighboring locations lying within the $(2\ \text{dist} + 1) \times (2\ \text{dist} + 1) \times (2\ \text{dist} + 1)$ cube centered at x, y, z. That is, all neighbors $\langle X_i, Y_i, Z_i \rangle$ of a location that satisfy $\max(|(\text{x} - X_i)|, |(\text{y} - Y_i)|, |(\text{z} - Z_i)|) \leq \text{dist}$. If dist$= 1$, this is equivalent to the center location itself and its eight neighbors. If toroidal is true, then the environment is toroidal and the cube is permitted to wrap around. For each such neighboring location $\langle X_i, Y_i, Z_i \rangle$, the values X_i, Y_i, and Z_i are added to xPos, yPos, and zPos respectively, clearing them first.

public void getNeighborsHamiltonianDistance(int x, int y, int z, int dist, boolean toroidal, IntBag xPos, IntBag yPos, IntBag zPos)

Computes the neighboring locations lying within the $(2\ \text{dist} + 1) \times (2\ \text{dist} + 1) \times (2\ \text{dist} + 1)$ diamond-shaped volume centered at x, y, z. That is, all neighbors $\langle X_i, Y_i, Z_i \rangle$ of a location that satisfy $|(\text{x} - X_i)| + |(\text{y} - Y_i)| + |(\text{z} - Z_i)| \leq \text{dist}$. If dist$= 1$, this is equivalent to the center location itself and its "Von-Neuman Neighborhood" (the four neighbors above, below, and to the left and right). If toroidal is true, then the environment is toroidal and the diamond is permitted to wrap around. For each such neighboring location $\langle X_i, Y_i, Z_i \rangle$, the values X_i, Y_i, and Z_i are added to xPos, yPos, and zPos respectively,

---

5.1.3 Hexagonal, Triangular, and Toroidal Grids

MASON handles toroidal (wrap-around) grids via two kinds of methods to determine toroidal neighbors. The slower general methods (tx(..), ty(..), and tz(..)) work for all toroidal situations. The faster methods (stx(..), sty(..), and stz(..)) assume that you will never query locations far from the extent (width, height, length) of the grid.

MASON doesn't have special classes for hexagonal or triangular grids. It just uses the regular bounded rectangular ones, and packs the hexagonal and triangular matrices into them. Hexagons and triangles are quite easily packed into a rectangular matrix. See Figure 5.2 to see how MASON does it. Furthermore **if your grid width is even, these packings work fine in toroidal fashion as well**.

To use MASON's grids in triangular form or hexagonal form, your primary need is how to get around from cell to cell. For hexagonal grids, this amounts to knowing how to get to your six neighbors. For triangular grids, the left and right neighbors are obvious (just add or subtract 1 from your X value), but you need to know the *nature* of your triangle — is it pointing "up" or "down" — to understand where your third neighbor is. For these tasks, MASON provides utility methods described below. Additionally, MASON provides a neighborhood lookup function for hexagonal grids:

sim.field.grid.Grid2D Utility Methods

public int tx(int x)
  Returns the value of x wrapped into within the width of the grid.

public int ty(int y)
  Returns the value of y wrapped into within the height of the grid.

public int stx(int x)
  Returns the value of x wrapped into within the width of the grid. Faster than tx(...). Assumes that $-(\text{width}) \leq x \leq 2(\text{width})$.

public int sty(int y)
  Returns the value of y wrapped into within the width of the grid. Faster than ty(...). Assumes that $-(\text{height}) \leq y \leq 2(\text{height})$.

public boolean trb(int x, int y)
  Returns whether the triangular cell packed at x, y has its horizontal edge on the bottom. Always true if $x + y$ is odd.

public boolean trt(int x, int y)
  Returns whether the triangular cell packed at x, y has its horizontal edge on the top. Always true when trb(x,y) is false and vice versa.

public int ulx(int x, int y)
  Returns the packed X index of the upper left neighbor of the hexagonal cell found at the packed location $\langle x, y \rangle$.

public int uly(int x, int y)
  Returns the packed Y index of the upper left neighbor of the hexagonal cell found at the packed location $\langle x, y \rangle$.

public int urx(int x, int y)
  Returns the packed X index of the upper right neighbor of the hexagonal cell found at the packed location $\langle x, y \rangle$.

public int ury(int x, int y)
  Returns the packed Y index of the upper right neighbor of the hexagonal cell found at the packed location $\langle x, y \rangle$.

public int dlx(int x, int y)
  Returns the packed X index of the lower left neighbor of the hexagonal cell found at the packed location $\langle x, y \rangle$.

public int dly(int x, int y)
  Returns the packed Y index of the lower left neighbor of the hexagonal cell found at the packed location $\langle x, y \rangle$.

public int drx(int x, int y)
  Returns the packed X index of the lower right neighbor of the hexagonal cell found at the packed location $\langle x, y \rangle$.

public int dry(int x, int y)
  Returns the packed Y index of the lower right neighbor of the hexagonal cell found at the packed location $\langle x, y \rangle$.

public int upx(int x, int y)
  Returns the packed X index of the neighbor directly above the hexagonal cell found at the packed location $\langle x, y \rangle$.

public int upy(int x, int y)
  Returns the packed Y index of the neighbor directly above the hexagonal cell found at the packed location $\langle x, y \rangle$.

public int downx(int x, int y)
  Returns the packed X index of the neighbor directly below the hexagonal cell found at the packed location $\langle x, y \rangle$.

public int downy(int x, int y)
Returns the packed Y index of the neighbor directly below the hexagonal cell found at the packed location $\langle x, y \rangle$.

public void getNeighborsHexagonalDistance(int x, int y, int dist, boolean toroidal, IntBag xPos, IntBag yPos)
Computes the neighboring locations lying within the hexagon centered at x, y and $(2\,\text{dist} + 1)$ cells from point to opposite point, inclusive. If dist$= 1$, this is equivalent to the center hexagon itself and its six immediate neighbors. If toroidal is true, then the environment is toroidal and the hexagon is permitted to wrap around. For each such neighboring location $\langle X_i, Y_i \rangle$, the values X_i and Y_i are added to xPos and yPos respectively, clearing them first.

How to Combine Toroidal and Hexagonal Methods Like this:

```
int upperLeftToroidal = tx(ulx(x,y));
```

3-Dimensional Toroidal Grids 3-dimensional grids have no hexagonal or triangular functions: they just have toroidal functions.

sim.field.grid.Grid3D Utility Methods

public int tx(int x)
Returns the value of x wrapped into within the width of the grid.

public int ty(int y)
Returns the value of y wrapped into within the height of the grid.

public int tz(int z)
Returns the value of z wrapped into within the length of the grid.

public int stx(int x)
Returns the value of x wrapped into within the width of the grid. Faster than tx(...). Assumes that $-(\text{width}) \leq x \leq 2(\text{width})$.

public int sty(int y)
Returns the value of y wrapped into within the width of the grid. Faster than ty(...). Assumes that $-(\text{height}) \leq y \leq 2(\text{height})$.

public int stz(int z)
Returns the value of z wrapped into within the width of the grid. Faster than tz(...). Assumes that $-(\text{length}) \leq z \leq 2(\text{length})$.

5.2 Array Grids

MASON provides six array-as-grid classes: sim.field.grid.DoubleGrid2D, sim.field.grid.DoubleGrid3D, sim.field.grid.IntGrid2D, sim.field.grid.IntGrid3D, sim.field.grid.ObjectGrid2D, and sim.field.grid.ObjectGrid3D. These provide 2- and 3-dimensional arrays of doubles, ints, and Objects respectively.

These classes are organized to encourage you to directly access the underlying array, which is always given the variable name field. For example, in DoubleGrid2D we have:

```
public double[/**x*/][/**y*] field;
```

... and in ObjectGrid3D we have:

```
public Object[/**x*/][/**y*/][/**z*/] field;
```

Are these neighborhood methods fast?

Not particularly. There's a lot of overhead. If you're trying to scan through a neighborhood of cells, I'd do it manually. See the code of the sim.app.heatbugs.Diffuser class in the sim.app.heatbugs application for a long explication on various ways of doing scans faster. If you're trying to scan through all the cells, or a large number of them, see also the Section "**A Hint on Scanning**" below.

You are also welcome to access values via the get(...), set(...), and setTo(...) methods. All six grid classes also have neighborhood lookup classes which not only place neighboring locations in various IntBags (as before) but return all the values at those locations as Bags of Objects, DoubleBags of doubles, or IntBags of ints (depending on the kind of grid class.

For example, DoubleGrid2D we have the following constructors (DoubleGrid3D, IntGrid2D, IntGrid3D, ObjectGrid2D, and ObjectGrid3D are similar):

sim.field.grid.DoubleGrid2D Constructor Methods

public DoubleGrid2D(int width, int height)
Creates a DoubleGrid2D with the given width and height, and an initial value of 0 for all cells.

public DoubleGrid2D(int width, int height, double initialValue)
Creates a DoubleGrid2D with the given width and height, and the given initial value for all cells.

public DoubleGrid2D(DoubleGrid2D values)
Creates a DoubleGrid2D which is a copy of the provided DoubleGrid2D.

Here are some DoubleGrid2D methods which are likewise common to all the above classes:

sim.field.grid.DoubleGrid2D Methods

public final double get(int x, int y)
Returns field[x][y].

public final void set(int x, int y, double val)
Sets field[x][y] to val.

public final DoubleGrid2D setTo(double val)
Sets all values of the field to val. Returns the grid.

public final DoubleGrid2D setTo(DoubleGrid2D values)
Sets all values of the field to those found in values. If values differs in dimensions, the grid is first reallocated to reflect the dimensions in values. Returns the grid.

public final double[] toArray()
Flattens the grid by row-major order into a single array of values and returns it.

public DoubleBag getNeighborsMaxDistance(int x, int y, int dist, boolean toroidal, DoubleBag result, IntBag xPos, IntBag yPos)

Computes the neighboring locations lying within the $(2\ \mathsf{dist} + 1) \times (2\ \mathsf{dist} + 1)$ square centered at x, y. That is, all neighbors $\langle X_i, Y_i \rangle$ of a location that satisfy $\max(|(\mathsf{x} - X_i)|, |(\mathsf{y} - Y_i)|) \leq \mathsf{dist}$. If $\mathsf{dist} = 1$, this is equivalent to the center square itself and its eight neighbors. If toroidal is true, then the environment is toroidal and the square is permitted to wrap around. For each such neighboring location $\langle X_i, Y_i \rangle$, the values X_i and Y_i are added to xPos and yPos respectively, clearing them first.

Then loads into result all the values at those locations and returns it (not clearing it first). None of the various bags need be provided (you can pass in null). If you provide null for result, a DoubleBag will be created and filled, then returned to you.

public DoubleBag getNeighborsHamiltonianDistance(int x, int y, int dist, boolean toroidal, DoubleBag result, IntBag xPos, IntBag yPos)

Computes the neighboring locations lying within the $(2\ \mathsf{dist} + 1) \times (2\ \mathsf{dist} + 1)$ diamond centered at x, y. That is, all neighbors $\langle X_i, Y_i \rangle$ of a location that satisfy $|(\mathsf{x} - X_i)| + |(\mathsf{y} - Y_i)| \leq \mathsf{dist}$. If $\mathsf{dist} = 1$, this is equivalent to the center square itself and its "Von-Neuman Neighborhood" (the four neighbors above, below, and to the left and right). If toroidal is true, then the environment is toroidal and the square is permitted to wrap around. For each such neighboring location $\langle X_i, Y_i \rangle$, the values X_i and Y_i are added to xPos and yPos respectively, clearing them first.

Then loads into result all the values at those locations and returns it (not clearing it first). None of the various bags need be provided (you can pass in null). If you provide null for result, a DoubleBag will be created and filled, then returned to you.

public DoubleBag getNeighborsHexagonalDistance(int x, int y, int dist, boolean toroidal, DoubleBag result, IntBag xPos, IntBag yPos)

Computes the neighboring locations lying within the hexagon centered at x, y and $(2\ \text{dist} + 1)$ cells from point to opposite point, inclusive. If dist= 1, this is equivalent to the center location itself and its six immediate neighbors. If toroidal is true, then the environment is toroidal and the square is permitted to wrap around. For each such neighboring location $\langle X_i, Y_i \rangle$, the values X_i and Y_i are added to xPos and yPos respectively, clearing them first.

Then loads into result all the values at those locations and returns it (not clearing it first). None of the various bags need be provided (you can pass in null). If you provide null for result, a DoubleBag will be created and filled, then returned to you.

The various other classes have equivalent methods (the 3-dimensional classes obviously do not have a hexagonal distance method).

A Hint on Scanning Java doesn't have 2- and 3-dimensional arrays. It has *arrays of arrays* or *arrays of arrays of arrays*. This means that every time you access a 2-dimensional array cell (say) using field[x][y] Java must first verify that field is non-null, then check that x is within the X dimensions of field, then check that field[x] is non-null, then finally check that y is within the Y dimensions of field[x]. For 3-dimensional arrays, this is even worse.

That's a lot of checking. But there's a way to reduce it 2- or 3-fold. If you want to scan through your field, instead of saying:

```
for(int i = 0; i < grid.field.length; i++)
    for(int j=0; j < grid.field[i].length)
        doSomethingWith(grid.field[i][j]);
```

Instead, use the following pattern (using DoubleGrid2D as an example):

```
double[] fieldx = grid.field;
for(int i = 0; i < fieldx.length; i++)
    {
    double[] field = fieldx[i];
    for(int j=0; j < fieldy.length; j++)
        doSomethingWith(fieldy[j]);
    }
```

Likewise, for 3-dimensional grids, you can use (again using DoubleGrid3D as an example):

```
double[] fieldx = grid.field;
for(int i = 0; i < fieldx.length; i++)
    {
    double[] fieldy = fieldx[i];
    for(int j=0; j < fieldy.length; j++)
        {
        double[] fieldz = fieldy[j];
        for(int k=0; k < fieldz.length; k++)
            doSomethingWith(fieldz[k]);
        }
    }
```

Notice too the use of local variables to cut down on instance variable accesses (which are slower).

5.2.1 Grids of Integers

The classes sim.field.grid.IntGrid2D and sim.field.grid.IntGrid3D have a number of additional methods you may find useful for doing bulk modifications and statistics of elements on the grid. Here are the IntGrid2D versions (similar methods are provided for IntGrid3D):

Why aren't these Java Bean Properties, like getMax()?

Because they're too expensive to compute, and otherwise would be a problem if you're inspecting the field in the GUI.

sim.field.grid.IntGrid2D Methods

public int max()
 Returns the maximum value over all cells in the grid.

public int min()
 Returns the minimum value over all cells in the grid.

public double mean()
 Returns the mean value over all cells in the grid.

public IntGrid2D upperBound(int toNoMoreThanThisMuch)
 Bounds all the values in the grid to no more than the value provided. Returns the grid.

public IntGrid2D lowerBound(int toNoLessThanThisMuch)
 Bounds all the values in the grid to no more than the value provided. Returns the grid.

public IntGrid2D add(int withThisMuch)
 Adds the value provided to every cell in the grid. Returns the grid.

public IntGrid2D add(IntGrid2D withThis)
 Adds the provided IntGrid2D values to the grid values. The two grids must be identical in dimension. Returns the grid.

public IntGrid2D multiply(int byThisMuch)
 Multiplies the value provided against every cell in the grid. Returns the grid.

public IntGrid2D multiply(IntGrid2D withThis)
 Multiplies the provided IntGrid2D values to the grid values. This is *not a matrix multiply*, but an element-by-element multiply. The two grids must be identical in dimension. Returns the grid.

5.2.2 Grids of Doubles

The classes sim.field.grid.DoubleGrid2D and sim.field.grid.DoubleGrid3D have the same basic methods as sim.field.grid.IntGrid2D and sim.field.grid.IntGrid3D, with the addition of some further methods for rounding. Here are the DoubleGrid2D versions (similar methods are provided for DoubleGrid3D):

sim.field.grid.DoubleGrid2D Methods

public double max()
 Returns the maximum value over all cells in the grid.

public double min()
 Returns the minimum value over all cells in the grid.

public double mean()
 Returns the mean value over all cells in the grid.

public DoubleGrid2D upperBound(double toNoMoreThanThisMuch)
 Bounds all the values in the grid to no more than the value provided. Returns the grid.

public DoubleGrid2D lowerBound(double toNoLessThanThisMuch)
Bounds all the values in the grid to no more than the value provided. Returns the grid.

public DoubleGrid2D floor()
Sets each value x in the grid to $\lfloor x \rfloor$.

public DoubleGrid2D ceiling()
Sets each value x in the grid to $\lceil x \rceil$.

public DoubleGrid2D truncate()
Sets each value x in the grid to $\begin{cases} \lfloor x \rfloor & \text{if } x \geq 0 \\ \lceil x \rceil & \text{if } x < 0 \end{cases}$

public DoubleGrid2D round()
Rounds each value x in the grid to the nearest integer using java.lang.Math.rint(x).

public DoubleGrid2D add(double withThisMuch)
Adds the value provided to every cell in the grid. Returns the grid.

public DoubleGrid2D add(DoubleGrid2D withThis)
Adds the provided DoubleGrid2D values to the grid values. The two grids must be identical in dimension. Returns the grid.

public DoubleGrid2D multiply(double byThisMuch)
Multiplies the value provided against every cell in the grid. Returns the grid.

public DoubleGrid2D multiply(DoubleGrid2D withThis)
Multiplies the provided DoubleGrid2D values to the grid values. This is *not a matrix multiply*, but an element-by-element multiply. The two grids must be identical in dimension. Returns the grid.

5.2.3 Grids of Objects

The classes sim.field.grid.ObjectGrid2D and sim.field.grid.ObjectGrid3D have rather fewer additional methods than their numerical counterparts. Here are the ObjectGrid2D versions (similar methods are provided for ObjectGrid3D):

sim.field.grid.ObjectGrid2D Methods

public Bag elements()
Loads into a Bag all of the values stored in the array in row-major order, discarding null elements. Returns the Bag. This is different from toArray() in two ways: first, a Bag is returned, and second, the null elements are discarded.

public Bag clear()
Loads into a Bag all of the values stored in the array in row-major order, discarding null elements. Then sets all the values in the array to null. Returns the Bag.

In Section 5.3.2.1 we compare ObjectGrid2D, DenseGrid2D, and SparseGrid2D to give you an idea of when you should pick each over the others.

5.2.4 Grids of Bags of Objects

The class sim.field.grid.DenseGrid2D[2] is different from ObjectGrid2D. ObjectGrid2D is a double array of Objects. But DenseGrid2D is a double array of **Bags of Objects**. This allows you to do various useful things, such as

[2]Note that this class is still somewhat experimental: you'll notice there's no DenseGrid3D for example.

- Place an Object in more than one Location.
- Place an Object at a given Location more than once (a rare need).
- Place multiple Objects in the same Location.

DenseGrid2D isn't used all that much: more common is using the class sim.field.grid.SparseGrid2D, discussed next, which allows similar functionality. In Section 5.3.2.1 we compare ObjectGrid2D, DenseGrid2D, and SparseGrid2D to give you an idea of when you should pick each over the others.

Because DenseGrid2D holds Objects in Bags, you can't just stick them in places manually (well you *can*, but it's not recommended). Instead there are a variety of methods and variables provided for you to do such things. To begin with, let's cover the variables:

```
public Bag[][] field;
public boolean removeEmptyBags = true;
public boolean replaceLargeBags = true;
```

The field variable is the actual field. It's just as we said: a double-array of Bags. Some cells in this field may be null; and some Bags in the field may be empty.

The removeEmptyBags and replaceLargeBags variables let you trade off memory efficiency for a small bit of speed (their default settings are aimed towards memory efficiency). When Objects leave a location on a DenseGrid2D, the Bag at the Location may become much too large for the elements in it. If the ratio of objects to Bag size drops below 1/4, the Bag is replaced. If you set replaceLargeBags to false, the Bag will never be replaced. Second, if the Bag is entirely emptied, by default it is removed and garbage collected and that Location is set to null. If you set removeEmptyBags to false, Bags will never be removed.

My recommendation is to keep the defaults, else you'll get a lot of memory growth and not a huge speed improvement.

So how do you place Objects in the field and move them about? Here are the relevant methods:

sim.field.grid.DenseGrid2D Methods

public Bag getObjectsAtLocation(int x, int y)
Returns the Bag storing all the Objects at a given location, or null if there are no objects (sometimes if there are no objects, an empty Bag may be returned). **The provided Bag is to be treated as read-only and not to be modified, and it may change at any time without warning.** You should use this method only to do quick read-only scans of the field without modification. If you want to modify the Bag, copy it first like this: Bag vals = new Bag(myDenseGrid.getObjectsAtLocation(location));

public Bag getObjectsAtLocation(Int2D location)
Returns the Bag storing all the Objects at a given location, or null if there are no objects (sometimes if there are no objects, an empty Bag may be returned). **The provided Bag is to be treated as read-only and not to be modified, and it may change at any time without warning.** You should use this method only to do quick read-only scans of the field without modification. If you want to modify the Bag, copy it first like this: Bag vals = new Bag(myDenseGrid.getObjectsAtLocation(location));

public int numObjectsAtLocation(int x, int y)
Returns the number of objects stored at the given location.

public int numObjectsAtLocation(Int2D location)
Returns the number of objects stored at the given location.

public void addObjectToLocation(Object obj, int x, int y)
Adds the given object to the given Location. Does not eliminate other instances of the same object at that Location.

public Bag addObjectToLocation(Object obj, Int2D location)
Adds the given object to the given Location. Does not eliminate other instances of the same object at that Location.

public void addObjectsToLocation(Object[] objs, int x, int y)
Adds the given objects to the given Location. Does not eliminate other instances of the same objects at that Location. The array may be null.

public Bag addObjectsToLocation(Object[] objs, Int2D location)
Adds the given objects to the given Location. Does not eliminate other instances of the same objects at that Location. The array may be null.

public void addObjectsToLocation(Bag objs, int x, int y)
Adds the given objects to the given Location. Does not eliminate other instances of the same objects at that Location. The Bag may be null.

public Bag addObjectsToLocation(Bag objs, Int2D location)
Adds the given objects to the given Location. Does not eliminate other instances of the same objects at that Location. The Bag may be null.

public void addObjectsToLocation(Collection objs, int x, int y)
Adds the given objects to the given Location. Does not eliminate other instances of the same objects at that Location. The Collection may be null.

public Bag addObjectsToLocation(Collection objs, Int2D location)
Adds the given objects to the given Location. Does not eliminate other instances of the same objects at that Location. The Collection may be null.

public boolean moveObject(Object obj, int from_x, int from_y, int to_x, int to_y)
Removes the given Object from the given Location ("from"), and adds it to the new location ("to"), then returns true. Only one instance of the object is moved — if the Object is at that Location multiple times, the other instances are left alone. Returns false if the Object does not exist at the original Location ("from").

public boolean moveObject(Object obj, Int2D from, Int2D to)
Removes the given Object from the given Location ("from"), and adds it to the new location ("to"), then returns true. Only one instance of the object is moved — if the Object is at that Location multiple times, the other instances are left alone. Returns false if the Object does not exist at the original Location ("from").

public void moveObjects(int from_x, int from_y, int to_x, int to_y)
Removes all Objects from a given Location ("from"), and adds them to the new location ("to").

public void moveObjects(Int2D from, Int2D to)
Removes all Objects from a given Location ("from"), and adds them to the new location ("to").

public Bag removeObjectsAtLocation(int x, int y)
Removes and returns the Bag storing all the Objects at a given location, or null if there are no objects (sometimes if there are no objects, an empty Bag may be returned). You are free to modify this Bag.

public Bag removeObjectsAtLocation(Int2D location)
Removes and returns the Bag storing all the Objects at a given location, or null if there are no objects (sometimes if there are no objects, an empty Bag may be returned). You are free to modify this Bag.

public boolean removeObjectAtLocation(Object obj, int x, int y)
Removes the given Object *once* from the given Location, and returns true. Returns false if the Object does not exist at the given Location.

public boolean removeObjectAtLocation(Object obj, Int2D location)
Removes the given Object *once* from the given Location, and returns true. Returns false if the Object does not exist at the given Location.

public boolean removeObjectMultiplyAtLocation(Object obj, int x, int y)
Removes all copies of the given Object from the given Location, and returns true. Returns false if the Object does not exist at the given Location.

public boolean removeObjectMultiplyAtLocation(Object obj, Int2D location)
Removes all copies of the given Object from the given Location, and returns true. Returns false if the Object does not exist at the given Location.

public Bag clear()
: Removes all Objects from the field and returns them in a Bag. You may modify this Bag. Note that this is a potentially expensive operation.

Like the various int, double, object, and sparse grids, DenseGrid2D can also provide objects within certain distances of a given point:

sim.field.grid.DenseGrid2D Methods

public void getNeighborsMaxDistance(int x, int y, int dist, boolean toroidal, IntBag xPos, IntBag yPos)
: Computes the neighboring locations lying within the $(2\,\mathsf{dist}+1)\times(2\,\mathsf{dist}+1)$ square centered at x,y. That is, all neighbors $\langle X_i, Y_i\rangle$ of a location that satisfy $\max(|(\mathsf{x}-X_i)|, |(\mathsf{y}-Y_i)|)\leq \mathsf{dist}$. If $\mathsf{dist}=1$, this is equivalent to the center location itself and its eight neighbors. If toroidal is true, then the environment is toroidal and the square is permitted to wrap around. For each such neighboring location $\langle X_i, Y_i\rangle$, the values X_i and Y_i are added to xPos and yPos respectively, clearing them first.

  Then loads into result all the values at those locations and returns it (not clearing it first). None of the various bags need be provided (you can pass in null). If you provide null for result, a Bag will be created and filled, then returned to you.

public void getNeighborsHamiltonianDistance(int x, int y, int dist, boolean toroidal, IntBag xPos, IntBag yPos)
: Computes the neighboring locations lying within the $(2\,\mathsf{dist}+1)\times(2\,\mathsf{dist}+1)$ diamond centered at x,y. That is, all neighbors $\langle X_i, Y_i\rangle$ of a location that satisfy $|(\mathsf{x}-X_i)|+|(\mathsf{y}-Y_i)|\leq \mathsf{dist}$. If $\mathsf{dist}=1$, this is equivalent to the center location itself and its "Von-Neuman Neighborhood" (the four neighbors above, below, and to the left and right). If toroidal is true, then the environment is toroidal and the diamond is permitted to wrap around. For each such neighboring location $\langle X_i, Y_i\rangle$, the values X_i and Y_i are added to xPos and yPos respectively, clearing them first.

  Then loads into result all the values at those locations and returns it (not clearing it first). None of the various bags need be provided (you can pass in null). If you provide null for result, a Bag will be created and filled, then returned to you.

public void getNeighborsHexDistance(int x, int y, int dist, boolean toroidal, IntBag xPos, IntBag yPos)
: Computes the neighboring locations located within the hexagon centered at x, y $2\times\mathsf{variabledist}+1$ cells from point to opposite point inclusive. If $\mathsf{dist}=1$, this is equivalent to the six neighbors immediately surrounding x, y, plus x, y itself. If toroidal is true, then the environment is toroidal and the diamond is permitted to wrap around. For each such neighboring location $\langle X_i, Y_i\rangle$, the values X_i and Y_i are added to xPos and yPos respectively, clearing them first.

  Then loads into result all the values at those locations and returns it (not clearing it first). None of the various bags need be provided (you can pass in null). If you provide null for result, a Bag will be created and filled, then returned to you.

5.3 Sparse Fields and Sparse Grids

A **sparse field** (in MASON terminology) is a many-to-one relationship between **objects** and their **locations** with the additional ability to **scan** through all the stored objects in an efficient manner. MASON provides four sparse fields: sim.field.grid.SparseGrid2D, sim.field.grid.SparseGrid3D, sim.field.continuous.Continuous2D, and sim.field.continuous.Continuous3D. Nearly all of the methods in SparseGrid2D, for example, are implemented in SparseField, so it's important to understand Sparse Fields first.

5.3.1 Sparse Fields

Sparse fields in MASON are implemented with subclasses of the abstract class sim.field.SparseField. This class enables the many-to-one mapping of objects to locations plus scanning, using a combination of two hash tables (implemented with java.util.HashMap) and sim.util.Bag.

Here's how it works. When you store an Object in a SparseField, you associate with it another arbitrary Object (its *location*). The SparseField stores three things:

- The Object is stored in a Bag containing all current Objects.
- The Object is stored as a key in a HashMap, with its value being a special object holding the Location of the Object, and also the index of the Object in the Bag of all currentObjects.
- The Location is stored as a key in a HashMap, with its value being a Bag of all Objects located at that Location. The Object is added to that Bag.

This allows us to do quite a number of things rapidly. We can add, move, test for existence, or remove objects in approximately $O(1)$ time (depending on the number of Objects stored at the same location, which usually negligible). We can also scan through all Objects in $O(n)$ time. We can query all the Objects stored at a given location in $O(1)$ time. We can clear the data structure quickly.

There are some things to know:

- You can't store null in a Sparse Field.
- You can't use null as a location.
- You can't scan through locations efficiently (at present, though we could rig that up).
- There's a lot of constant overhead involved in hashing.

And most important of all:

- Your Objects and Locations must have good hash keys and must not violate hashing rules: if an Object or Location is **mutable** (meaning that its internal values can be modified), it must hash by *reference*. If it is **immutable**, it should hash by value. Generally speaking, Locations should always be immutable.[3]

SparseField implements the following methods for you automatically:

sim.field.SparseField Methods

public boolean exists(Object obj)
  Returns true if the Object has been stored in the field.

public int size()
  Returns the number of Objects stored in the field.

public final Bag getObjectsAtLocation(Object location)
  Returns all Objects stored at the given Location, or null if there are no Objects. **The provided Bag is to be treated as read-only and not to be modified, and it may change at any time without warning.** You should use this method only to do quick read-only scans of the field without modification. If you want to modify the Bag, copy it first like this: `Bag vals = new Bag(mySparseField.getObjectsAtLocation(location));`

public final int numObjectsAtLocation(Object location)
  Returns the number of Objects stored at the given Location.

public Bag removeObjectsAtLocation(Object location)
  Removes and returns the number of Objects stored at the given Location, or null if there are no Objects.

[3]This is why MASON has sim.util.Double2D, sim.util.Double3D, sim.util.Int2D, and sim.util.Int3D: because similar classes found elsewhere in the Java standard libraries — like java.awt.Point — violate hash rules with gusto. Use MASON's versions of these classes as locations.

public final Bag getObjectsAtLocationOfObject(Object obj)
: Returns all Objects stored at the Location of the given Object, or null if the Object is not stored in the SparseField. **The provided Bag is to be treated as read-only and not to be modified, and it may change at any time without warning.** You should use this method only to do quick read-only scans of the field without modification. If you want to modify the Bag, copy it first like this: Bag vals = new Bag(mySparseField.getObjectsAtLocationOfObject(obj));

public final int numObjectsAtLocationOfObject(Object obj)
: Returns the number of Objects stored at the Location of the given Object.

public Object remove(Object obj)
: Removes and returns the given Object, else null if there is no such Object.

public Bag clear()
: Loads into a Bag all of the Objects stored in the field, then removes all of them from the field, then returns the Bag.

public Bag getObjectsAtLocations(Bag locations, Bag result)
: Places into the result Bag all objects found at any of the given locations, and returns the result. You may provide null for the result Bag, in which case a Bag is created for you and returned.

public final Bag getAllObjects()
: Returns all objects stored in the field. **The provided Bag is to be treated as read-only and not to be modified, and it may change at any time without warning.** You should use this method only to do quick read-only scans of the field without modification. If you want to modify the Bag, copy it first like this: Bag vals = new Bag(mySparseField.getAllObjects());

public int getObjectIndex(Object obj)
: Returns the index where the Object may be found in the Bag provided by the method getAllObjects().

public Iterator iterator()
: Returns an Iterator over the Bag provided by the method getAllObjects(). Iterators are slow, so this is largely a convenience method.

public Iterator locationBagIterator()
: Returns an Iterator over all Locations at which Objects are stored. For each such Location, a Bag is provided which holds all the Objects at that Location. **The provided Bags are to be treated as read-only and not to be modified, and it may change at any time without warning.** You should use this method only to do quick read-only scans of the field without modification. If you want to modify a Bag, copy it first like this: Bag vals = new Bag(bagFromIterator); Iterators are slow in general, and this Iterator is particularly slow, so this is largely a convenience method.

5.3.1.1 Tuning

SparseField has two parameters which you can modify to trade speed for memory a little bit:

```
public boolean removeEmptyBags = true;
public boolean replaceLargeBags = true;
```

For each Location where an Object is located, SparseField maintains a Bag holding all Objects at that Location. Bags start out 16 in size and can grow in size as more Objects are located at that Location all at once. If many Objects move elsewhere, this Bag may be very large in size, yet hold very little, wasting memory. By default, MASON shrinks the size of the Bag to 1/2 its size when it has dropped below 1/4 full and only if it is over 32 in size.

If all Objects vacate a Location, SparseField by default deletes the Bag entirely. Of course, if Objects return to the Location, SparseField will have to re-build the Bag as a result.

If you have the rare phenomenon where Objects repeatedly migrate to, then leave, a given Location, you may wish to modify these two parameters so SparseField does not spend so much time creating, shrinking, and deleting Bags. On the other hand, by doing so you're wasting memory: large Bags take up a lot of room, and SparseFields can potentially have an infinite number of them! So think twice before doing so.

5.3.1.2 Under the Hood

You can skip this if you like.

SparseField subclasses largely differ based on the type of Location they allow. To implement a SparseField subclass, you only need to customize two methods based on that type.

- public *LocationType* getObjectLocation(Object obj) will return the location of the given Object.
- public boolean setObjectLocation(Object obj, *LocationType* location) will set an Object to a given Location and return true. If the Object or Location is null, or if some other error occurs such that the Object cannot be set to that Location, then false is returned.

For example, let's say you want to implement a SparseField where the type is real valued doubles between 0.0 and 1.0 inclusive. You could write it like this:

```
package sim.app.fieldexample;
import sim.field.SparseField;

public class BoundedRealSparseField extends SparseField
    {
    public Double getObjectLocation(Object obj)
        {
        return (Double) super.getRawObjectLocation(obj);
        }

    public boolean setObjectLocation(Object obj, Double location)
        {
        double d = location.doubleValue();
        if (d >= 0.0 && d <= 1.0)      // it's a valid location
                return super.setObjectLocation(obj, location);
        else return false;
        }
    }
```

Notice that this implementation relies on two additional methods in SparseField which are normally only used by implementing subclasses. There are actually three such methods:

sim.field.SparseField Implementation Methods

protected final Object getRawObjectLocation(Object obj)
Returns the location of a given Object.

protected final Bag getRawObjectsAtLocation(Object location)
Returns all the objects at a given location as a Bag which should not be modified (it's used internally). This method is called by getObjectsAtLocation(...) and by all internal methods (instead of calling getObjectsAtLocation(...).

protected boolean setObjectLocation(Object obj, Object location)
Sets the Location of a given Object and returns true. If either the Object or Location is null, this method fails and false is returned.

There's also a helpful abstract constructor:

sim.field.SparseField Abstract Constructor Methods

protected SparseField(SparseField other)
Creates a Sparse Field which is a copy of the provided one. The Sparse Field's hash tables and Bag are cloned, but not the objects stored within them (those are just pointer-copied).

5.3.2 Sparse Grids

The classes sim.field.grid.SparseGrid2D and sim.field.grid.SparseGrid3D are **sparse grids**. These are simply SparseField subclasses where the Location is defined as a point on a 2D or 3D integer grid, specifically either a sim.util.Int2D point or a sim.util.Int3D point. Thus there are very few methods beyond those defined by SparseField: the two required methods (see Section 5.3.1.2) and various convenience methods to make it easier to do coding. The extra methods for sim.field.grid.SparseGrid2D are (sim.field.grid.SparseGrid3D is similar):

sim.field.grid.SparseGrid2D Constructor Methods

public SparseGrid2D(int width, int height)
Creates an empty SparseGrid2D with the given width and height.

public SparseGrid2D(SparseGrid2D values)
Creates a SparseGrid2D which is a copy of the provided SparseGrid2D.

sim.field.SparseGrid2D Methods

public boolean setObjectLocation(Object obj, Int2D location)
Sets the Location of a given Object and returns true. If either the Object or Location is null, this method fails and false is returned.

public Int2D getObjectLocation(Object obj)
Returns the location of a given Object.

public Double2D getObjectLocationAsDouble2D(Object obj)
Returns the location of a given Object converted to a Double2D.

public int numObjectsAtLocation(int x, int y)
Returns number of Objects stored at the Location new Int2D(x,y).

public Bag getObjectsAtLocation(int x, int y)
Returns all the Objects stored at the Location new Int2D(x,y). **The provided Bag is to be treated as read-only and not to be modified, and it may change at any time without warning.** You should use this method only to do quick read-only scans of the field without modification. If you want to modify the Bag, copy it first like this: Bag vals = new Bag(mySparseGrid2D.getObjectsAtLocation(x, y));

public Bag removeObjectsAtLocation(int x, int y)
Removes and returns the number of Objects stored at new Int2D(x,y), or null if there are no Objects.

public boolean setObjectLocation(Object obj, int x, int y)
Sets the Location of a given Object to new Int2D(x,y) and returns true. If the Object is null, this method fails and false is returned.

The (max/Hamiltonian/hexagonal) neighborhood methods and various other utility methods are omitted for brevity, but they're there.

5.3.2.1 Sparse Grids Versus Object Grids and Dense Grids

Sparse Grids, Dense Grids, and Object Grids store objects at locations. Sparse Grids create a many-to-one mapping using hash tables. Object Grids store objects in arrays. Dense Grids store objects in arrays of Bags, why use one or the other?

Object Grids are best when you have any of:

- At most one object per grid Location, and (optionally) multiple Locations per object (though as a hack you could make multiple "wrapper objects" for your Objects and store the wrappers in the Sparse Grid at different locations).
- A need for very fast lookup and modification.
- A need for very fast scans over all Locations in the grid.

Dense Grids are best when you have any of:

- Multiple Objects per Location, *and* multiple Locations per Object.
- Objects which must be storable multiply at the same Location.
- A need for very fast scans over all Locations in the grid.

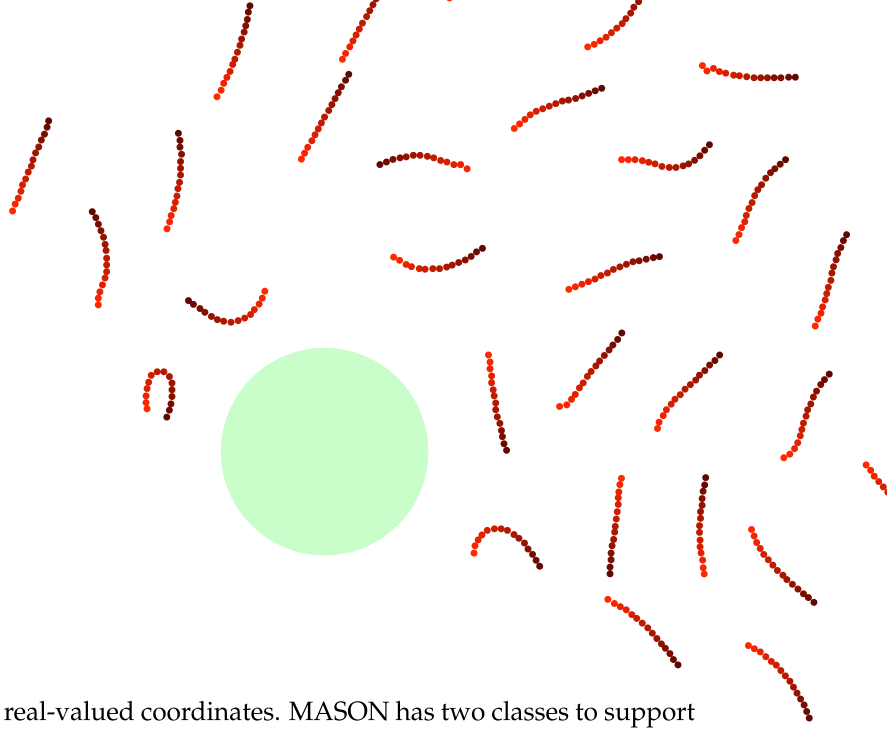
Sparse Grids are good for situations where you have:

- A very large or unbounded grid.
- Relatively few Objects compared to the total number of possible grid Locations.
- (Dense Grids also support this.) Multiple Objects per grid Location.
- A need for very fast scans over all Objects in the grid (though you could of course maintain the list of Objects you've stored in your Object grid yourself).
- The objects are also nodes in a Network (and will be drawn that way). Networks can only be drawn if their nodes are in a SparseGrid or in a Continuous space.
- Fast drawing in the GUI.

Dense or Sparse Grid? The hard choice is between dense and sparse grids. My recommendation: use a sparse grid until speed becomes a real issue, then switch to dense grids (you might get 15% improvement, but with various disadvantages).

Summary Here is a summary of Object Grids, Dense Grids, and Sparse Grids in terms of functionality.

Feature	**Object Grids**	**Dense Grids**	**Sparse Grids**
Supported Grid Types	Bounded, Toroidal	Bounded, Toroidal	Bounded, Toroidal, Unbounded
Objects per Location	0 or 1	Any Number	Any number
Locations per Object	Any number	Any Number	0 or 1
Object May Exist Multiply at a Location	No	Yes	No
Scales With Larger	Grid Sizes	Grid Sizes	Numbers of Agents
Scanning over Objects	No	No	Yes
Scanning over Locations	Yes	Yes	Yes, with constant overhead
Neighborhood Queries	Yes	Yes	Yes, with constant overhead
Adding an Object	Yes	Yes	Yes, with constant overhead
Removing or Moving an Object	Yes	Yes, with constant overhead	Yes, with *more* constant overhead
Moving Objects Without Knowing Location	No	No	Yes
Removing Objects Without Knowing Location	No	No	Yes
Removing All Objects At Location	Yes (1 object)	Yes	Yes, with constant overhead
Removing All Objects	Yes, but slowly	Yes, but slowly	Yes, and fast
Objects can also be Drawn in Networks	No	No	Yes
Speed of Drawing in GUI	Medium	Slow	Fast



Chapter 6

Continuous Space

In continuous space, Objects are associated with real-valued coordinates. MASON has two classes to support continuous space: sim.field.continuous.Continuous2D and sim.field.continuous.Continuous3D. Though they can be used for many purposes, these data structures are tuned for the most common scenario found in multiagent simulations: many small objects sparsely filling a large, often unbounded, continuous region. Continuous2D and Continuous3D associate Objects with locations in the form of 2-dimensional or 3-dimensional real-valued point coordinates. These points are defined by the classes sim.util.Double2D and sim.util.Double3D respectively.

Continuous space is more complicated to implement than grids. The most complex issue arises when performing neighborhood queries. In a grid, the typical query is: "what objects are in the immediate eight neighbors of grid cell X?" Or perhaps "what objects are in the cells located *N* cells away from grid cell X?" These are simple queries to respond to: just sweep through the cells. But in continuous space there are no cells per se: so a typical query is"what are all the objects up to *N* units away from point X?". If *N* varies, than this can be quite challenging to write an efficient data structure for.

The issue is complicated by whether or not the Objects in the continuous space fill a volume. In grid environments, Objects usually fill a single grid cell. But Continuous space may contain either **point objects**, which essentially fill no space at all, or **solid region objects**, which fill some arbitrary-sized area or volume in the space. Solid region objects make neighborhood querying tougher: the question then becomes whether or not the region intersects with the query region.

There are many approaches to solving such problems: for example, quad-trees, KD-trees, various range trees, hierarchical grids, etc. All have advantages and disadvantages. For example, some assume that the environment is static, while others have trade-offs in terms of memory or time overhead, and so on. There's a long history on the topic.

MASON's continuous data structures take a fairly rudimentary and common approach to representing space with neighborhood queries: by discretizing the space into an infinite grid using a fixed discretization size of your choosing. When you store an Object in a continuous space, for example, and associate with

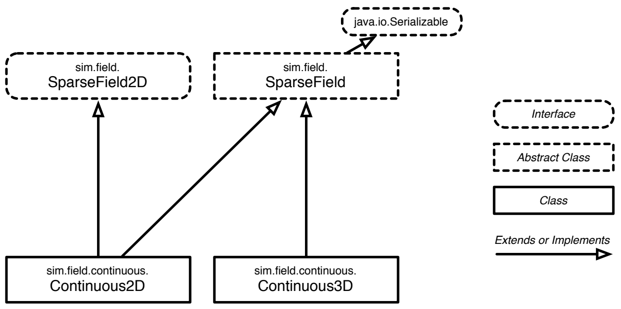


Figure 6.1 UML diagram of MASON's 2-dimensional and 3-dimensional continuous space classes.

What is a Sparse Field? Or a SparseGrid2D?

A Sparse Field is a data structure which associates Objects with Locations using a hash table. See Section 5.3.1 (Sparse Fields). A sim.field.grid.SparseGrid2D is a Sparse Field which represents a 2-dimensional integer grid. See Section 5.3.2.

it a real-valued coordinate, MASON first discretizes this coordinate into an integer coordinate. It then stores the Object in a Sparse Field (essentially a SparseGrid2D). It further retains the real-valued coordinate associated with the Object proper.

For example, if your Object is being stored at $\langle 92.3, -41.4\rangle$ in a sim.field.continuous.Continuous2D with a discretization of 10 units, MASON will first create a discretized coordinate of the form $\langle 9, -5\rangle$, and will then store the Object internally associated with the discretized coordinate. Finally, it will then retain the fact that the Object is actually located at $\langle 92.3, -41.4\rangle$, using a hash table.

Can I use this for vector GIS?

Sure, in theory yes. But vector GIS has its own domain-specific assumptions which are somewhat different, so it wouldn't be particularly efficient. However you're in luck! MASON has an entire vector GIS facility available as a plug-in called **GeoMason**, which rests on a more appropriate facility, the Java Topology Suite (or JTS). See the MASON web page at http://cs.gmu.edu/∼eclab/projects/mason/ for more information.

Why do this? Because it makes neighborhood lookups efficient, if most of your lookups tend to fall within a certain range. Let's say that you've picked a discretization of 10 because you typically want to know all the elements within 4 units in any direction of a target point. That's your typical query. If you ask MASON, for example, for all the Objects within 4 units distance of the point $\langle 32.3, 49.4\rangle$, MASON creates the bounding box from $\langle 28.3, 45.4\rangle$ to $\langle 36.3, 53.4\rangle$. This discretizes to the box of four cells from $\langle 2, 4$ to $\langle 3, 5\rangle$ inclusive. Depending on the query, MASON either looks up the contents of these four cells and either returns them (they might contain Objects further than you asked for), or it whittles the Objects down to just the ones in the region you asked for by testing distance using the Objects' real-valued coordinates. This works reasonably well in practice but there are guidelines you'll need to consider as discussed in the next Section.

MASON's continuous package is sim.field.continuous, and it contains only two classes:

- sim.field.continuous.Continuous2D represents 2-dimensional continuous space.
- sim.field.continuous.Continuous3D represents 3-dimensional continuous space.

These two classes extend the sim.field.SparseField class (Section 5.3.1). We'll repeat some of its methods below. Additionally, sim.field.continuous.Continuous2D implements the sim.field.SparseField2D interface (see Section 5.1), which is used to aid 2-dimensional Field Portrayals. We've not yet found it useful to implement an equivalent thing for sim.field.continuous.Continuous3D:

sim.field.continuous.Continuous2D Utility Methods

public Double2D getDimensions()
    Returns the width and height of the field.

public Double2D getObjectLocationAsDouble2D(Object obect)
    Returns the location, as a Double2D, of a given Object stored in the field.

6.1 Extents

Unlike grids, there's no notion of hexagonal, triangular, or square space here. However continuous fields *do* provide facilities for:

- Bounded space
- Toroidal (wrap-around) bounded space
- Unbounded (infinite) space

Just as was the case for grids, continuous fields implement these facilities via utility methods. Both Continuous2D and Continuous3D have bounds (width, height, and (for Continuous2D) length), even in the case of unbounded space, where the bounds largely exist as a hint for MASON to display the fields on-screen. Once set, the bounds should not be modified.

MASON's continuous space data structures handle toroidal situations in the same way that the grid data structures did: with utility methods to compute toroidal wrap-around values. Additionally, continuous space introduces a common notion of distance: the **Cartesian** or **"as-the-crow-flies"** distance between two points. Toroidal space complicates such distance measurements, because there are several different ways you could connect the two dots in a toroidal environment. Thus MASON also provides a few additional toroidal methods to simplify this calculation. Finally, as a SparseField2D, Continuous2D implements its two utility methods. Here are the Continuous2D versions:

sim.field.continuous.Continuous2D Utility Methods

public double getWidth()
  Returns the width of the field (X dimension).

public double getHeight()
  Returns the height of the field (Y dimension).

public double tx(double x)
  Returns the value of x wrapped into within the width of the field.

public double ty(double y)
  Returns the value of y wrapped into within the height of the field.

public double stx(double x)
  Returns the value of x wrapped into within the width of the field. Faster than tx(...). Assumes that $-(\text{width}) \leq x \leq 2(\text{width})$.

public double sty(double y)
  Returns the value of y wrapped into within the width of the field. Faster than ty(...). Assumes that $-(\text{height}) \leq y \leq 2(\text{height})$.

public double tdx(double x1, double x2)
  Returns the minimum distance in the X dimension between the values x1 and x2 assuming a toroidal environment.

public double tdy(double y1, double y2)
  Returns the minimum distance in the Y dimension between the values y1 and y2 assuming a toroidal environment.

public double tds(Double2D d1, Double2D d2)
  Returns the minimum squared cartesian distance between the locations d1 and d2 assuming a toroidal environment.

public Double2D tv(Double2D d1, Double2D d2)
  Returns the minimum toroidal difference vector between two points. That is, returns new Double2D(tdx(d1.x, d2.x), tdy(d1.y, d2.y)).

Naturally, the Continuous3D class has a few more:

sim.field.continuous.Continuous3D Utility Methods

public double getWidth()
  Returns the width of the field (X dimension).

public double getHeight()
  Returns the height of the field (Y dimension).

public double getDepth()
  Returns the height of the field (Z dimension).

public double tx(double x)
Returns the value of x wrapped into within the width of the field.

public double ty(double y)
Returns the value of y wrapped into within the height of the field.

public double tz(double z)
Returns the value of z wrapped into within the length of the field.

public double stx(double x)
Returns the value of x wrapped into within the width of the field. Faster than tx(...). Assumes that $-(\text{width}) \leq$ x $\leq 2(\text{width})$.

public double sty(double y)
Returns the value of y wrapped into within the width of the field. Faster than ty(...). Assumes that $-(\text{height}) \leq$ y $\leq 2(\text{height})$.

public double stz(double z)
Returns the value of z wrapped into within the width of the field. Faster than tz(...). Assumes that $-(\text{length}) \leq$ z $\leq 2(\text{length})$.

public double tdx(double x1, double x2)
Returns the minimum distance in the X dimension between the values x1 and x2 assuming a toroidal environment.

public double tdy(double y1, double y2)
Returns the minimum distance in the Y dimension between the values y1 and y2 assuming a toroidal environment.

public double tdz(double z1, double z2)
Returns the minimum distance in the Z dimension between the values z1 and z2 assuming a toroidal environment.

public double tds(Double3D d1, Double3D d2)
Returns the minimum squared cartesian distance between the locations d1 and d2 assuming a toroidal environment.

public Double3D tv(Double3D d1, Double3D d2)
Returns the minimum toroidal difference vector between two points. That is, returns new Double2D(tdx(d1.x, d2.x), tdy(d1.y, d2.y), tdz(d1.z, d2.z)).

6.2 Storing, Moving, Looking Up, and Removing Objects

Continuous2D and Continuous3D store Objects in the same way as other Sparse Fields: by associating them with a *location*: in this case, either a Double2D or a Double3D. Here are the methods for Continuous2D. Continuous3D methods are similar:

sim.field.Continuous2D Methods

public boolean exists(Object obj)
Returns true if the Object has been stored in the field.

public int size()
Returns the number of Objects stored in the field.

public final Bag getObjectsAtLocation(Double2D location)
Returns all Objects stored precisely at the given Location, or null if there are no Objects. **Unlike other SparseField implementations, this Bag is yours and you can do with it as you like.**

public final int numObjectsAtLocation(Double2D location)
Returns the number of Objects stored precisely at the given Location.

public Bag removeObjectsAtLocation(Double2D location)
Removes and returns the number of Objects stored precisely at the given Location, or null if there are no Objects.

public final Bag getObjectsAtLocationOfObject(Object obj)
Returns all Objects stored Double2D at the Location of the given Object, or null if the Object is not stored in the SparseField. **Unlike other SparseField implementations, this Bag is yours and you can do with it as you like.**

public final int numObjectsAtLocationOfObject(Object obj)
Returns the number of Objects stored precisely at the Location of the given Object.

public Object remove(Object obj)
Removes and returns the given Object, else null if there is no such Object.

public Bag clear()
Loads into a Bag all of the Objects stored in the field, then removes all of them from the field, then returns the Bag.

public Bag getObjectsAtLocations(Bag locations, Bag result)
Places into the result Bag all objects found precisely at any of the given locations, and returns the result. You may provide null for the result Bag, in which case a Bag is created for you and returned.

public final Bag getAllObjects()
Returns all objects stored in the field. **The provided Bag is to be treated as read-only and not to be modified, and it may change at any time without warning.** You should use this method only to do quick read-only scans of the field without modification. If you want to modify the Bag, copy it first like this: Bag vals = new Bag(myContinuousField.getAllObjects());

public int getObjectIndex(Object obj)
Returns the index where the Object may be found in the Bag provided by the method getAllObjects().

public Iterator iterator()
Returns an Iterator over the Bag provided by the method getAllObjects(). Iterators are slow, so this is largely a convenience method.

public Iterator locationBagIterator()
Returns an Iterator over all **discretized locations** at which Objects are stored. See the next section for discussion of discretization. This Iterator does *not* iterate over Double2D or Double3D locations. For each such Location, a Bag is provided which holds all the Objects at that Location. **The provided Bags are to be treated as read-only and not to be modified, and it may change at any time without warning.** You should use this method only to do quick read-only scans of the field without modification. If you want to modify a Bag, copy it first like this: Bag vals = new Bag(iteratorBag); Iterators are slow in general, and this Iterator is particularly slow, so this is largely a convenience method.

6.3 Discretization

Internally, Continuous2D and Continuous3D do not actually store Objects at their Double2D or Double3D locations. Instead, they define a virtual grid of Int2D or Int3D locations and store Objects in this grid. The Objects still are related with their Double2D and Double3D locations, so it's more or less transparent.

Why do this? Because it's MASON's approach to doing relatively fast neighborhood lookup. Let's say that you typically need to look up agents within a distance of N away from a query point. If MASON has discretized the environment into rectangular cells $2 \times N$ across, then you'd only need to look up (for Continuous2D) four cells to find all possible neighbors in that distance. If MASON has discretized the environment into cells N across, you'd need to look up nine cells. Note that when you receive Objects from a neighborhood lookup, it'll also include Objects which were in the relevant cells but were actually further than N away. Figure 6.2 shows the general idea of Objects stored at discretized locations and associated with real-valued locations.

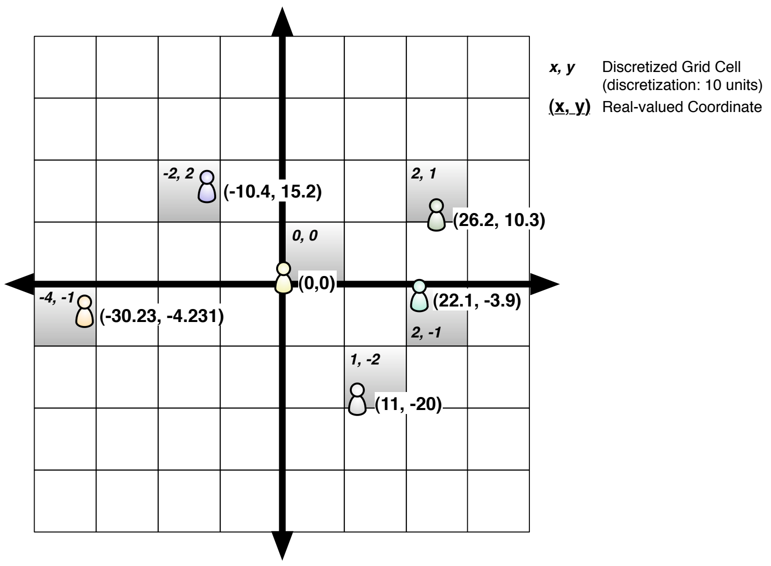


Figure 6.2 Example of discretization of various continuous (real-valued) 2-dimensional coordinates into grid cells. Discretization is ten units. Notice that the $\langle 0, 0\rangle$ grid cell is to the upper right of the $\langle 0, 0\rangle$ real-valued origin. **Important Note:** in order to retain some semblance of sanity, this example retains the Cartesian coordinate system; but MASON will draw Continuous space onscreen in the same way as grids: with the Y-axis flipped and the origin in the top-left corner of the window (per Java coordinate space tradition). It's probably best to keep this in mind.

When you create a Continuous2D or Continuous3D object, you'll be required to set this discretization size. The size you want is largely based on your typical neighborhood lookup requirements. You'll want to pick a size which isn't so small that a great many cells are looked up (each lookup of a cell incurs a hash overhead), but not so large that high percentage of Objects in the cell or cells are in fact further than N and thus not of interest to you.

Usually a discretization of N or $2 \times N$ is recommended as the balance. But you'll need to do some tests to find out the optimum value.

6.3.1 Objects with Area or Volume

If your objects are simply points, the above is sufficient. But if your objects have area or volume, then there's some complexity. Such objects are stored at a *point*, but when you query objects within a distance of N, you don't want to know if that *point* is within N of your query point: but rather typically want to know if *any portion of the object* falls within N away from the query point.

It's up to you to determine whether an Object falls within a certain distance of a query point: but MASON can provide you with all viable candidate Objects using its neighborhood lookup query functions if you **make certain that the discretization size is at least N in size**.

6.3.2 Discretization Methods

Continuous2D and Continuous3D both require you provide a discretization when you create them. Here are their constructors:

sim.field.continuous.Continuous2D Constructor Methods

public Continuous2D(double discretization, double width, double height)
Creates a Continuous2D with the given width, height and discretization.

public Continuous2D(Continuous2D other)
Creates a Continuous2D which is a copy of the provided Continuous2D.

---

sim.field.continuous.Continuous3D Constructor Methods

public Continuous3D(double discretization, double width, double height, double depth)
Creates a Continuous3D with the given width, height, depth and discretization.

public Continuous3D(Continuous3D other)
Creates a Continuous3D which is a copy of the provided Continuous3D.

Additionally, Continuous2D and Continuous3D have certain utility methods for accessing objects in discretized cells or for discretizing a location:

sim.field.continuous.Continuous2D Utility Methods

public Bag getObjectsAtDiscretizedLocation(Int2D location)
Returns a Bag consisting of all the Objects stored at the given discretized location. **The provided Bag is to be treated as read-only and not to be modified, and it may change at any time without warning.** You should use this method only to do quick read-only scans of the field without modification. If you want to modify the Bag, copy it first like this: Bag vals = new Bag(myContinuousField.getObjectsAtDiscretizedLocation());

public Int2D discretize(Double2D location)
Discretizes the given Double2D location according to the Continuous field's discretization.

---

sim.field.continuous.Continuous3D Utility Methods

public Bag getObjectsAtDiscretizedLocation(Int3D location)
Returns a Bag consisting of all the Objects stored at the given discretized location. **The provided Bag is to be treated as read-only and not to be modified, and it may change at any time without warning.** You should use this method only to do quick read-only scans of the field without modification. If you want to modify the Bag, copy it first like this: Bag vals = new Bag(myContinuousField.getObjectsAtDiscretizedLocation());

public Int3D discretize(Double3D location)
Discretizes the given Double3D location according to the Continuous field's discretization.

6.4 Neighborhood Lookup

Armed with a properly discretized Continuous space, we can now perform neighborhood lookup queries on it. These queries can find nearest neighbors,[1] and look up an exact set or a superset of objects within a given distance of a query point. Here are the Continuous2D versions. The Continuous3D versions are identical except that Continuous3D lacks a nearest neighbors lookup (too expensive).

Be warned that these methods aren't cheap. But if you have a large number of objects and relatively small neighborhood lookups, they're more than worth it. If you're doing large neighborhood lookups and have a fairly small number of objects in the environment, you may instead want to consider just scanning through the AllObjects bag:

```
Bag objs = myContinuousField.getAllObjects(); // don't modify or hold onto this bag!
```

sim.field.continuous.Continuous2D Neighborhood Query Methods

[1]Warning: potentially very expensive, only use in crowded environments.

public Bag getNearestNeighbors(Double2D position, int atLeastThisMany, boolean toroidal, boolean nonPointObjects, boolean radial, Bag result)

Returns the *atLeastThisMany* items closest to the given *position*, plus possibly some other Objects, putting the result in *result* if provided (else generating a new Bag), and returning the result. Pass in the appropriate flags if your field is toroidal or has non-point (area-filling or volume-filling) objects. If you want the region to be searched to be radial, that is, a circular region around the query point, set that flag (almost always you want this), else it will be assumed to be a rectangular region.

public Bag getObjectsExactlyWithinDistance(Double2D position, double distance)

Returns as a new Bag objects precisely at or within the given distance of the query position. Assumes a non-toroidal field and point objects only.

public Bag getObjectsExactlyWithinDistance(Double2D position, double distance, boolean toroidal)

Returns as a new Bag objects precisely at or within the given distance of the query position. Assumes point objects only.

public Bag getObjectsExactlyWithinDistance(Double2D position, double distance, boolean toroidal, boolean radial, boolean inclusive, Bag result)

Places into *result* (or a new Bag if *result* is not provided) all objects precisely within the given distance of the query position. Returns the result. If you want only objects *within* the given distance, but not those exactly *at* the given distance, set the *inclusive* flag to false; else set it to true (almost always you'll want it to be true). Assumes point objects only. Pass in the appropriate flag if your field is toroidal. If you want the region to be searched to be radial, that is, a circular region around the query point, set that flag (almost always you want this), else it will be assumed to be a rectangular region.

public Bag getObjectsWithinDistance(Double2D position, double distance)

Returns as a new Bag objects at or within the given distance of the query position, plus possibly some others. Assumes a non-toroidal field and point objects only.

public Bag getObjectsWithinDistance(Double2D position, double distance, boolean toroidal)

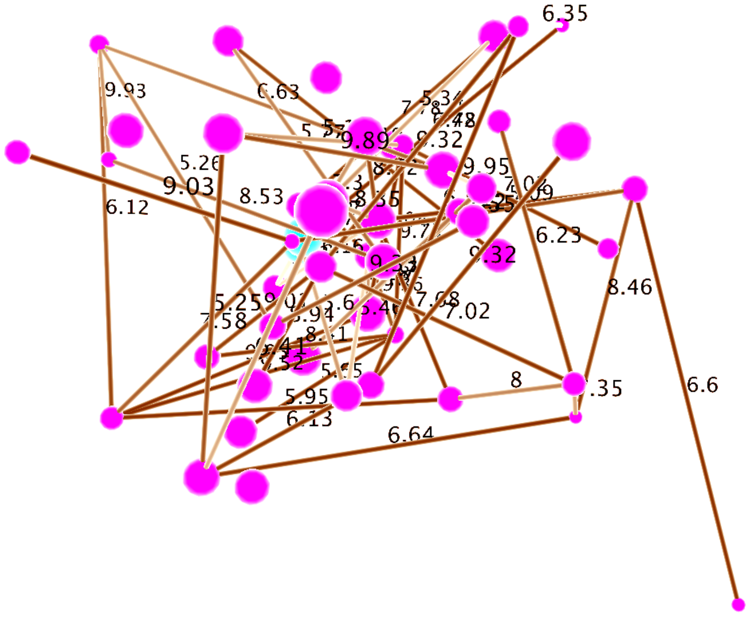
Returns as a new Bag objects at or within the given distance of the query position, plus possibly some others. Assumes point objects only.

public Bag getObjectsWithinDistance(Double2D position, double distance, boolean toroidal, boolean nonPointObjects)

Returns as a new Bag objects at or within the given distance of the query position, plus possibly some others. Pass in the appropriate flags if your field is toroidal or has non-point (area-filling or volume-filling) objects.

public Bag getObjectsWithinDistance(Double2D position, double distance, boolean toroidal, boolean nonPointObjects, Bag result)

Places into *result* (or a new Bag if *result* is not provided) all objects at or within the given distance of the query position, plus possibly some others. Returns the result. Pass in the appropriate flags if your field is toroidal or has non-point (area-filling or volume-filling) objects.



Chapter 7

Networks

MASON has a general-purpose facility, in the sim.field.network package, for graphs and networks of various kinds:

- Graphs and multigraphs.[1]
- Directed and undirected edges.
- Unlabeled, labeled, and weighted edges.
- Arbitrary values for nodes (except null) and for edge labels/weights.
- Dynamically changing graph structures.

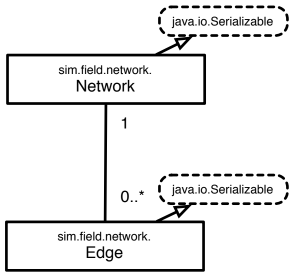


Figure 7.1 UML diagram of MASON's Network package. Not a very complex package!

The network package does *not* presently support hypergraphs[2], nor does it enforce constraints of various kinds (such as connectivity or planar guarantees). Objects in networks are also not associated with physical locations: if you want this, you should place them in a continuous or grid field as well as in a Network.

Graphs in general have two elements: **nodes** and **edges**. Each edge connects two nodes (the two nodes can be the same node). An edge can be optionally **directed**, that is, it specifies that one of its nodes is the "*from*" node and the other is the "*to*" node. An edge can also optionally be **labeled** with an Object. If this Object is a number, the edge is said to be **weighted** with that number.

Hey, I came here looking for ***Social Networks!***

The Network class is very general purpose, and so has no social network facilities by default. However there are two packages you can use with MASON which provide such facilities.

First, there's MASON's own SocialNetworks plug-in, which subclasses Network to provide a fairly rich collection of social network facilities. This package integrates well with MASON and is fast: but it's been only lightly tested, so beware of the possibility of bugs.

Second, MASON can be integrated with Jung, a large and well-regarded Java-based social networks library. You can get more information about both options on the MASON web page at http://cs.gmu.edu/∼eclab/projects/mason/

At right, Figure 7.1 shows the sim.field.network package in its entire two-class glory, consisting of the class sim.field.network.Network, which holds the graph (or network),[3] and the class sim.field.network.Edge. Notice the item that's missing: a class for *nodes*. This is because in the Network class, any Object except for null can be a node in the graph. Edges on the other hand require the distinctive Edge class: however this class can hold an Object as its label or weight.

[1]A multigraph is a graph with potentially more than one edge between two nodes.
[2]A hypergraph is a graph with an edge which connects more than two nodes.
[3]They're synonymous. I'm a computer science guy: we tend to say *graph*.

7.1 Edges

The sim.field.network.Edge class defines an **edge** which connects two **nodes**. Each Edge is associated with exactly one Network, known as its **owner**. And edge contains four basic elements (plus some auxiliary internal indexes to make things faster):

- The "*from*" node. If the network is unweighted, the particular node which is "from" versus "to" is meaningless.
- The "*to*" node. If the network is unweighted, the particular node which is "from" versus "to" is meaningless.
- The info object: the label or weight if any.
- The owner: the Network which controls this Edge.

You cannot change the owner, from, or to values of an Edge, but you are free to change the info object as much as you like. An Edge has the following constructors, though it's rare that you'd create an Edge from scratch rather than let a Network do it for you:

sim.field.network.Edge Constructor Methods

public Edge(Edge other)
  Creates a duplicate of the given edge, but with the owner left unset.

public Edge(Object from, Object to, Object info)
  Creates an Edge from the given object, to the given object, and with the given label or weight (info).

Here's how you access these values:

sim.field.network.Edge Methods

public Network owner()
  Returns the Edge's owner.

public Object getFrom()
  Returns the "from" Object.

public Object getTo()
  Returns the "to" Object.

public int indexFrom()
  Returns the index of the "from" Object in the Edge's owner.

public int indexTo()
  Returns the index of the "to" object in the Edge's owner.

public Object getInfo()
  Returns the "info" Object (the label or weight).

public void setInfo(Object val)
  Sets the "info" Object (the label or weight).

public double getWeight()
  Returns the "info" Object as a weight. If the Object is a subclass of java.lang.Number, its doubleValue() is returned. Else if the Object implements the sim.util.Valuable interface, its doubleValue() is returned. Else 0 is returned. Creates a duplicate of the given edge.

public Edge(Object from, Object to, Object info)
  Creates an Edge from the given object, to the given object, and with the given label or weight (info).

7.2 Using a Network

When you create a Network, you must specify whether or not it is directed:

sim.field.network.Network Constructor Methods

public Network()
    Creates a directed graph.

public Network(boolean directed)
    Creates a directed or undirected graph.

public Network(Network other)
    Creates duplicate copy of an existing Network.

MASON's sim.field.network.Network class stores graphs in a fashion somewhat similar to Sparse Fields (Section 5.3.1): it uses a Bag to hold nodes, and a HashMap to store groups of Edges associated with a given node. Hash lookups incur a significant constant overhead, so this isn't the fastest way to store a graph. But it's a good choice when the graph is constantly changing, as is common in many multiagent simulations.

You can store in the Network either arbitrary objects (as *nodes*) or Edges connecting nodes. If you store an Edge and its associated nodes have not yet been entered into the Network, they'll be entered at that time: additionally, the Edge's owner will be set (permanently). You can also modify an Edge once it's in the Network, changing its "from", "to" and "info" (label or weight) objects to new values. Again, if the "from" and "to" objects are not yet in the Network, they'll be added as nodes.

sim.field.network.Network Methods

public void addNode(Object node)
    Adds a node to the Network.

public void addEdge(Object from, Object to, Object info)
    Adds an edge to the network connecting the "from" node to the "to" node. If either of these nodes has not yet been added to the Network, it will be at this time. You can also specify the "info" (label or weight) value for the edge, or null if there is none.

public void addEdge(Edge edge)
    Adds an edge to the network. If either of the nodes specified in the Edge has not yet been added to the Network, it will be at this time.

public Edge updateEdge(Edge edge, Object from, Object to, Object info)
    Modifies an existing edge in the Network, changing its "from", "to", and "info" values.

Similarly you can remove an Edge or a node. If an Edge is removed, its ownership is reset and it's eligible to be added to a different Network. If a Node is removed, all Edges associated with that node are deleted.

sim.field.network.Network Methods

public void removeNode(Object node)
    Removes a node from the Network, and deletes all associated Edges.

public void removeEdge(Edge edge)
    Removes an edge from the Network and returns it. This Edge may now be added to a different Network if you like.

public Bag clear()
    Removes all nodes from the network, deleting all Edges as well. Returns a Bag consisting of the nodes, which you are free to modify.

Network also contains various methods for querying Nodes and Edges.

sim.field.network.Network Methods

public Bag getAllNodes()
: Returns all the nodes in the Network as a Bag. **The provided Bag is to be treated as read-only and not to be modified, and it may change at any time without warning.** You should use this method only to do quick read-only scans of the field without modification. If you want to modify the Bag, copy it first like this: Bag vals = new Bag(myNetwork.getAllNodes());

public Iterator iterator()
: Returns an Iterator over all nodes provided by the getAllNodes() method. Iterators are slow, so this is largely a convenience method.

public Bag getEdgesIn(Object node)
: Returns in a Bag all the incoming edges to the given node (that is, those for which the node is the "to" object), or, if the Network is undirected, returns a Bag of all edges associated with the node. **The provided Bag is to be treated as read-only and not to be modified, and it may change at any time without warning.** You should use this method only to do quick read-only scans of the field without modification. If you want to modify the Bag, copy it first like this: Bag vals = new Bag(myNetwork.getEdgesIn(myNode));

public Bag getEdgesOut(Object node)
: Returns in a Bag all the outgoing edges from the given node (that is, those for which the node is the "from" object), or, if the Network is undirected, returns a Bag of all edges associated with the node. **The provided Bag is to be treated as read-only and not to be modified, and it may change at any time without warning.** You should use this method only to do quick read-only scans of the field without modification. If you want to modify the Bag, copy it first like this: Bag vals = new Bag(myNetwork.getEdgesOut(myNode));

public Bag getEdges(Object node, Bag result)
: Returns in the Bag *result*, or (if all the outgoing edges from the given node (that is, those for which the node is the "from" object). **The provided Bag is to be treated as read-only and not to be modified, and it may change at any time without warning.** You should use this method only to do quick read-only scans of the field without modification. If you want to modify the Bag, copy it first like this: Bag vals = new Bag(myNetwork.getEdgesOut(myNode));

7.2.1 Adjacency Lists and Adjacency Matrices

If you want faster graph access, and you know that your graph isn't going to change in its structure, you can export the Network to an adjacency list or an adjacency matrix. These are well-known standard graph representations using arrays:

sim.field.network.Network Methods

public Edge[][] getAdjacencyList(boolean outEdges)
: Returns an adjacency list of Edges. This is an array of Edge arrays, one Edge array for each node in the Bag provided in getAllNodes(), and ordered in the same way. If *outEdges* is true, then outgoing edges from nodes are provided in the Edge array, else incoming edges are provided. This list is not updated when the Network is modified, though Edge objects may be updated. Building this list is an $O(E)$ operation, where E is the number of edges.

public Edge[][] getAdjacencyMatrix()
: Returns an adjacency matrix of Edges for a regular graph (not a multigraph). If you have N nodes, this is an $N \times N$ double array of Edges, each Edge connecting two nodes. If the graph is directed, the matrix is organized so that the first dimension is *from* and the second dimension is *to*. If the graph is undirected, the same Edge is provided both places. If the graph is in fact a multigraph, then an arbitrary edge is chosen among those connecting any two nodes. This matrix is not updated when the Network is modified, though Edge objects may be updated. Building this matrix is an $O(E \times N^2)$ operation, where E is the number of edges.

public Edge[][][] getMultigraphAdjacencyMatrix()

Returns an adjacency matrix of Edges for a multigraph. If you have N nodes, this is an $N \times N$ double array of Edge arrays, each Edge array holding those edges which connect two nodes. If the graph is directed, the matrix is organized so that the first dimension is *from* and the second dimension is *to*. If the graph is undirected, the same Edges are provided both places. This matrix is not updated when the Network is modified, though Edge objects may be updated. Building this matrix is an $O(N^3 + E \times N^2)$ operation, where E is the number of edges.

Chapter 8

Making a GUI

MASON's GUI control and visualization facilities are divided into five major packages:

- sim.display: top-level controllers and 2D display facilities.
- sim.display3d: top-level 3D display facilities.
- sim.portrayal: *portrayals* which draw agents and fields in 2D.
- sim.portrayal3d: *portrayals* which draw agents and fields in 3D.
- sim.portrayal.inspector: *inspectors* to inspect, track, and tweak model parameters and field objects.

MASON enforces a very strict separation of model from GUI control and visualization. MASON models generally know nothing of visualization (unless you have endowed certain objects with the ability to portray themselves visually, as discussed in the Section 9.3). **This is a good thing.** It allows MASON models to be run with or without visualization, to be separated at any time from visualization and moved to a non-visualization environment (a back-end supercomputer server, say), or have visualization changed at any time mid-model. It's a basic feature of any good high-performance modeling library.

But this division can be confusing to people coming from environments, such as NetLogo, where the model and the visualization are tightly entwined. So before we discuss the **control** of models (in this Chapter) and then later the **visualization** of models (in Chapters 9 and 11) and the inspection of their elements (Chapter 10), first, let's talk a bit about how the model and its visualization/control are related.

The Figure at right shows the basic elements of the **model** (the **schedule** with its **agents**, the **random number generator**, and **fields** holding various **objects**. All these elements are encapsulated, ultimately, within a single instance of a subclass of sim.engine.SimState. More detail of the controller section (discussed in this Chapter) is shown in Figure 8.2.

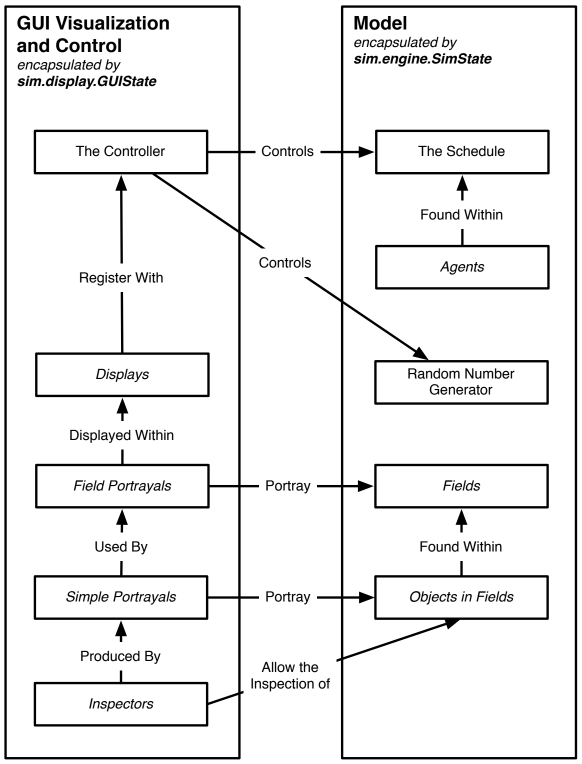


Figure 8.1 Primary relationships between model and GUI visualization/control.

When we build a GUI for the model, we pair with many of these model objects an object whose job is to control or visualize them. The schedule and random number generator are controlled by a **controller**, the most common of which is the sim.display.Console. One or more **displays** show whole fields in a GUI window. Displays are capable of producing movies or screenshots of their fields. The fields are drawn by **field portrayals**, possibly overlaid on one another. In order to draw the fields, the Field Portrayals often (but not always) rely on the services of **simple portrayals** designed to draw or control specific kinds of objects. When the user wishes to **inspect**[1] an object, the associated SimplePortrayal might produce an **inspector** to provide the GUI widgets to do the job. Inspectors can call forth other inspectors to track the object, chart it, etc.

All GUI elements in a visualized MASON model are also encapsulated by an instance of a subclass of a special class: in this case, the class sim.display.GUIState. The GUIState object knows about the SimState model object, but not the other way around, and ultimately encapsulates all the elements mentioned above to visualize and control the model.

8.1 Encapsulation of the Model, Control, and Visualization

When you build a GUI for your model, you'll begin by creating a subclass of sim.display.GUIState, of which MASON will build a single instance to encapsulate your control and visualization of the model. It's traditional in MASON, but not required, that if your SimState model is called Foo, your GUIState subclass will be called FooWithUI.

The GUIState contains only three variables:

```
public SimState state;
pubic Controller controller;
public HashMap storage;
```

The first item is a pointer to the model itself. The second item is a pointer to the **controller** which starts and stops the model (among other things). The final item is a HashMap of elements for those occasional widgets which need to stash a global element somewhere (MASON frowns on using static variables).

At present the only widgets which use the storage variable are certain charting inspectors which need to retain the charts open at a given time. You probably shouldn't mess around with the storage variable unless you're making plug-in inspectors as well. In that case, the rule is: you can put things in the HashMap, and take them out and change them, but you can't modify or take out anything someone *else* put in there.

When you subclass GUIState (and you will), you can add any additional fields you like. A GUIState is instantiated like this:

sim.display.GUIState Constructor Methods

public GUIState()
Builds a GUIState. In this constructor, you must create an appropriate SimState, then call super(state); or this(state); as appropriate.

public GUIState(SimState state)
Builds a GUIState. You'll need to implement this, but in it you typically just call super(state);

The implementation of these contructors can be confusing, but the easy way to think of it is (1) you have to implement both of them and (2) *both* of them must call super(state) or this(state) (I always just call super(state)). Let's say that your model is MyModel. Your GUIState subclass (in MASON tradition called MyModelWithUI) would usually have constructors like this:

[1]SWARM and Repast users would call this *probing* object. Coming from a NeXTSTEP and MacOS X background, I was more comfortable with the term *inspection*.

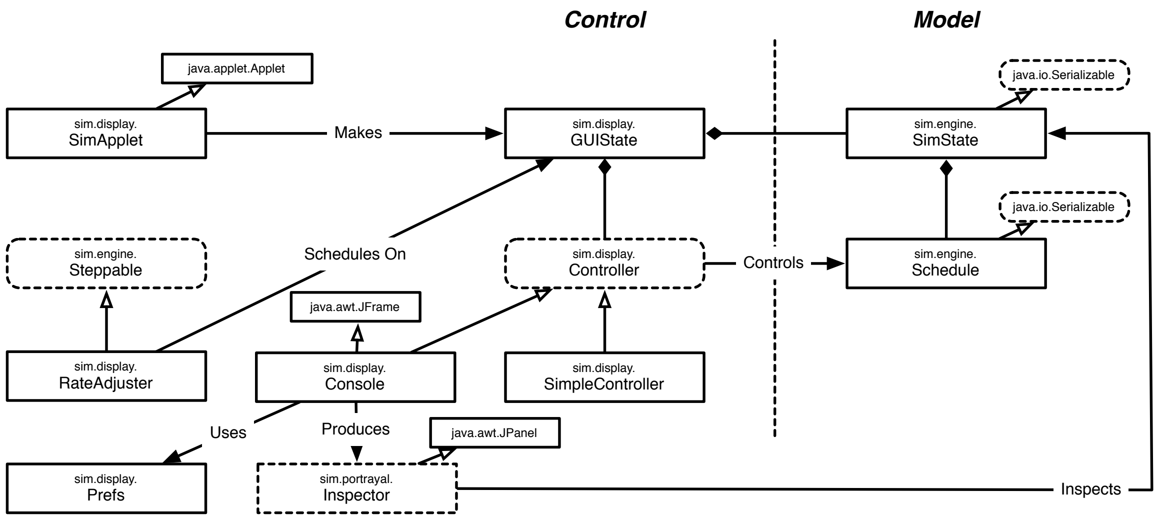


Figure 8.2 UML diagram of MASON's GUI control code.

```
public MyModelWithUI() { super(new MyModel(System.currentTimeMillis())); }
public MyModelWithUI(SimState state) { super(state); }
```

The first constructor creates a MyModel instance, passing in a random number seed pulled from the current time in milliseconds. You are free to add more stuff to these constructors of course, but you should follow these templates.

Next you need to override certain methods. Here are the most common ones:

sim.display.GUIState Methods

public void init(Controller controller)
: Initializes the GUIState with the given controller at the start of the GUI. If you override this, be sure to call super.init(controller) first. This method is called in the GUI thread well before the model has started, so there's no need to obtain a lock on the schedule.

public void quit()
: Shuts down the GUIState in preparation for quitting the GUI. If you override this, be sure to call super.quit() first. This method is called in the GUI thread well after the model has ended, so there's no need to obtain a lock on the schedule.

public void start()
: Starts the simulation. If you override this, be sure to call super.start() first, which, among other things, will call start() on the underlying model. This method does not have a lock on the schedule, so if you have AsynchronousSteppables in your model, you should obtain a lock on the schedule *after* calling super.start().

public void finish()
: Ends the simulation. If you override this, be sure to call super.finish() first, which, among other things, will call finish() on the underlying model. After super.finish(), the model is no longer running and there are no AsynchronousSteppables, so there is at present no need to obtain a lock on the schedule (you might as well though, for good measure).

public void load(SimState state)
: Continues a simulation which had been checkpointed. The model is provided, already been loaded from the checkpoint. If you override this, be sure to call super.load(state) first. This method does not have a lock on the schedule, so if you have AsynchronousSteppables in your model, you should obtain a lock on the schedule *after* calling super.load().

You will almost certainly override init(...), start(), quit(), and load(...). Here are the common tasks you'll almost always do in these methods:

- init(...) Construct at least one Display, create a JFrame for it, register the JFrame with the Controller, make the JFrame visible, and attach one or more Field Portrayals to the JFrame.
- start() and load(...) These methods typically have identical code, so much so that in most of the MASON application demos, they just call a common made-up method called setupPortrayals() which does that code. Typically the tasks are: attach to each Field Portrayal the Field it is portraying, then in each Field Portrayal construct and attach the Simple Portrayals for the various objects in the Field, or its ColorMap. Last, reset the Displays and request them to be repainted.
- quit() Dispose the Displays' JFrames and set them and the Displays to null.

You might also override finish(). The most common task here is to stop charts etc. which might potentially have movies being run on them. (Standard MASON 2D and 3D Displays stop themselves, don't worry about them). This task is also commonly done in quit() as well.

More on Displays and Portrayals will be covered in Chapters 9 and 11.

8.1.1 Running a GUI-based Simulation

The easiest way to start a GUI-based Simulation is:

1. Run MASON as `java sim.display.Console`
2. Enter the full class name of your GUIState subclass in the "Simulation class name:" text field which pops up.
3. Press "Select"

That'll get tired fast. There are two other ways to do it.

First Way You'll note that the pop-up window that appeared has a list of MASON models you can just click on. Where did MASON get this list of models? Answer: from the sim/display/simulation.classes file. This is just a text file of GUIState subclasses. If you add yours to it, it'll appear in the list with the others. See Section 8.2.2.2 for more information about adding to this file.

Second Way If you create a GUIState subclass, it should include a main(...) method which fires it up as follows. Let's say your GUIState is called MyModelWithUI. You'd include:

```
public static void main(String[] args)
    {
    new MyModelWithUI().createController();
    }
```

The createController() method builds an appropriate Controller and makes it visible, which is sufficient to start up the GUI simulation:

sim.display.GUIState Methods

public Controller createController()
    Builds a controller appropriate for this GUIState, then sets it visible (starting the GUI). Returns the controller. The default builds a sim.display.Console, which should work in most cases, but you may wish to override it to build something else, such as a sim.display.SimpleController.

This way, if you start MASON like this:

```
java MyModelWithUI
```

... then your GUI simulation will start immediately. All the MASON tutorials and demos have been set up this way.

8.1.2 Properly Scheduling non-Model (GUI) Events

This is an advanced topic, so if you want you can skip it for now.

When the user starts the Controller, start() is called, and then the Controller goes into a loop, each time calling step() on the GUIState. This in turn calls step() on the underlying Schedule of the model. Displays, Inspectors, and a variety of other UI elements need to be updated every timestep as well — so they can repaint themselves for example — **but you must not schedule them on the Schedule**. They are not serializable, and violate model/visualization separation. So where do you "schedule" them to be called?

The answer: the **GUIState's Mini-Schedule**. This isn't *really* a Schedule per se, but a collection of methods sufficient for most GUI simulation updating tasks. The GUIState lets you schedule Steppables to be stepped at the start of the model, at the end of the model, or immediately before or immediately after a step of the Schedule. The most commonly used method is scheduleRepeatingImmediatelyAfter(...), which schedules a Steppable to be stepped immediately after every iteration of the model, to repaint a Display to show the latest results, for example.

Displays and Inspectors schedule themselves on the Mini-Schedule automatically; there's no reason for you to do it. But you might want to do something periodically after each step of the model: for example, you might wish to update a special Java window you have created. Feel free to use the Mini-Schedule, though remember that such scheduled items are *not* stepped if the model is running independently of the visualization. **Remember to never schedule GUI or non-serializable items on the Schedule.** Doing so is a major source of bugs among MASON users. Only schedule model things on the model's schedule.

Here are the methods in question:

sim.display.GUIState Methods

public boolean step()
    Steps any pre-model Steppables, then steps the model forward one step, then steps any post-model Steppables.

public boolean scheduleImmediatelyBefore(Steppable event)
    Schedules an event to occur once immediately before the next model step. The event will be called in the model thread, not the GUI thread, and you will already have a lock on the Schedule. If you need to modify a GUI feature, you should use SwingUtilities.invokeLater(...) as described next. Returns false if the item could not be scheduled (perhaps the simulation is already over, for example).

public boolean scheduleImmediatelyAfter(Steppable event)
    Schedules an event to occur once immediately after the next model step. The event will be called in the model thread, not the GUI thread, and you will already have a lock on the Schedule. If you need to modify a GUI feature, you should use SwingUtilities.invokeLater(...) as described next. Returns false if the item could not be scheduled (perhaps the simulation is already over, for example).

public Stoppable scheduleRepeatingImmediatelyBefore(Steppable event)
    Schedules an event to occur immediately before each future model step. The event will be called in the model thread, not the GUI thread, and you will already have a lock on the Schedule. If you need to modify a GUI feature, you should use SwingUtilities.invokeLater(...) as described next.Returns a Stoppable to stop further steps of the Steppable, or null if the item could not be scheduled (perhaps the simulation is already over, for example).

public Stoppable scheduleRepeatingImmediatelyAfter(Steppable event)
    Schedules an event to occur immediately after each future model step. The event will be called in the model thread, not the GUI thread, and you will already have a lock on the Schedule. If you need to modify a GUI feature, you should use SwingUtilities.invokeLater(...) as described next.Returns a Stoppable to stop further steps of the Steppable, or null if the item could not be scheduled (perhaps the simulation is already over, for example).

public boolean scheduleAtStart(Steppable event)
    Schedules an event to occur immediately *after* the model's start() method is called but *before* the GUIState subclass's start() code is executed. The event will be called in the GUI thread, not in the model thread: but the model thread will not be running yet, and you will already have a lock on the Schedule. Returns false if the item could not be scheduled (perhaps the simulation is already over, for example).

public boolean scheduleAtEnd(Steppable event)
    Schedules an event to occur immediately *before* the model's finish() method is called and *before* the GUIState subclass's finish() code is executed. The event will be called in the GUI thread, not in the model thread. The model thread will no longer be running but AsynchronousSteppables may still be running. However, you will already have a lock on the Schedule. Returns false if the item could not be scheduled (perhaps the simulation is already over, for example).

Okay, so we can schedule an event. But this event runs in the model thread, not the Swing Event thread. How do we guarantee the GUI is updated in a threadsafe way? Before using any of these methods, **be sure to read the next section carefully.**

8.1.3 Calling GUI Elements From Schedule/Mini-Schedule Thread (and Vice Versa)

This is an advanced topic, so if you want you can skip it for now.

When the simulation runs, it does so in its own separate thread independent of the GUI. It's often the case that you need to communicate between the two, typically by calling one thread from the other. **You need to be careful** in doing this so as not to incur a race condition.

Four common situations are:

- **Modifying a GUI widget each time the model iterates** In this scenario, you have scheduled an event on the GUIState mini-schedule to change (for example) the value of text field. The mini-schedule runs in the model thread, but the GUI runs in the Swing Event thread.
- **Asking a GUI widget to repaint each time the model iterates** In this scenario, you have scheduled an event on the GUIState mini-schedule send a repaint() request to a widget. repaint() is threadsafe, but the paintComponent method which is ultimately called in response still needs to be careful about race conditions.
- **Updating the model synchronously in response to the user changing a GUI widget** Perhaps you need to update the model in response to the user modifying a widget, and prevent the user from doing further modifications until this has happened.
- **Updating the model asynchronously in response to the user changing a GUI widget** This is much less common than the synchronous situation. Suppose that every time the user presses a button you need to send a message to the model, but the model doesn't have to respond to it immediately, just before the next time tick. Several such requests can pile up.

Let's start with the last two first.

Updating the Model from the GUI Asynchronously If you don't need to make the change immediately, you could avoid pausing the simulation, by inserting a Steppable in the GUIState's Mini-Schedule (which runs either in the model thread or when the model is paused), like so:

```
GUIState guistate = ...
guistate.scheduleImmediatelyAfter(
   new Steppable()
       {
       public void step(SimState state)
           {
           // put your model-modifying code here
           }
      });
```

Note that this modifies the simulation asynchronously. So if the user presses the button twice, two things will be scheduled on the mini-schedule.

Updating the Model from the GUI Synchronously You can grab a lock on the Schedule. If the simulation presently holds the lock, you'll block until given it. Then the simulation thread will wait until you're done. This is the recommended approach.

```
GUIState guistate = ...
synchronized(guistate.state.schedule)
      {
      // put your model-modifying code here
      }
```

Modifying the GUI from the GUIState mini-schedule (in the model thread) This is fairly common. Let's say that you want to schedule an event which for some reason sets the color of a javax.swing.JLabel to a random color every iteration of the Schedule — using the model's random number generator. In your GUIState's start() method, you might **think** you could do the following, but **part of it is wrong**:

```
final JLabel label = ...
GUIState guistate = ...
guistate.scheduleRepeatingImmediatelyAfter(new Steppable()
    {
    public void step(SimState state)
        {
        Color c = new Color(state.random.nextInt(256), state.random.nextInt(256), state.random.nextInt(256));
        label.setForeground(c);    // THIS IS WRONG
        }
    });
```

This is wrong because the Steppable is being called in the model thread. Most of Swing is not threadsafe: modifications of Swing must be done from within the Swing Event Thread (the GUI's thread). The correct way to do this is:

```
final JLabel label = ...
scheduleRepeatingImmediatelyAfter(new Steppable()
    {
    public void step(final SimState state)
        {
        SwingUtilities.invokeLater(new Runnable()
            {
            public void run()
                {
                synchronized(state.schedule)  // now we can modify or query the model
                    {
                    // inside the synchronized section, use the model as you need and get out quickly
                    int r = state.random.nextInt(256);
                    int g = state.random.nextInt(256);
                    int b = state.random.nextInt(256);
                    }
                // now do the GUI stuff using this data
               label.setForeground(new Color(r, g, b));
                }
            });
        }
    });
```

The SwingUtilities.invokeLater(...) method takes a Runnable and queues it up to be called at some point in the future from within the Swing Event Thread (where it'd be threadsafe), along with other events (repaint requests, mouse and key events, etc.). Ordinarily you don't have any control over when this Runnable will get called. But MASON can make a guarantee for you: it'll definitely be called, in the GUI, before the next iteration of the Schedule.

However, since the Runnable doesn't get called from within the model thread, you need to obtain a lock on the Schedule to make certain that the model thread isn't running when you make your modifications. It's important that you stay in the lock as little as possible — just access or manipulate the model inside the lock, and then get out. Otherwise you'll slow down the simulation.

So in short: we schedule a Steppable to be executed in the model thread each time immediately after the schedule is stepped. This Steppable tells the panel to repaint itself in the future (it'll be before the next Schedule step). The Steppable then posts on Swing's Event loop a Runnable to be called in the future (again, before the next Schedule step). When the Runnable is run, it synchronizes on the schedule to be certain it has complete control, then fools around with the model as necessary.

Repainting a widget from the GUIState mini-schedule (in the model thread) It's often the case that all you need to do is ask a widget to repaint itself. This is fairly easy because repaint(), unlike other Swing methods, is threadsafe:

```
final JPanel panel = ...
GUIState guistate = ...
guistate.scheduleRepeatingImmediatelyAfter(new Steppable()
    {
    public void step(SimState state)
        {
        panel.repaint();
        }
    });
```

repaint() just puts an event request on the Swing event schedule which will eventually call paintComponent(). (It'll be done before the next model iteration). However paintComponent() is not threadsafe: if you need to access the model inside paintComponent, you *must* lock on the Schedule first:

```
protected void paintComponent(Graphics g)
    {
    synchronized(state.schedule)
        {
        /* gather information from the model */
        }
    /* do repaint code using that information */
    }
```

8.1.4 Handling Checkpoints

This is an advanced topic, so if you want you can skip it for now.

When a Controller wants to checkpoint out a model, it merely needs to call the writeToCheckpoint(...) method on the SimState model (see Section 4.2.1). But when a Controller needs to *load* a model from a checkpoint, it needs to also update the displays and inspectors etc. to reflect this load.

Here's the procedure. The Controller calls the GUIState's method readNewStateFromCheckpoint(...). This method loads a complete model from the checkpoint, then hands it to the GUIState's validSimState(...) method. If that method returns true, the old model is finished and the new model is passed to load(...) to set it up.

You have two methods you can override to control this process. First, you may (of course should) override the load(...) method to inform Displays and other GUI widgets of the new model, as discussed earlier. Second, you can override validSimState(...) to double-check that the SimState is correct. The default version says it's correct if it's not null and if its class is the same as the current model's class. You might want to be more detailed than that (though it's rarely needed).

The methods in question:

sim.display.GUIState Methods

public boolean validSimState(SimState state)
Returns true if the provided SimState (likely loaded from a checkpoint file) can be used with this visualization. The default implementation, which compares the SimState's class with the existing SimState's class, is usually sufficient.

public boolean readNewStateFromCheckpoint(File file)
Loads and builds a SimState from a given File, calling validSimState(...) on it to verify it, then finish() to end the existing model and load(...) to load the new model. Returns false if the SimState could not be loaded or was not considered valid.

8.2 Controllers

A **controller** an object whose job is to perform the equivalent of the MASON top-level loop when running under a UI. (Discussion of the "big loop" or "top-level loop" was in Section 4.2). In addition to handling the top-level loop, Controllers also handle 2D and 3D Displays (Sections 9 and 11) and also Inspectors (Section 10):

- You can **register JFrames** (holding 2D or 3D Displays) with the Controller to be closed, hidden, and updated as necessary.
- Inspectors (Section 10) are managed updated by the Controller.
- You can ask the Controller to update (refresh) all displays and inspectors.
- You can ask the Controller to pause the entire simulation in order to run a certain chunk of GUI code in a threadsafe manner. This is rarely done.

MASON provides two controllers built-in for you:

- sim.display.Console provides a GUI window to start, stop, etc. the simulation, as well as lots of other gizmos (checkpointing, creating new simulations with other controllers, etc.). This is the most common Controller by far.
- sim.display.SimpleController has no GUI at all — it's entirely programmatic. It's useful for controlling a MASON simulation which has no UI widgets except for a display (for example, a video game). SimpleController is much simpler in capability than Console.

A Controller defines a half-dozen methods to do these various tasks. Rather than discuss them here, most will be discussed in later sections (Displays, Inspectors), where they're more relevant.

8.2.1 The Controller's Big Loop

Recall that when running the model without a UI, the top-level loop is typically something like this:

```
 1. Create an instance of a SimState subclass called state.
 2. Call state.nameThread(...); to label the thread with a name you'll recognize in debuggers.
 3. Loop some jobs times:
 4.     Call state.setJob(...);
 5.     Call state.start();
 6.     Loop:
 7.         Call boolean result = state.schedule.step(state);
 8.         If result == false or if too much time has passed, break from Loop
 9.     Call state.finish();
10. Call System.exit(0); for good measure (to kill any wayward threads you may have accidentally created).
```

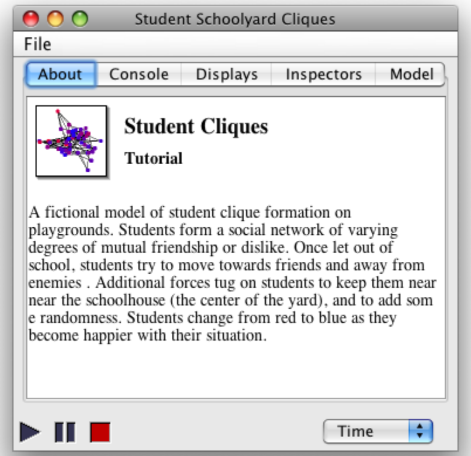
Also recall that MASON provided a convenience method, doLoop(), which handled this for you if you liked. In the GUI, the loop is slightly different but not by much. It more or less looks like this:

```
 1. Create an instance of a SimState subclass called state.
 2. Create a GUIState called gui, passing in state.
 3. Build a controller and set it up. This is usually done with gui.createcontroller();
 4. The controller calls gui.init()
 5. The controller Loops, doing the following:
 6.     When the user presses the play or pause buttons:
 7.         Keep the job at 0.
 8.         Call gui.start(), which in turn will call state.start();
 9.         Loop:
10.             If the simulation is not paused (the pause button isn't pressed):
11.                 Step GUI Steppables meant to be run before a Schedule step (Section 8.1.3).
12.                 Call boolean result = state.schedule.step(state);
13.                 Step GUI Steppables meant to be run after a Schedule step (Section 8.1.3).
14.                 If result == false or if the user has pressed stop, break from the inner Loop
15.         Call state.finish();
16.     When the user quits the GUI, break from the outer Loop.
17. The controller calls gui.finish()
```

This is all a lie actually, but a useful lie to give you the idea of how things basically work. The controller doesn't really do a loop: it fires of an underlying thread which does a fairly complex dance with the GUI thread, plus listening to events coming in to pause and play and stop, etc. But it gives you the general idea: one call to init(); followed by a loop which repeatly calls start(), then steps the Schedule N times, then calls finish(); and finally one call to quit().

8.2.2 The Console

The **Console** is a Controller defined by the class sim.display.Console. It's the most common Controller and is often viewed as the Grand Central Station of a typical simulation. Because it is sort of the central GUI object, the Console handles a lot of simulation tasks in MASON.



A fictional model of student clique formation on playgrounds. Students form a social network of varying degrees of mutual friendship or dislike. Once let out of school, students try to move towards friends and away from enemies . Additional forces tug on students to keep them near near the schoolhouse (the center of the yard), and to add som e randomness. Students change from red to blue as they become happier with their situation.

The Console looks like the window you see on the right. When you create a Console and set it going, it will call init() on your GUIState to set up the rest of the GUI. You can then press play, pause, or stop on the Console to control the simulation.

The Console's constructor is listed below, though most commonly you'd create one by simply calling createController() on your GUIState. During construction of the Console, it calls init(...) on the GUIState. The GUIState can respond by changing the location of the Console, like this:

```
console.setLocation(200, 200);
```

In computing a desired location, one useful piece of information is the width of the screen:

```
GraphicsEnvironment.getLocalGraphicsEnvironment().getDefaultScreenDevice().
                                                  getDefaultConfiguration().getBounds();
```

But in fact usually you needn't bother. The default location for the Console will be directly to the right of the primary Display in your GUI, if there's enough room onscreen. Usually that looks great.

By the way, the default width and height of the Console, and its *gutter* (the distance between it and the primary Display to its left), are defined as constants in the Console class:

```
public final static int DEFAULT_WIDTH;
public final static int DEFAULT_HEIGHT;
public final static int DEFAULT_GUTTER;
```

Here's how you make a Console:

sim.display.Console Constructor Methods

public Console(GUIState simulation)
  Constructs a Console based on the given GUIState. Calls init(...) on the GUIState. If the init(...) method does not position the Console, it will by default be set immediately to the right of the primary Display in the model, if there is enough room.

Notice that the Console is presently displaying an **HTML page** and has a **name in the title bar**. Here's how it got those. Your GUIState subclass may define *static* methods called getName() and getInfo(), which return a name for the simulation and some HTML to display, respectively. getName() returns a String, and getInfo() returns a String (plain text or HTML) or a URL pointing to an HTML file.

If you do not specify these methods, GUIState will use default values: the name for the simulation

Why are these methods static?

Because in some cases MASON needs to display this information before ever instantiating a simulation: for example, when displaying the "New Simulation..." dialog.

How can you "override" a static method? That's absurd.

Ah, the magic of reflection. MASON looks up to see if your GUIState subclass has defined the static method, then calls it using the reflection facility.

will be the shortened class name of the GUIState subclass; and the HTML will be drawn from a file named index.html (if there is one) located right next to the .class file of your GUIState subclass. It's usually smart to override the getName() method; but the getInfo() method is rarely overridden — instead, just create the index.html file, put it in the right place, and you're good.

sim.display.GUIState Methods

public static Object getInfo()
: Even though this method is static, a GUIState sublass may implement it. Returns an object to display in the HTML window of the Console. The Object returned can be a plain String, an HTML String, or a URL pointing to an HTML file.

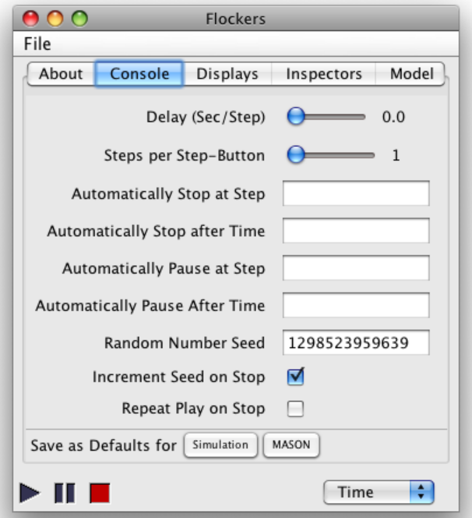
public static Object getInfo(Class theClass)
: Calls the static method getInfo() on *theClass*, if it's defined, and returns its value. If there is no such method, then looks for an HTML file called index.html located next to the .class file of *theClass*, and returns a URL to it if there is one, else a blank HTML page.

public static String getName()
: Even though this method is static, a GUIState sublass may implement it. Returns a String defining the desired name of the simulation.

public static String getName(Class theClass)
: Calls the static method getName() on *theClass*, if it's defined, and returns its value. If there is no such method, calls getTruncatedName(theClass) and returns that.

public static String getTruncatedName(Class theClass)
: Returns a shortened name for *theClass*.

If you click on the **console tab** of the Console, you'll get the items at right. It's a variety of options for controlling the Schedule and the Random Number Generator:



Delay (Sec/Step) 0.0
Steps per Step-Button 1
Automatically Stop at Step
Automatically Stop after Time
Automatically Pause at Step
Automatically Pause After Time
Random Number Seed 1298523959639
Increment Seed on Stop ✓
Repeat Play on Stop

- **Delay** How many seconds must transpire between each iteration of the schedule. This basically gives you a mechanism for slowing the simulation down.
- **Steps per Step-Button** When the Pause button is depressed, the Play button turns into a Step Button, allowing you to step through the simulation. This slider specifies how many steps occur each time you press the Step Button.
- **Automatically Stop/Pause at Step/Time** You can tell MASON to stop or pause the simulation at any simulation time after some number of simulation steps (iterations) have occurred.
- **Random Number Seed** This specifies the random number seed for the next simulation. Note that MersenneTwister only uses the low 32 bits of the seed.
- **Increment Seed on Stop** After the simulation has stopped, should the seed be incremented in anticipation of another simulation iteration?
- **Repeat Play on Stop** After the simulation has stopped, should MASON automatically start playing a new simulation iteration?

- **Defaults** This allows you to save appropriate values as your preferences for MASON as a whole, or for this particular application. Application defaults override MASON defaults. The facility which handles Preferences is discussed a bit later, in Section 8.3.

All of these items can be set programmatically as well:

sim.display.Console Methods

public long getWhenShouldEnd()
Returns when the simulation should end automatically, in number of steps (or Long.MAX_VALUE if not set).

public void setWhenShouldEnd(long val)
Sets when the simulation should end automatically, in number of steps (or Long.MAX_VALUE if it shouldn't).

public long getWhenShouldPause()
Returns when the simulation should pause automatically, in number of steps (or Long.MAX_VALUE if not set).

public void setWhenShouldPause(long val)
Sets when the simulation should pause automatically, in number of steps (or Long.MAX_VALUE if it shouldn't).

public double getWhenShouldEndTime()
Returns when the simulation should end automatically, in number of steps (or Schedule.AFTER_SIMULATION if not set).

public void setWhenShouldEndTime(double val)
Sets when the simulation should end automatically, in number of steps (or Schedule.AFTER_SIMULATION if it shouldn't).

public double getWhenShouldPauseTime()
Returns when the simulation should pause automatically, in number of steps (or Schedule.AFTER_SIMULATION if not set).

public void setWhenShouldPauseTime(double val)
Sets when the simulation should pause automatically, in number of steps (or Schedule.AFTER_SIMULATION if it shouldn't).

public boolean getShouldRepeat()
Returns true if the Console repeats play on stop.

public void setShouldRepeat(boolean val)
Sets whether the Console repeats play on stop.

public boolean getIncrementSeedOnStop()
Returns true if the Console increments the seed when stopped in preparation for the next play.

public void setIncrementSeedOnStop(boolean val)
Sets whether the Console increments the seed when stopped in preparation for the next play.

Recall that if you call any of these methods from the model thread (say from the GUIState mini-schedule), you need to obey certain rules to avoid race-conditions. See Section 8.1.3.

If you would like to add a tab to the tabs in the Console, you can do that too:

sim.display.Console Methods

public JTabbedPane getTabPane()
Returns the tab pane for the Console.

8.2.2.1 Setting up and Shutting Down the GUI

MASON can support more than one simulation running at a time: each has its own Console and displays. New simulations can be created from the **New Simulation...** menu in an existing Console, which pops up the Simulation-Chooser Window that lets the user enter a simulation class to start (see Figure 8.3). This window can also be called forth with a method call as described below. A simulation is automatically created simply by creating the Console (or other controller) and setting it visible if appropriate (see Section 8.1.1). Ultimately this is what the dialog box does, or it can be called from your main(...) method as described in Section 8.1.1.

Because MASON can have multiple simulations running at a time, there are really two "quit" procedures in the Console: either quit a given simulation, or quit *all* simulations. The user can quit a given simulation by closing its Console. All simulations are quit by selecting the **Quit** menu. Both of these have method calls as well.

sim.display.Console Methods

public void doClose()
: Closes the console and shuts down the simulation, calling quit(). This method can be called manually, and will also be called via doQuit().

public static void doQuit()
: Closes all consoles and shuts down their simulations, calling quit() on them. This method can be called manually.

public void doNew()
: Fires up the Simulation-Chooser box to allow the user to pick a simulation to create.

public boolean isNewMenuAllowed()
: Returns true if the "New Simulation..." menu is permitted.

public void setNewMenuAllowed(boolean val)
: Stipulates whether or not the "New Simulation..." menu is permitted.

I didn't disable it, yet the New Simulation... menu is disabled.

The menu will also be disabled if your simulation.classes file has no entries for some reason. See Section 8.2.2.2, coming up next, for more information on this file.

The above methods allow you to restrict the user's access of the **New Simulation...** menu. Why would you want to do this? Primarily to keep the user from entering and starting up a different simulation. By default the user is permitted.

You typically start up the MASON GUI in two ways. First, you can create your own main(...) method and then run Java on that class, as discussed ad nauseum elsewhere. Or, if you like, you can just run Java on the Console itself, like this:

```
java sim.display.Console
```

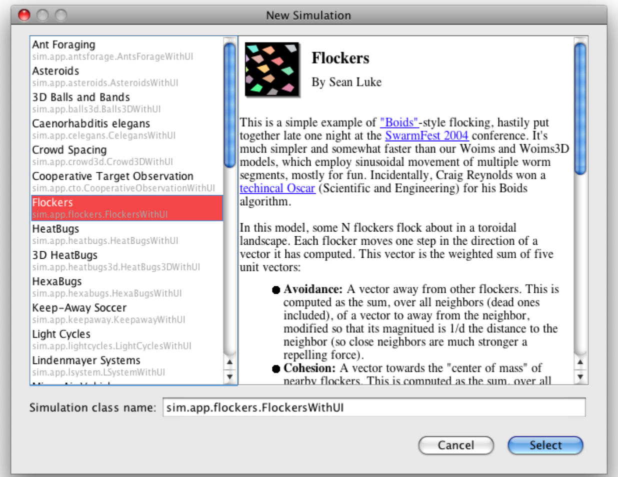
Because a few MASON simulations require considerable amounts of memory (particularly ones using Java3D) I recommend specifying a large maximum heap size. Here's how you stipulate 500 megabytes for example:

```
java -Xmx500M sim.display.Console
```

This will pop up the Simulation-Chooser Window directly. Thus the Console defines the following method (of course):

sim.display.Console Methods

public static void main(String[] args)
: Fires up the MASON GUI and creates the SImulation Chooser Window.



Ant Foraging
sim.app.antsforage.AntsForageWithUI
Asteroids
sim.app.asteroids.AsteroidsWithUI
3D Balls and Bands
sim.app.balls3d.Balls3DWithUI
Caenorhabditis elegans
sim.app.celegans.CelegansWithUI
Crowd Spacing
sim.app.crowd3d.Crowd3DWithUI
Cooperative Target Observation
sim.app.cto.CooperativeObservationWithUI
Flockers
sim.app.flockers.FlockersWithUI
HeatBugs
sim.app.heatbugs.HeatBugsWithUI
3D HeatBugs
sim.app.heatbugs3d.HeatBugs3DWithUI
HexaBugs
sim.app.hexabugs.HexaBugsWithUI
Keep-Away Soccer
sim.app.keepaway.KeepawayWithUI
Light Cycles
sim.app.lightcycles.LightCyclesWithUI
Lindenmayer Systems
sim.app.lsystem.LSystemWithUI

Flockers

By Sean Luke

This is a simple example of "Boids"-style flocking, hastily put together late one night at the SwarmFest 2004 conference. It's much simpler and somewhat faster than our Woims and Woims3D models, which employ sinusoidal movement of multiple worm segments, mostly for fun. Incidentally, Craig Reynolds won a techincal Oscar (Scientific and Engineering) for his Boids algorithm.

In this model, some N flockers flock about in a toroidal landscape. Each flocker moves one step in the direction of a vector it has computed. This vector is the weighted sum of five unit vectors:

- **Avoidance:** A vector away from other flockers. This is computed as the sum, over all neighbors (dead ones included), of a vector to away from the neighbor, modified so that its magnitued is 1/d the distance to the neighbor (so close neighbors are much stronger a repelling force).
- **Cohesion:** A vector towards the "center of mass" of nearby flockers. This is computed as the sum, over all

Simulation class name: sim.app.flockers.FlockersWithUI

Figure 8.3 The Simulation-Chooser Window.

8.2.2.2 The Simulation Chooser

Where does MASON get the list of classes to display in the Simulation Chooser Window? Answer: a special text file called sim/display/simulation.classes

This file usually little more than a list of full class names, like this:

```
sim.app.antsforage.AntsForageWithUI
sim.app.asteroids.AsteroidsWithUI
sim.app.balls3d.Balls3DWithUI
sim.app.celegans.CelegansWithUI
sim.app.crowd3d.Crowd3DWithUI
sim.app.cto.CooperativeObservationWithUI
sim.app.flockers.FlockersWithUI
sim.app.heatbugs.HeatBugsWithUI
```

... and so on. You can also have blank lines and comments (which start with a #), like this:

```
sim.app.antsforage.AntsForageWithUI
sim.app.asteroids.AsteroidsWithUI
# This is a comment  And the line immediately after it is a blank line.  Both are ignored.

sim.app.balls3d.Balls3DWithUI
sim.app.celegans.CelegansWithUI
sim.app.crowd3d.Crowd3DWithUI
```

The classes listed in this file are the ones shown in the window. But there are two additional gizmos you should be aware of. First off, note that the user is free to type in his own class name if he likes. But if there is a line all by itself which says ONLY then MASON will not allow this (the text file will simply be missing). You do it like this:

```
sim.app.antsforage.AntsForageWithUI
sim.app.asteroids.AsteroidsWithUI
# Below is the 'ONLY' line.  You can put it anywhere you like as long as it's on a line by itself.

ONLY
sim.app.balls3d.Balls3DWithUI
sim.app.celegans.CelegansWithUI
sim.app.crowd3d.Crowd3DWithUI
```

Second, notice that the Simulation Chooser Window displays names for simulations and (if you click on them) their HTML info, that is, the values which were returned by the GUIState.getName() and GUIState.getInfo() methods (see Section 8.2.2). The getInfo() method is only called lazily if you click on that simulation. The getName() method is called to get the names of each simulation to place them in the Simulation Chooser Window's list.

However, what if a certain GUIState subclass is for some reason very expensive to load? You can bypass the call to getName() by specifying a name in the sim/display/simulation.classes file, like this:

```
sim.app.antsforage.AntsForageWithUI
sim.app.asteroids.AsteroidsWithUI
# This Name: declaration must appear *immediately* before the class in question

Name: 3D Balls and Bands
sim.app.balls3d.Balls3DWithUI
sim.app.celegans.CelegansWithUI
sim.app.crowd3d.Crowd3DWithUI
```

That way if you fire up MASON via sim.display.Console, you can display certain classes in the list without having to load their class files. This is a very rare need.

8.2.2.3 Running the Simulation

The Console can **play** the simulation, **pause** it, **unpause** it, **step** it some N iterations while presently paused, and **stop** the simulation (end it). Naturally there are some obvious constraints: for example, if the simulation is presently playing, you can't play it again; and if the simulation is over, it can't be stopped again, etc. Normally these actions are performed by the user pressing various start/stop/pause/etc. buttons on the Console, but they can also be performed via method calls. The Console also keeps track of the Schedule's current state, one of:

```
public int PS_STOPPED = 0;
public int PS_PLAYING = 1;
public int PS_PAUSED = 2;
```

You can also set the Console to require dialog-box confirmation in order to stop a simulation.

Relevant methods:

sim.display.Console Methods

public void pressPlay()
: If the simulation is *stopped*, reinitializes the simulation to and starts it, calling start() on the GUIState, and then firing off the model thread. The model thread then repeatedly iterates the model. If the simulation is *paused*, steps the simulation N times, then reverts to being paused. The value of N can be set with setNumStepsPerStepButtonPress(...).

public void pressStop()
: Stops the simulation and calls finish() on the GUIState.

public void pressPause()
: If the simulation is presently *playing*, pauses the simulation, suspending the model thread. If the simulation is presently *paused*, unpauses the simulation and resumes the model thread. If the simulation is presently *stopped*, starts the simulation (as if pressing pressPlay()), then immediately pauses it.

public void setNumStepsPerStepButtonPress(int val)
: Sets the number of iterations the simulation will perform when the user presses the play button while the simulation is paused. You can set this to any value you like > 0, but in the console GUI the user can only set it to between 1 and MAXIMUM_STEPS, presently 20. Note that this method is so named because the play button transforms into a "step" button when the simulation is paused.

public int getNumStepsPerStepButtonPress()
: Returns the number of iterations the simulation will perform when the user presses the play button while the simulation is paused. that this method is so named because the play button transforms into a "step" button when the simulation is paused.

public int getPlayState()
: Returns the current state of the Schedule.

public void setRequiresConfirmationToStop(boolean val)
: Sets the Console to require (or not require) dialog box confirmation in order to stop the simulation.

public boolean getRequiresConfirmationToStop()
: Returns whether or not the Console requires dialog box confirmation in order to stop the simulation.

public double getStepsPerSecond()
: Returns the current frame rate of the simulation in steps per second.

8.2.2.4 Loading and Saving

The Console can save running simulations out to a checkpoint file, and also load current checkpoint files. This is done with menu options (Load, Save As, etc.). You can also do this programmatically:

sim.display.Console Methods

public void doSaveAs()
: Pops up the save-as panel to enable the user to save the simulation to a checkpoint file.

public void doSave()
: Saves the simulation to a checkpoint file, if one was already stipulated via the save-as panel; otherwise pops up the save-as panel.

public void doOpen()
: Pops up the Open panel to enable the user to load a simulation from a checkpoint file. If the simulation is loaded, it is then paused.

8.2.3 The Simple Controller

Like the Console, the sim.display.SimpleController class is a Controller, and it shares a number of facilities in common with Console. However, as befits its name, SimpleController is *much* simpler.

First off, SimpleController is not a JFrame, or even a graphical widget. It is a simple class which implements the Controller interface. Like the Console, it registers displays and inspectors, but it gives the user no access to them. You cannot save or open simulations or create new ones.

Why would you want such a class? Primarily when you don't want a Console displayed — for example if you're creating a game — and would prefer a more lightweight approach.

SimpleControllers are generally made the same way as Consoles, though there's an extra constructor which dictates whether or not the SimpleController will display inspectors registered with it — else it'll simply ignore all registered inspectors and the user will never see them.

sim.display.SimpleController Constructor Methods

public SimpleController(GUIState simulation)
: Creates a SimpleController to control the given simulation.

public SimpleController(GUIState simulation, boolean displayInspectors)
: Creates a SimpleController to control the given simulation, displaying inspectors registered with it.

8.2.3.1 Running the Simulation

Like Console, SimpleController can **play** the simulation, **pause** it, **unpause** it, **step** it some *N* iterations while presently paused, and **stop** the simulation (end it). If the simulation is presently playing, you can't play it again; and if the simulation is over, it can't be stopped again, etc. Unlike in Console, these actions cannot be made by the user: only your code can perform them. SimpleController also can increment the seed automatically if you stop and start the simulation.

Just like Console, SimpleController keeps track of the Schedule's current state, one of:

```
public int PS_STOPPED = 0;
public int PS_PLAYING = 1;
public int PS_PAUSED = 2;
```

Relevant methods:

sim.display.SimpleController Methods

public void pressPlay()
: If the simulation is *stopped*, reinitializes the simulation to and starts it, calling start() on the GUIState, and then firing off the model thread. The model thread then repeatedly iterates the model. If the simulation is *paused*, steps the simulation *N* times, then reverts to being paused. The value of *N* can be set with setNumStepsPerStepButtonPress(...).

public void pressStop()
: Stops the simulation and calls finish() on the GUIState.

public void pressPause()
: If the simulation is presently *playing*, pauses the simulation, suspending the model thread. If the simulation is presently *paused*, unpauses the simulation and resumes the model thread. If the simulation is presently *stopped*, starts the simulation (as if pressing pressPlay()), then immediately pauses it.

public int getPlayState()
: Returns the current state of the Schedule.

public boolean getIncrementSeedOnStop()
: Returns true if the Console increments the seed when stopped in preparation for the next play.

public void setIncrementSeedOnStop(boolean val)
: Sets whether the Console increments the seed when stopped in preparation for the next play.

8.2.3.2 Setting up and Shutting Down the GUI

Just like the Console, you "quit" your simulation by calling doClose(), which in effect "closes" the SimpleController. Unlike the Console, there's no doQuit() method to quit all of MASON, but you can just call Console's version (which is static). You can also pop up the Simulation Chooser Window with doNew() (See Section 8.2.2.2).

sim.display.SimpleController Methods

public void doClose()
: "Closes" the SimpleController and shuts down the simulation, calling quit(). This method can be called manually, and will also be called via Console.doQuit().

public void doNew()
: Fires up the Simulation-Chooser box to allow the user to pick a simulation to create.

8.3 Preferences

The Console can save various settings to permanent user preferences: certain buttons on the GUI allow you to make settings the preference defaults for MASON as a whole, or for the simulation in question (simulation preferences override MASON default preferences).

How does the Console do this? Java has a Preferences facility, but it's a bit complex. MASON's sim.display package has a simplifying class which makes this facility somewhat easier to use: the sim.display.Prefs class.

The Prefs class has two kinds of preferences: MASON-wide preferences and application-specific (that is, simulation-specific) preferences. For the Java Preferences buffs among you, these are stored with the Preference key paths:

```
public static final String MASON_PREFERENCES = "edu/gmu/mason/global/";
public static final String APP_PREFERENCES = "edu/gmu/mason/app/";
```

If you're creating a library object or widget which requires preferences and will which will be used across many different simulations, you'll probably want to enable the user to save both to MASON (global) preferences and to simulation-specific (app) preferences. If on the other hand you're creating an object or widget which will only be used for a specific simulation, there's no reason to bother with the global preferences. Just create an app-specific preference.

Within the global and app preferences, a given object or widget must carve out its own **namespace**, typically a short string tacked onto the preferences string. For example, the namespace for the preferences for sim.display.Display2D is presently *Display2D*, which is sufficient to distinguish it from other objects at present. A single word *Foo* is probably fine if you pick a unique string. But if you want to be more cautious, you could use something like a full classname, along the lines of *sim/display/Foo*. You should not use periods — use slashes instead.

Once you've picked out a Namespace, Prefs will make it easy for you to get the java.util.prefs.Preferences object corresponding to your object or widget's preferences. You'll need to read up on how to add and remove preferences from the Preferences object.

sim.display.Prefs Methods

public static java.util.prefs.Preferences getGlobalPreferences(String namespace)
: Returns the java.util.prefs.Preferences object corresponding to the name space *namespace* within MASON's global preferences. If your namespace is *Foo*, then the Java Preference key path will be edu/gmu/mason/global/Foo.

public static java.util.prefs.Preferences getAppPreferences(GUIState simulation, String namespace)
: Returns the java.util.prefs.Preferences object corresponding to the name space *namespace* within the application preferences of the given simulation. If your namespace is *Foo*, and the simulation's GUIState is *sim.app.mysim.MySimWithUI*, then the Java Preference key path will be edu/gmu/mason/app/sim/app/mysim/MySimWithUI/Foo.

public static boolean save(Preferences prefs)
: Saves to disk the given Preferences. Returns false if an exception occurred when deleting the object: for example, when your file system does not support preferences. This is most often the case if you're running as an applet. Else returns true.

public static boolean removeGlobalPreferences(String namespace)
: Deletes the java.util.prefs.Preferences object corresponding to the name space *namespace* within MASON's global preferences. Returns false if an exception occurred when deleting the object: for example, when your file system does not support preferences. This is most often the case if you're running as an applet. Else returns true.

public static boolean removeAppPreferences(GUIState simulation, String namespace)
: Deletes the java.util.prefs.Preferences object corresponding to the name space *namespace* within the application preferences of the given simulation. Returns false if an exception occurred when deleting the object: for example, when your file system does not support preferences. This is most often the case if you're running as an applet. Else returns true.

MASON'S Widgets MASON's Console and MASON's sim.display.Display2D and sim.display3d.Display3D classes use the Prefs object to save preferences. These widgets use MASON-level and simualtion-level preferences in the following way:

1. If a user has specified a simulation-level preference for a given value, that preference is used.
2. Else if the user has specified a MASON-level preference for the value, it is used.
3. Else the default setting for that value is used.

Console's namespace, stored in sim.display.Console.DEFAULT_PREFERENCES_KEY, is at present "Console". Display2D's namespace, stored in sim.display.Display2D.DEFAULT_PREFERENCES_KEY, is "Display2D". Display3D's namespace, stored in sim.display3d.Display3D.DEFAULT_PREFERENCES_KEY, is "Display3D".

8.4 Producing a Consistent Framerate

In certain rare cases you may wish to guarantee that your simulation runs no faster than a given frame rate. Almost invariably this would be used if you're using MASON to develop a game: and occasionally you might need it to make certain movies. The Console has a facility for specifying the amount of sleep time between iterations, but if each iteration takes a variable amount of time, this can't be used to guarantee a certain speed.

MASON has a simple class you can add to your simulation which will do the trick. The class, sim.display.RateAdjuster, attempts to guarantee that your simulation will maintain a frame rate fixed to no more than a given number of iterations per second. The class is a sim.engine.Steppable. All that matters to you is the constructor:

sim.display.RateAdjuster Constructor Methods

public RateAdjuster(double targetRate)
Produces a RateAdjuster designed to achieve the given target rate (in iterations per second).

During your GUIState's start() and load() methods, you'd schedule the RateAdjuster on your GUIState's minischedule, like so:

```
// I want 60 iterations per second
myGUIState.scheduleRepeatingImmediatelyAfter(new RateAdjuster(60.0));
```

It's not guaranteed: in some pathological situations you may get slightly faster than the target rate. But it usually does a good job.

8.5 Making an Applet

MASON simulations can be converted into Java applets without much difficulty. You more or less just need to make a jar file and add an appropriate HTML page.

Some gotchas however: applets cannot read or write to disk without special permissions set up, so you'll need to eliminate reading and writing from

But I need to load files for my simulation!

Then you shouldn't be loading files using the java.io.File class. Instead, open a stream to a file using java.lang.Class.getResourceAsStream(...). Here's how you use that method. Let's say you have a file called *Foo* which is stored right next to the *MySim.class* file holding the compiled version of a class called MySim which you use in your simulation. You can then say:

```
InputStream s = MySim.class.getResourceAsStream("Foo");
```

This approach will work inside a JAR file in an applet; or in a JAR file version of your simulation running as an application, or in pretty much any other context. We recommend this approach for loading every file or resource you need.

your simulation. Also note that preferences won't be savable. Last, applets often have memory restrictions, and MASON simulations can be memory hungry. For example, certain 3D visualization may not work properly.

Here are the steps you'll need to take:

Step 0. Tweak the Java compiler target if necessary
[This is optional] Not all operating systems run Java 1.6. For example, most web browsers on OS X still run Java 1.4.2. If you compile your code for too high a version of Java, it won't run on those web browsers which don't support that. If you're using MASON's makefile, this is easy. The compiler target and source code versions are defined in the line:

```
JAVACFLAGS = -target 1.4 -source 1.4 ${FLAGS}
```

The setting here is 1.4, which is probably already fine in most circumstances. If it doesn't say that, change it to 1.4 temporarily. Rebuild MASON using the compiler target you've chosen:

```
make clean
make 3d
```

If you're using Eclipse or NetBeans, you'll need to change the compiler target and java source version on those systems. They're probably defaulting to 1.6.

Step 1. Modify the simulation.classes file You'll probably want to restrict this file to just those simulations you want the user to have access to. See See Section 8.2.2.2 for more information about this file and its options.

Step 2. Make the MASON directory smaller [This is optional] You may wish to back up the MASON directory, then remove from the directory any files you don't need. For example, if you're not doing 3D, you can remove the sim.portrayal3d and sim.display3d directories. You'll probably want to remove all the application directories in the sim.app package that you're not going to use as well. Your goal is to make the JAR file smaller.

Step 3. Build the JAR file You can build a big and ugly JAR file by backing out of the MASON directory, then calling:

```
java cvf mason.jar mason
```

... but it's probably better to let MASON build a smarter JAR file with just the kinds of files it needs (class files, JPG and PNG files, certain HTML files, the simulation.classes file, etc.). If you're in UNIX or on a Mac, go into the mason directory and type:

```
make jar
```

Step 4. Deploy the Applet MASON has HTML which will deploy your mason.jar file as an applet. It's located at sim/display/SimApplet.html. Just stick the mason.jar file right next to that HTML file on your server and you're set.

Chapter 9

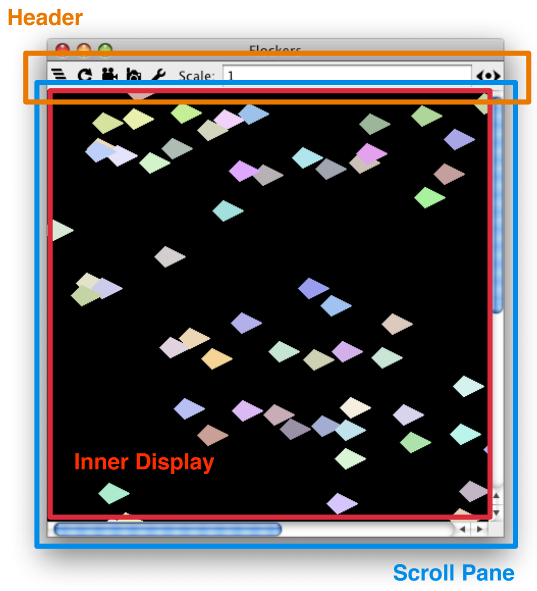
Visualization in 2D

MASON provides a wide range of 2D visualization options for your underlying model. Visualization in MASON is divided into three parts:

- The **2D Display** object (sim.display.Display2D) is the Swing graphical interface widget responsible for drawing and selecting model objects.
- **2D Field Portrayals** (subclasses of sim.portrayal.FieldPortrayal2D) are registered with the Display2D and are responsible for doing drawing and selection on a given field. Each acts as the interface between the Display and the underlying field.
- **2D Simple Portrayals** (subclasses of sim.portrayal.SimplePortrayal2D) are registered with each Field Portrayal and are responsible for performing actual drawing or selection of a given object stored in the field. Simple Portrayals may be registered for various kinds of objects in a field: and in fact objects in the field may act as their own Simple Portrayals. In some cases (so-called **"fast field portrayals"** the Field Portrayal eschews calling forward Simple Portrayals to do its dirty work, and instead just draws the objects directly using simple visualization (colored rectangles, for example).

The general structure of MASON's 2D visualization facility is shown in Figure 9.1.

9.1 The 2D Display



Your simulation can have as many Display2Ds as you like. Each Display2D is registered with the Console and appears in its "displays" tab. If you close the Display2D window, it's not gone permanently — just double-click on it in the "displays" tab and it'll reappear. However, closing a Display2D has the benefit of not requiring it to repaint itself, which will allow the simulation to run much faster.

The Display2D is the central hub to visualizing the majority of MASON simulations. It can do a lot of tasks, including:

- Displaying your model (of course).
- Choosing which Field Portrayals to display (they're layered on top of one another).

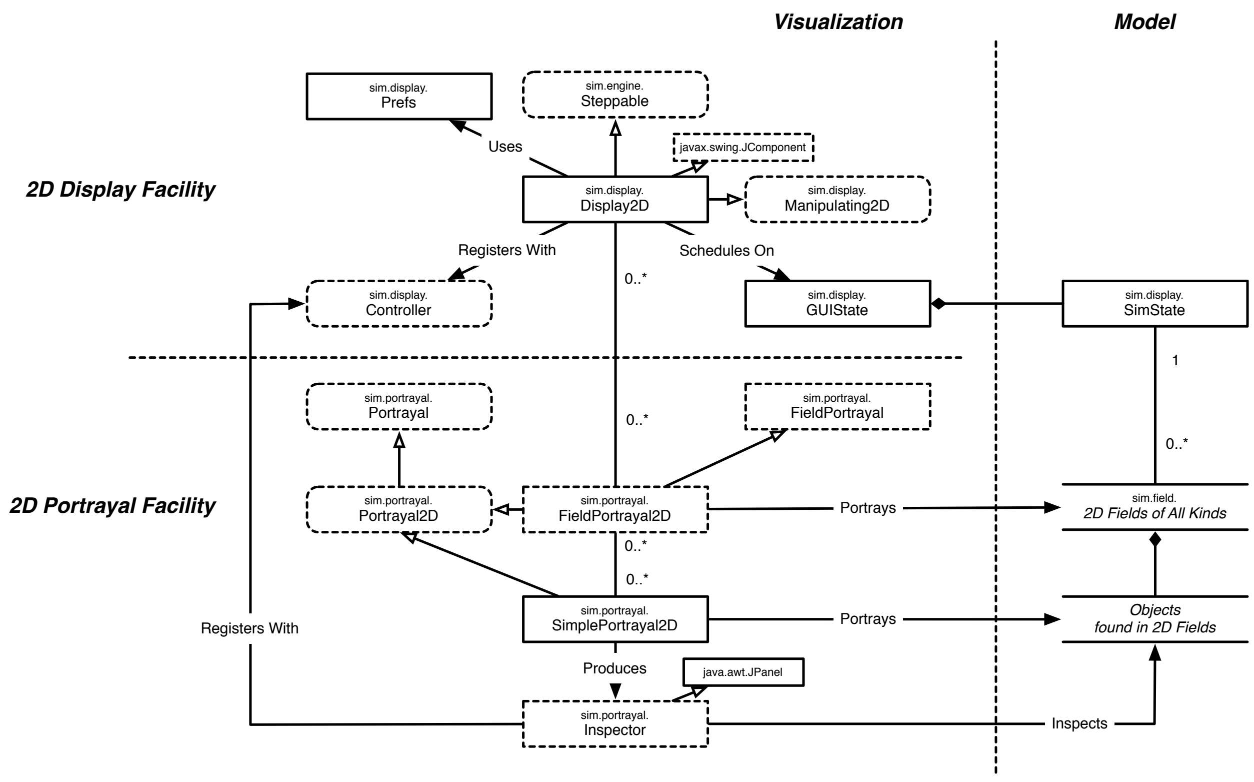


Figure 9.1 UML diagram of MASON's Top-Level 2D Visualization Code. Notice the three sections. The Visualization section is divided into 2D Display and 2D Portrayal facilities.

- Zooming in and out, resizing the window, and scrolling the display region.
- Generating bitmaps, true vector PDFs, and movies of the simulation.
- *Inspecting* elements in the model: just double-click on an element and an inspector will pop up in the "inspectors" tab of the Console to allow you to examine and modify it.
- *Selecting* elements in the model: if you single-click on an element, the object will be selected. Selected objects can have various operations performed on them: for example, having them display a special circle or a label or a trail. Certain selected objects can also be dragged about or have their size or orientation changed.
- Adding antialiasing, per-displayed-object tooltips, etc.
- Handling user-customized mouse event routing.

The sim.display.Display2D class is how you make 2-dimensional display widgets and windows for your simulation. It's a very big class, with a lot of parts. Here are the main parts:

- The **Inner Display** is the region which handles the actual drawing and selection.
- A **Scroll Pane** holds the Inner Display and allows you to scroll around in it (the Inner Display is often larger than the Display2D proper).
- A **Header** holds the button bar and widgets at the top of the Display2D.

- An **Option Pane**, called forth by a widget in the Header, presents additional options to the user.

These are defined by the following variables, which are rarely accessed by yourself:

```
public InnerDisplay2D insideDisplay;
public OptionPane optionPane;
public Box header;
public JScrollPane display;
```

The header has various other widgets, which exist as variables in the Display2D:

```
// The button () and popup menu for toggling Field Portrayals
public JToggleButton layersbutton;
public JPopupMenu popup;
public static final ImageIcon LAYERS_ICON;
public static final ImageIcon LAYERS_ICON_P; // (pressed)

// The button () and popup menu for specifying how often the Display should be redrawn
public JToggleButton refreshbutton;
public JPopupMenu refreshPopup;
public static final ImageIcon REFRESH_ICON;
public static final ImageIcon REFRESH_ICON_P; // (pressed)

// The frame, combo box, and field for additional Display redraw options.
public JFrame skipFrame;
public JComboBox skipBox;
public NumberTextField skipField;

// The button () for starting or stopping a movie
public JButton movieButton;
public static final ImageIcon MOVIE_OFF_ICON;
public static final ImageIcon MOVIE_OFF_ICON_P;
public static final ImageIcon MOVIE_ON_ICON; // (movie is running)
public static final ImageIcon MOVIE_ON_ICON_P; // (pressed, movie is running)

// The button () for taking a snapshot
public JButton snapshotButton;
public static final ImageIcon CAMERA_ICON;
public static final ImageIcon CAMERA_ICON_P; // (pressed)

// The button () for calling forth the Options pane.
public JButton optionButton;
public static final ImageIcon OPTIONS_ICON;
public static final ImageIcon OPTIONS_ICON_P; // (pressed)
// The field (Scale: 1 ) for scaling/zooming.
public NumberTextField scaleField;
```

A Display2D is constructed as follows:

sim.display.Display2D Constructor Methods

public Display2D(double width, double height, GUIState guistate)

Returns a Display2D connected to the given GUIState. The viewable region of the Display2D's Inner Display is set to the given width and height in pixels.

When constructing a Display2D, it automatically connects to the GUIState and adds itself to the GUIState's minischedule to be stepped every simulation iteration so it can update itself if it feels so inclined.

Once you set up a Display2D, your next task is usually to arrange its display parameters and put it in a JFrame. Display2D is capable of sprouting its own JFrame. If you choose to use your own JFrame instead, be sure to call Display2D.quit() when the frame is disposed.

Why isn't this just a standard Java background?

Because the Display2D's setBackground(...) method would set the background color of the Display2D, not of the model being displayed in the Inner Display.

Display2D can also add a **backdrop** color or other paint: furthermore, you can set this value to null, which instructs MASON not to draw a background at all: the actual backdrop will be undefined. Why would you want to do this? Because if your fields are being drawn opaquely (such as a grid of values), there's no reason to draw a backdrop and it's faster not to do so. The Display2D can also change the scale of the display, and can **clip** the display to the bounds of the field: for fields with infinite space (such as sim.field.grid.SparseGrid2D, you'll probably want to turn the clipping off — it's on by default.

sim.display.Display2D Methods

public void setScale(double val)
Sets the scale (zoom factor), which must be a positive number ≥ 0.

public double getScale()
Returns the scale (zoom factor).

public void setClipping(boolean val)
Sets the display to clip to the bounds of its fields.

public boolean isClipping()
Returns whether or not the display is clipping to the bounds of its fields (the default is true).

public void setBackdrop(Paint c)
Sets the backdrop paint of the Display. Set to null to instruct MASON not to draw the backdrop at all, which is faster if your fields are opaque and the backdrop wouldn't be seen anyway.

public Paint getBackdrop()
Returns the backdrop paint of the Display.

public JFrame createFrame()
Directs the Display2D to spout a JFrame and put itself in it. The JFrame has not been set visible yet.

public Frame getFrame()
Returns the Display2D's current JFrame (whether it created it or not).

Once you've created a JFrame, you'll want to register it with your sim.display.Console or other sim.display.Controller. Your Controller will include the Display in its list of objects to refresh or update in certain situations; and will also include it in its "displays" tab to you can hide or show the Display at your leisure. Here are the relevant Controller methods:

sim.display.Controller Methods

public boolean registerFrame(JFrame frame)
Registers the JFrame, notionally holding a Display of some sort, with the Controller, and returns true, unless the Controller cannot register JFrames of any kind, in which case this method returns false.

public boolean unregisterFrame(JFrame frame)
Unregisters the JFrame, notionally holding a Display of some sort, with the Controller, and returns true, unless the Controller cannot register or unregister JFrames of any kind, in which case this method returns false.

public boolean unregisterAllFrames()
: Unregisters all JFrames registered with the Controller, and returns true, unless the Controller cannot register or unregister JFrames of any kind, in which case this method returns false.

public void refresh()
: Schedules updates and redraws of all Inspectors and Displays registered with the Controller to occur at some time soon in the future. This is an expensive procedure and should not be called unless necessary: typically in response to some event (a button press etc.) rather than changes in the model itself. Only call this method from the Swing even thread.

Next you'll want to attach Field Portrayals and certain global Inspectors to the Display. When the Display is updated, it'll instruct its Field Portrayals to redraw themselves. A Field Portrayal can in theory be attached to multiple Displays but it rarely makes sense to do so. Usually you'll just attach the Field Portrayal, though sometimes you may wish to translate it and scale it relative to other Field Portrayals so it's lined up properly with them in certain circumstances.

sim.display.Display2D Methods

public void attach(final sim.portrayal.Inspector inspector, final String name)
: Attaches the given Inspector to the Display, assigning it the provided name. The user can call forth this Inspector by choosing that name from the Display's "layers" menu.

public void attach(FieldPortrayal2D portrayal, String name)
: Attaches a FieldPortrayal2D to this Display, assigning it the given name, setting it initially visible, and placing it at the $\langle 0, 0\rangle$ position in the InnerDisplay (in pixels). The width and height of the FieldPortrayal2D is set to that of the Inner Display. The user can toggle the FieldPortrayal2D's visibility by choosing its name in the Display's "layers" menu.

public void attach(FieldPortrayal2D portrayal, String name, Rectangle2D.Double bounds)
: Attaches a FieldPortrayal2D to this Display, assigning it the given name, setting it initially visible, and placing it at the given bounds in the InnerDisplay (in pixels). This allows both translation and prescaling of the Field-Portrayal2D relative to others. The user can toggle the FieldPortrayal2D's visibility by choosing its name in the Display's "layers" menu.

public void attach(FieldPortrayal2D portrayal, String name, boolean visible)
: Attaches a FieldPortrayal2D to this Display, assigning it the given name, setting it initially visible or not, and placing it at the $\langle 0, 0\rangle$ position in the InnerDisplay (in pixels). The width and height of the FieldPortrayal2D is set to that of the Inner Display. The user can toggle the FieldPortrayal2D's visibility by choosing its name in the Display's "layers" menu.

public void attach(FieldPortrayal2D portrayal, String name, double x, double y, boolean visible)
: Attaches a FieldPortrayal2D to this Display, assigning it the given name, setting it initially visible or not, and placing it at the given $\langle x, y\rangle$ position in the InnerDisplay (in pixels). The width and height of the FieldPortrayal2D is set to that of the Inner Display. This merely translates the FieldPortrayal2D relative to others. The user can toggle the FieldPortrayal2D's visibility by choosing its name in the Display's "layers" menu.

public void attach(FieldPortrayal2D portrayal, String name, Rectangle2D.Double bounds, boolean visible)
: Attaches a FieldPortrayal2D to this Display, assigning it the given name, setting it initially visible or not, and placing it at the given bounds in the InnerDisplay (in pixels). This allows both translation and prescaling of the FieldPortrayal2D relative to others. The user can toggle the FieldPortrayal2D's visibility by choosing its name in the Display's "layers" menu.

public ArrayList detatchAll()
: Detatches all FieldPortrayal2Ds from the Display.

When a simulation is begun, you'll want to reset the Display. This causes it to clear all its selected objects and reschedule itself on the GUIState's minischedule. When you quit the entire simulation, you'll want to quit the Display as well so it can free resources and finish any movies.

sim.display.Display2D Methods

public void reset()
: Causes the Display to clear all of its current selected objects and reschedule itself in the GUIState's minischedule.

public void quit()
: Quits the display, stopping it and finishing any movies and freeing resources.

The Display2D is sim.engine.Steppable. When the simulation is running, every iteration the Display2D is stepped: it first calls shouldUpdate() to determine if it should update and redraw itself. If the answer is yes, then it redraws itself, including writing out to any movie.

Speaking of movies, the Display2D is capable of generating movies and taking screenshots (both as PNG bitmaps and publication-quality PDF vector images). The kind of image (PNG or PDF) is specified by the following image type, defined in Display2D:

```
public final static int TYPE_PDF;
public final static int TYPE_PNG;
```

sim.display.Display2D Methods

public void step(final SimState state)
: Called every model iteration to pulse the Display, ultimately causing it to (if appropriate) update and repaint itself, and write out movies.

public void takeSnapshot(File file, int type)
: Takes a snapshot of the given type and saves it to the given file. Throws an IOException if the file could not be written.

public void takeSnapshot()
: Asks the user what kind of snapshot to take, and what file to save to, and then takes the snapshot, saving it to the file.

public void startMovie()
: Starts a movie, asking the user what kind of movie to make and what file to save it to. Only one movie can be generated at a time.

public void stopMovie()
: Stops any currently started movie.

public boolean shouldUpdate()
: Returns true or false to indicate if a Display2D should update and redraw itself at a given time. By default this method queries the Display2D's GUI regarding the user choices for updating. You may override this method if you feel inclined.

For no particularly good reason, the Display2D is the source of certain variables MASON sets, then uses to determine how to draw properly (different versions of Java, on different platforms, have different optimal settings). They're accessed like this:

```
public static final boolean isMacOSX; public static final boolean isWindows; public
static final String javaVersion;
```

(Obviously, if you're not OS X, and you're not Windows, you must be Linux!)

As discussed in Section 8.3, the Display2D maintains certain MASON and simulation preferences, using the key:

```
public String DEFAULT_PREFERENCES_KEY = "Display2D";
```

This key is used to store preferences information associated with the Display2D (such as preferred antialiasing, etc.). However if your simulation has more than one Display2D, you may need to make their preference keys distinct. I would associate the default preference key ("Display2D") with your "primary" Display2D, and use alternative keys for each of the others (for example, "Display2D-a" or some such). This can be done with the following methods:

sim.display.Display2D Methods

public void setPreferencesKey(String s)
  Sets the preferences key for the Display2D to the given string.

public String getPreferencesKey()
  Returns the current preferences key (the default is DEFAULT_PREFERENCES_KEY, set to "Display2D").

9.1.1 Drawing

Display2D doesn't actually draw anything. Instead, it holds a JScrollPane whose ViewPort holds an *Inner Display* (sim.display.Display2D.InnerDisplay2D which does the actual drawing. The drawing procedure works like this:

1. MASON asks Swing to update all displays.
2. Sometime soon thereafter, Swing asks the Display2D's Inner Display to repaint itself, calling paintComponent(...).
3. paintComponent(...) calls paintToMovie(...) if appropriate, then calls paint(...)
4. If paintToMovie(...) is called, it too calls paint(...)
5. paint(...) paints either to the window or to an image (which is saved to disk or added to a movie). In either case, it iterates through each of the attached Field Portrayals, telling each to paint itself by calling the Field Portrayal's draw(...) method. The paint(...) method also informs the Field Portrayal of the crop rectangle so it doesn't bother drawing beyond that.
6. Each Field Portrayal draws all the objects visible on-screen, either by drawing them itself, or calling forth Simple Portrayals to draw each object. In the latter case, the Field Portrayal calls draw(...) on the appropriate Simple Portrayal.
7. Some Simple Portrayals, known as **wrapper portrayals**, hold underlying Simple Portrayals. Wrapper portrayals typically first call draw(...) on their underlying portrayals, then add additional graphics. You can have a chain of any number of wrapper portrayals.

This section describes methods dealing with steps 2, 3, 4, and 5. Steps 5, 6 and 7 are discussed later in the sections on 2D Field and Simple Portrayals (that is, Sections 9.2 and 9.3 respectively).

9.1.1.1 The Inner Display

As described above, the Inner Display first has paintComponent(...) called, which in turn results in paintToMovie(...) and paint(...) being called. These are defined as:

sim.display.InnerDisplay2D Methods

public synchronized void paintComponent(Graphics g)
  Called by Swing to paint the InnerDisplay2D.

public void paintToMovie(Graphics g)
  Called by paintComponent(...) to add a frame to the current movie, when appropriate.

public BufferedImage paint(Graphics graphics, boolean buffered, boolean shared)
  Called by paintComponent(...) or paintMovie(...), or when snapshot is taken, to draw to graphics. The operation of this method differs depending on the arguments. If *buffered* is true, then the elements are first drawn to a BufferedImage, which is ultimately returned, else null is returned. If *shared* is false, then a shared BufferedImage is used (or reused) and returned, else a new BufferedImage is returned. Shared BufferedImages reduce memory allocation overhead, but if you need a BufferedImage of your own to permanently keep, request a non-shared image. If *graphics* is non-null, the BufferedImage is written to the Display, or if *buffered* was false, the elements are drawn directly to the Display.

The Inner Display has a *width* , a *height*, an *xOffset*, and a *yOffset*, all in pixels:

```
public double width;
public double height;
public double xOffset;
public double yOffset;
```

These define the bounding box, in pixels, for the Fields which are drawn in the Display. When a Field Portrayal is displayed, its drawing is scaled and translated so that the box from the field's origin $\langle 0, 0 \rangle$ to the Field Portrayal's own $\langle$*field.getWidth(),field.getHeight()*$\rangle$ corner match this bounding box.

The Inner Display is scalable: it has a factor retrievable from getScale(). Let's call this scale factor s. Thus we might roughly define the location and size of the field's bounding box as starting at $xOffset \times s, yOffset \times s$ and being of $width \times s, height \times s$ in dimension.

In addition to the offset, the precise location of the origin depends on where the user has scrolled using the JScrollPane. Furthermore, if you zoom out enough, the field is automatically centered in the window, which requires some tweaking of the origin as well.

The width and height of the Inner Display are set when you construct the Display2D: it would be quite rare to change them afterwards. The xOffset and yOffset are initially 0 each, but are changed by the Display2D's Options pane to shift its origin about in special circumstances (they're *not* used for scrolling).

These sizes influence the return values of the following two methods, which help the Inner Display work with its JScrollPane:

sim.display.InnerDisplay2D Methods

public Dimension getPreferredSize()
  Returns the width and height, each multiplied by the current scaling factor.

public Dimension getMinimumSize()
  Returns the width and height, each multiplied by the current scaling factor.

The Inner Display also has two RenderingHints which add features like antialiasing. The first is used when the Inner Display draws elements either to the screen or to an image. The second is used when drawing that image to the screen.

```
public RenderingHints unbufferedHints;
public RenderingHints bufferedHints;
```

Normally these are defined by user options in the Display's Options Pane. But you can hard-set your own rendering hints by overriding the following method:

sim.display.InnerDisplay2D Methods

public void setupHints(boolean antialias, boolean aphaInterpolation, boolean interpolation)
Sets up buffered and unbuffered hints according to the following requested values.

9.1.2 Selecting, Inspecting, and Manipulating Objects

One of the things a user can do is click on objects, drag them, etc. This is handled by the Inner Display as well, in conjunction with various Field Portrayals and Simple Portrayals.

The primary task here is **hit testing**: determining what objects in each Field fall within a region or intersect with a point (typically where the mouse clicked). MASON's 2D hit-testing structure is similar to how it does drawing:

1. A user clicks on the Inner Display.
2. Swing sends a mouse event to the Inner Display.
3. This event is routed through the method handleMouseEvent(...) (which gives you a chance to override it for your own custom purposes).
4. The default implementation of handleMouseEvent(...) routes raw mouse events to Field Portrayals by calling their handleMouseEvent(...) methods. It does this by first telling them to act on selected objects; failing this, it tells them to act on potentially hit objects.
5. The Field Portrayals may in turn call handleMouseEvent(...) on certain Simple Portrayals to move or rotate an object.
6. If handleMouseEvent(...) has not done anything with the event — the usual case — and it's a "mouse clicked" event, then Display2D either selects objects or constructs inspectors for them, depending on the number of mouse clicks.
7. In order to select objects or construct inspectors, Display2D must know what objects were hit by the mouse. It does this by calling objectsHitBy(...) to gather the objects in each Field hit by the point or region.
8. objectsHitBy(...) calls the method hitObjects(...) on each Field Portrayal.
9. Field Portrayals gather possible hit objects in their Fields. For further refinement, they may test each such object by calling hitObject(...) on their respective SimplePortrayals.

Again, we'll cover the Display2D-related steps. Other steps will be covered in Sections 9.2 and 9.3.

The first task is to handle the mouse event. Display2D has consolidated the mouseClicked(...), mouseExited(...), mouseEntered(...), mousePressed(...), mouseReleased(...), mouseDragged(...), and mouseMoved(...) MouseListener and MouseMotionListener methods into single method called handleMouseEvent(...). Display2D itself reacts to two of these events: mouseClicked(...) (of course) and mouseExited(...); and the default implementation of handleMouseEvent(...) calls equivalent methods in certain Simple Portrayals to give them an opportunity to do things such as move or rotate an object. You can override this method to handle other events (remember to call super(...). Alternatively you can eliminate all mouse listeners entirely so these methods are never called. This is done when you need more control in cases, for example, like if you're building a game.

Where's mouseWheelMoved(...)?

If you override that method, Java thinks that the JScrollPane should no longer be scrolled via a scroll wheel. So that one's out.

sim.display.Display2D Methods

public boolean handleMouseEvent(MouseEvent event)
Handles most mouse events that are presented to the Inner Display, except selection and inspection (single- and double-clicks). The default implementation calls handleMouseEvent(...) on each Field Portrayal regarding currently selected objects, then (if no Field Portrayal has reacted) calls handleMouseEvent(...) on each Field Portrayal regarding hit objects, then (if no Field Portrayal has still reacted, the usual case) returns false. If you override this method to add more event handling, be sure to call return super(...); if you have not handled the event yourself., else return true.

public void removeListeners()
Removes all listeners from the Inner Display, likely in preparation for adding your own. handleMouseEvent(...) will then never be called at all.

Let's presume that the mouse event resulted attempting to select or inspect objects. Next we need to do some hit testing to gather all the objects in all FIelds hit by the mouse or rectangular region. Display2D returns an array of Bags, one per Field, holding these object. Or more correctly, the Bags hold **Location Wrappers** (instances of sim.portrayal.LocationWrapper). A Location Wrapper, discussed later in Section 9.2.3, contains various information about the Object, including the Object itself, the Field Portrayal (and thus Field) it's located in, its location in the Field, and so on.

Display2D **selects** objects by first clearing all selections, then hit-testing for objects, then calling setSelected(...) on each Field Portrayal for each object.

Display2D **inspects** objects by first clearing all Inspectors from the Controller, then hit-testing for objects, then calling getInspector(...) on each Field Portrayal for each object, then submitting the resulting Inspectors to the Controller. The details of how Inspectors work and how they're constructed is discussed in a later Section (10).

Display2D's relevant methods are:

sim.display.Display2D Methods

public Bag[] objectsHitBy(Rectangle2D.Double rect)
Returns an array of Bags of LocationWrappers for every object which is hit by the given rectangular region. The size and order of the array is exactly that of the Field Portrayals registered with the Display2D, and each Bag represents the objects in one Field.

public Bag[] objectsHitBy(Point2D point)
Returns an array of Bags of LocationWrappers for every object which is hit by the given point. The size and order of the array is exactly that of the Field Portrayals registered with the Display2D, and each Bag represents the objects in one Field.

public void performSelection(LocationWrapper wrapper)
Selects the Object represented by the given LocationWrapper by calling the relevant FieldPortrayal's setSelected(...) method.

public void performSelection(Bag locationWrappers)
Selects the Object represented by the given LocationWrappers by calling each of their relevant FieldPortrayals' setSelected(...) methods.

public void performSelection(Rectangle2D.Double rect)
Selects all objects hit by the given rectangle, by calling each of their relevant FieldPortrayals' setSelected(...) methods.

public void clearSelections()
Instructs all Field Portrayals to clear all selections on all objects.

public void createInspectors(Rectangle2D.Double rect, GUIState simulation)
Generates and submits to the Console inspectors for each object hit by the given rectangle, by calling their relevant Field Portrayal's getInspector(...) method.

Display2D also uses hit testing to compute tool tip information. Tool tips are turned on by the user in the Options Pane. Various methods for tool tips, mostly for interaction with Swing, are in Inner Display. You'll probably never need to touch this:

sim.display.InnerDisplay2D Methods

public JToolTip createToolTip()
    Generates a tool tip.

public String getToolTipText(MouseEvent event)
    Creates the tool tip text for a given mouse event.

public String createToolTipText(Rectangle2D.Double rect, final GUIState simulation)
    Creates the tool tip text for a given rectangle and simulation. Used by getToolTipText(...).

public void updateToolTips()
    Revises the tool tip text information as the model changes.

9.2 2D Field Portrayals

For every field that's portrayed in a Display2D, there's a sim.portrayal.FieldPortrayal2D whose job is is to portray it. Display2D draws, selects, inspects, and otherwise manipulates objects and data in fields by asking Field Portrayals to do the job on its behalf. When drawing, Field Portrayals are layered one on top of each other in the Display2D: it asks the bottom-most Field Portrayal to draw itself first, then the next lowest Field Portrayal, and so on, up to the top.

The general structure of MASON's 2D visualization facility was shown shown in Figure 9.1, on page 148.

A Field Portrayal does five primary tasks:

- Draw its field.
- Perform hit testing for objects in its field.
- Select objects in its field.
- Provide Inspectors for objects in its field.
- Translate back and forth between the location of objects in the field and their location on-screen.

2D Field Portrayals are subclasses of sim.portrayal.FieldPortrayal2D. This is in turn a subclass of sim.portrayal.FieldPortrayal, which it shares with 3D Field Portrayals. 2D Field Portrayals also implement the interface sim.portrayal.Portrayal2D, which they share with all simple and field portrayals.

9.2.1 Portrayals and 2D Portrayals

A **portrayal** is how MASON completely separates model from visualization. Models do not draw themselves on-screen: rather, portrayals are assigned for fields in the models, and for objects in those fields, to draw on their behalf. Thus the same model can be portrayed in different ways: in 2D in various ways, in 3D in various ways, or not at all. The model objects don't need to know *anything* about the visualization system at all: there's a bright line separating the two.

This isn't to say that objects *can't* portray themselves: in fact occasionally objects in fields do act as their own portrayals. However MASON does not *require* them to portray themselves, and in a great many cases objects are assigned portrayals to act on their behalf.

All portrayals, both in 2D and 3D, and for both fields and the objects they contain, implement the interface sim.portrayal.Portrayal. This interface has the following methods:

sim.portrayal.Portrayal Methods

public Inspector getInspector(LocationWrapper wrapper, GUIState state)
Produces an Inspector for the object stored in the given wrapper. If the portrayal is a Field Portrayal, it will typically do this by calling forth a Simple Portrayal appropriate for the object and calling the same method on it.

public String getName(LocationWrapper wrapper)
Returns an appropriate name for the object in the given wrapper. If the portrayal is a Field Portrayal, it will typically do this by calling forth a Simple Portrayal appropriate for the object and calling the same method on it.

public String getStatus(LocationWrapper wrapper)
Returns an appropriate status (a short string description of the object's current state) for the object in the given wrapper. If the portrayal is a Field Portrayal, it will typically do this by calling forth a Simple Portrayal appropriate for the object and calling the same method on it.

public boolean setSelected(LocationWrapper wrapper, boolean selected)
Sets the object in the given wrapper to be selected or deselected. If the portrayal is a Field Portrayal, it might do this by storing this status and later, when the object is being drawn, calling the setSelected(...) method on an appropriate Simple Portrayal prior to having the Simple Portrayal draw the object. Or it might store the selected state in the Object somewhere or in the Simple Portrayal.

Notice what's missing: **drawing**. The sim.portrayal.Portrayal interface doesn't define any method for drawing objects or fields: this is entirely up to the particular mechanism used by the drawing facility (for example, AWT/Java2D versus Java3D). These and other details are handled by subclasses of Portrayal. In the 2D case, drawing is handled by the Interface sim.portrayal.Portrayal2D, which extends Portrayal. It adds the single method:

sim.portrayal.Portrayal2D Methods

public void draw(Object object, Graphics2D graphics, DrawInfo2D info)
Draws the given Object according to the arguments in the provided DrawInfo2D.

DrawInfo2D describes where to draw objects and how large, and also the clip rectangle and various other information. It's described next.

9.2.2 DrawInfo2D

When a Portrayal is told to draw itself, it's passed a sim.portrayal.DrawInfo2D object which details the where and how to do the drawing. This object contains several pieces of information:

- The **draw rectangle**: fields are supposed to scale themselves to fit within bounds of this rectangle. Individual objects are supposed to center themselves at the the origin of the rectangle, and be drawn with the assumption that the width and height of the rectangle (in pixels) represent one unit of width and height in the model. Typically objects should be drawn roughly one unit high and one unit wide. The Draw rectangle is also used for hit-testing to scale objects appropriately.

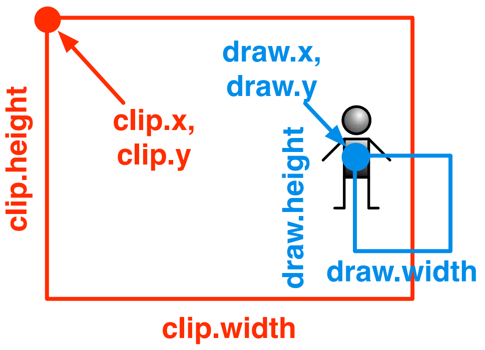


Figure 9.2 The Clip and Draw Rectangles, and their relationship to an object being drawn. Recall that in Java graphics, 0, 0 is the top-left corner of the environment and Y increases as you go *down*. The object is scaled to the width and height of the Draw rectangle and centered at its origin. If the object doesn't overlap the Clip rectangle, it need not be drawn at all.

- The **clip rectangle**: this rectangle specifies the clip region of the Display (the portion of the field which is visible to the user). If an object does not fall within the clip rectangle, there's no need to draw it at all: in fact, Field Portrayals may omit sending drawing requests to objects that they know fall outside the clip. The Clip rectangle is also used for hit-testing: it specifies the hit region. If an object overlaps this region, it is considered to be hit.
- Whether or not the object has been **selected** and should draw itself appropriately. This is a temporary setting.
- Whether or not the object should be drawn **precisely**, that is, using double-floating-point resolution Java2D primitives instead of (often faster) AWT integer primitives. This will be set when the object is being drawn to a high-quality PDF file for example.
- The **FieldPortrayal2D** which is being asked to draw objects.
- The **location** of the object in the FieldPortayal. This is set by the Field Portrayal before the Display2D is sent to the SimplePortrayal, and FieldPortrayals are free to set this or not: it's useful to some Field Portrayals but not others. Furthermore, the object may not necessarily be the actual location object (for example, it might be a sim.util.MutableDouble2D when the actual location is a sim.util.Double2D.

These six variables are publicly accessible:

```
public FieldPortrayal2D fieldPortrayal;
public Rectangle2D.Double draw;
public Rectangle2D.Double clip;
public boolean selected;
public boolean precise;
public Object location;
```

What's the point of DrawInfo2D? DrawInfo2D basically stores graphics context information (scaling, translating, clipping). You might be asking yourself: why do this when there's a perfectly good graphics context system in the form of Graphics2D's clip region and affine transforms? There are several reasons. First, DrawInfo2D is somewhat faster: because it doesn't allow rotation, we don't have to repeatedly apply, then remove, affine transform matrices. Second, and more importantly, Graphics2D's affine transform mechanism scales *everything*: font sizes, line thicknesses, etc., when one zooms in. Very often this is *not* what we want. By using DrawInfo2D we can choose to scale line thickness (for example) if we wish. With affine transforms we have no choice. Third, DrawInfo2D is *much simpler to understand*. It's probably not reasonable to ask simulation developers to perform rigorous affine transformation and resets on Java's graphics library. Fourth, if you want to use AWT instead of Java2D (it's often much faster), affine transforms aren't even an option.

Beyond the variables above, DrawInfo2D is mostly constructors:

sim.portrayal.DrawInfo2D Constructor Methods

public DrawInfo2D(GUIState gui, FieldPortrayal fieldPortrayal, RectangularShape draw, RectangularShape clip)
Builds a DrawInfo2D from the given draw and clip rectangles. precise and selected are both set to false and location is set to null.

public DrawInfo2D(DrawInfo2D other, double translateX, double translateY)
Builds a DrawInfo2D with its draw and clip rectangles translated by a certain factor from another DrawInfo2D. precise is copied from the other, but selected is set to false and location is set to null.

public DrawInfo2D(DrawInfo2D other)
Builds a DrawInfo2D with its draw and clip rectangles copied from another DrawInfo2D. precise is copied from the other, but selected is set to false and location is set to null.

Beyond this, there are only a few methods:

sim.portrayal.DrawInfo2D Methods

public boolean equals(Object obj)
: Compares against another DrawInfo2D's draw and clip rectangles and precise flag.

public String toString()
: Prints the DrawInfo2D to a string in a pleasing fashion.

The primary reason you might need to create a DrawInfo2D is for custom hit-testing on fields. If you need to create a DrawInfo2D object for this reason, don't make one from scratch: let Display2D do it for you:

sim.portrayal.Display2D Methods

public DrawInfo2D getDrawInfo2D(FieldPortrayal2D portrayal, Rectangle2D clip)
: Produces a DrawInfo2D suitable for hit-testing. The hit-test region is provided in clip. The Draw rectangle is set to the bounds of the entire Field Portrayal.

public DrawInfo2D getDrawInfo2D(FieldPortrayal2D portrayal, Point2D point)
: Produces a DrawInfo2D suitable for hit-testing. The hit-test region is a single point as provided. The Draw rectangle is set to the bounds of the entire Field Portrayal.

9.2.3 Location Wrappers

All four methods defined in sim.portrayal.Portrayal take **location wrappers** as arguments. A Location Wrapper (sim.portrayal.LocationWrapper) is a simple tuple which stores three things:

- An object in the model
- The location of the object in a field
- The field portrayal for the field

Where's the field?
You can get the field by querying the field portrayal.

LocationWrappers are used in all sorts of places where one needs to know not only an Object but where it is to be found. FieldPortrayals produce LocationWrappers for all sorts of things: and Inspectors are built using LocationWrappers. Note that these elements may change within a given LocationWrapper as the objects move about the field. The nature of this depends on the field in question. In some fields (such as sim.field.SparseGrid2D, objects move about, and so the location will change but the object will stay constant for that LocationWrapper. For other fields (such as sim.field.IntGrid2D, the location stays constant but the *value* of the location (returned as the "object") changes over time.

The constructor for a LocationWrapper is straightforward:

sim.portrayal.LocationWrapper Constructor Methods

LocationWrapper(Object object, Object location, FieldPortrayal fieldPortrayal)
: Produces a LocationWrapper with the given object, its location in a field, and the field portrayal for that field.

This LocationWrapper handes the default case. However this constructor is really pro-forma: almost all LocationWrappers are custom subclasses: rarely if ever is the default used in practice.

The methods are also quite simple:

sim.portrayal.LocationWrapper Methods

getObject()
: Returns the current object associated with this LocationWrapper. In LocationWrappers generated by certain Field Portrayals, this may change over time.

getLocation()
: Returns the current location of the object associated with this LocationWrapper. In LocationWrappers generated by certain Field Portrayals, this may change over time.

getLocationName()
: Returns a simple name for the location of the object returned by getLocation().

getFieldPortrayal()
: Returns the field portrayal associated with this LocationWrapper.

Again the default implementations of these methods are straightforward, but custom subclasses do many variations on them according to the needs of their particular Field Portrayal.

9.2.4 Field Portrayals

All 2D and 3D Field Portrayals are subclasses of sim.portrayal.FieldPortrayal. The primary job of a Field Portrayal is to draw and otherwise manipulate a specific field. Thus a Field Portayal acts as a kind of go-between for a 2D or 3D Display to work with an underlying field in your model.

To use a Field Portrayal, you need to do at least two things:

- Set the Field Portrayal's field
- Attach the Field to one or more Displays (usually just one).

Attaching the Field Portrayal to a Display was discussed earlier (page 151 of Section 9.1). To set the field, you simply call setField(...), as shown below.

You can also specify that the field is **immutable**, meaning that the FieldPortrayal should expect it never to change. This is useful for certain "fast" Field Portrayals to buffer up a display to draw faster without having to query the field over and over again.

Last, even if a FieldPortrayal is declared immutable, you can still force it to requery its field next time around, by setting the "dirty" flag with setDirtyField(...). This flag is later cleared. This is done sometimes by the FieldPortrayals themselves when appropriate (for example setField(...) sets the flag), but you can also use this to create a FieldPortrayal which *usually* doesn't update the flag except very rarely when occasional changes are made to the field. In this latter case, it's helpful that the "dirty" flag getter and setter methods are synchronized.

sim.portrayal.FieldPortrayal Methods

public void setField(Object field)
: Sets the FieldPortrayal's field. Also sets the "dirty" flag to true.

public Object getField()
: Returns the FieldPortrayal's field.

public void setImmutableField(boolean val)
: Makes the FieldPortrayal assume the field is immutable (or not).

public boolean isImmutableField()
: Returns whether the FieldPortrayal assumes the field is immutable.

public synchronized void setDirtyField(boolean val)
: Makes the FieldPortrayal dirty (or not).

public synchronized boolean isDirtyFlag()
Returns whether the FieldPortrayal is dirty.

Many Field Portrayals rely on **simple portrayals**, discussed in Section 9.3, to actually portray the individual objects in the field. These Field Portrayals take advantage of a facility in sim.portrayal.FieldPortrayal by which one can register Simple Portrayals and associate them with objects in the field: when told to (say) draw an object, the Field Portrayal looks up the appropriate registered Simple Portrayal, then asks it to do the task.

This process is highly flexible: you can register *any* Simple Portrayal you like to draw the object in any way appropriate. But it can also be slow, as it requires looking up the proper Simple Portrayal for every object being drawn. An alternative is to use a **"fast" Field Portrayal**, usually used for drawing grids of objects or values, which eschews Simple Portrayals entirely and just draws the grid as a bunch of rectangles of different colors. In this case, rather than provide Simple Portrayals, you provide a **Color Map** (sim.util.gui.ColorMap, described in Section 12) which maps values to colors.

You can register Simple Portrayals to be associated with individual objects, with Java Classes of objects, with all objects, and so on. Here's how a Field Portrayal figures out what Simple Portrayal to use for a given object:

1. If there is a **portrayalForAll**, use it.
2. Else if the object is null:
   (a) If there is a **portrayalForNull** registered, use it.
   (b) Else if a portrayal is **registered for null as an object**, use it (this is somewhat silly — use portrayalForNull).
   (c) Else use the FieldPortrayal's **default portrayal for null**.
3. Else (the object is non-null):
   (a) If the object is itself a Portrayal, use the object itself.
   (b) Else if there is a **portrayalForNonNull** registered, use it.
   (c) Else if a portrayal is **registered for the object**, use it.
   (d) Else if a portrayal is **registered for the object's Java Class**, use it.
   (e) Else if there is a **portrayalForRemainder**, use it.
   (f) Else use the FieldPortrayal's **default portrayal for non-null objects**.

The default portrayals for null and non-null objects are defined in abstract methods by subclasses of FieldPortrayal.

FieldPortrayal does this lookup with a method called getPortrayalForObject(...). This method, plus the various methods for registering Simple Portrayals, are:

sim.portrayal.FieldPortrayal Methods

public Portrayal getPortrayalForObject(Object obj)
Returns the Simple Portrayal registered for the given object. This runs through multiple checks to determine what Simple Portrayal to use: see the text above for an explanation as to how the method operates and what checks it uses.

public void setPortrayalForAll(Portrayal portrayal)
Sets the "portrayalForAll" of the Field Portrayal.

public Portrayal getPortrayalForAll()
Returns the "portrayalForAll" of the Field Portrayal, or null if not set.

public void setPortrayalForNull(Portrayal portrayal)
Sets the "portrayalForNull" of the Field Portrayal.

public Portrayal getPortrayalForNull()
Returns the "portrayalForNull" of the Field Portrayal, or null if not set.

public void setPortrayalForNonNull(Portrayal portrayal)
Sets the "portrayalForNonNull" of the Field Portrayal.

public Portrayal getPortrayalForNonNull()
Returns the "portrayalForNonNull" of the Field Portrayal, or null if not set.

public void setPortrayalForRemainder(Portrayal portrayal)
Sets the "portrayalForRemainder" of the Field Portrayal.

public Portrayal getPortrayalForRemainder()
Returns the "portrayalForRemainder" of the Field Portrayal, or null if not set.

public void setPortrayalForObject(Object obj, Portrayal portrayal)
Registers the portrayal for the given Object.

public void setPortrayalForClass(Class cls, Portrayal portrayal)
Registers the portrayal all Objects of a given class. The Object's class must be *exactly* this class: subclasses will not trigger this portrayal.

public Portrayal getDefaultNullPortrayal()
Returns the "default portrayal for null": by default this method simply calls getDefaultPortrayal().

public abstract Portrayal getDefaultPortrayal()
Returns the "default portrayal for non-null objects". Field Portrayals are required to implement this method to provide at *least* some fallback Simple Portrayal.

FieldPortrayal implements all the methods in Portrayal discussed above (getName(...), getStatus(...), getInspector(...), setSelected(...)), implemented by calling getPortrayalForObject(...) to extract the Simple Portrayal, then calling the equivalent-named method on the Simple Portrayal. "Fast" Field Portrayals override these methods to handle things themselves. Field Portrayals also have one additional convenience version of setSelected(...), which selects a whole Bag of objects at once:

sim.portrayal.FieldPortrayal Methods

setSelected(Bag locationWrappers, boolean selected)
Selects (or deselects) all object found in the LocationWrappers in the provided Bag.

.

9.2.5 2D Field Portrayals

2D Field Portrayals have a variety of standard methods for handling 2D issues. To begin, Field Portrayals implement the standard draw(...) method, except that the object passed in is ignored (in fact Display2D passes in the field itself, but don't rely on that). Field Portrayals also implement a method called hitObjects(...) which places into a Bag various LocationWrappers for all objects in the Field which were hit by the Clip rectangle of a given DrawInfo2D. This method is used by the Display2D to gather selected or inspected objects.

These two methods often share nearly identical code at the FieldPortrayal level. As a result the default implementation of these two methods call a single method called hitOrDraw(...): the hitObjects(...) version passes in null as the Graphics2D. Quite a lot of FieldPortrayals in MASON simply implement the hitOrDraw(...) method only.

sim.portrayal.FieldPortrayal2D Methods

public void draw(Object object, Graphics2D graphics, DrawInfo2D info)
Draws the underlying Field. Note that *object* is ignored (Display2D presently passes in the Field but could as well pass in null). The default implementation simply calls hitOrDraw(...).

public void hitObjects(DrawInfo2D range, Bag putInHere)
Places into the provided Bag all objects hit by the Clip rectangle of the given DrawInfo2D. The default implementation simply calls hitOrDraw(...).

protected void hitOrDraw(Graphics2D graphics, DrawInfo2D info, Bag putInHere)
Either draws the underlying field or places into the Bag all objects hit by the clip rectangle of the given DrawInfo2D. If graphics is null, then performs the hit-testing function: else performs the drawing function.

Field Portrayals are also **scaled** in the X and Y dimensions to fit within the expected region of the Display2D when the user zooms in and out. Sometimes it's helpful to know what the current scaling is:

sim.portrayal.FieldPortrayal2D Methods

public Double2D getScale(DrawInfo2D fieldPortrayalInfo)
Returns, as a Double2D, the width and height of a 1×1 unit in the Field Portrayal as specified by the given DrawInfo2D. The default implementation throws a RuntimeException: but overriden versions will return a proper value.

2D Field Portrayals also have a large collection of utility methods for translating between model and screen coordinates. To use them it's useful to distinguish between three different terms, which can be a bit confusing:

Why have methods for location? Why not just query the field?

Because certain fields may not have locations for objects, and thus no "getLocation" method. Notably, Network doesn't have "locations" for its objects or edges.

- The **object** is the object in the Field. Objects can be anything.
- The **location** of the object is where it's located in the Field. Locations can be any kind of object, as appropriate to the Field. Some fields have no "location" per se of objects.
- The **position** of the object is where it's located on-screen. Positions are instances of java.awt.geom.Point2D.Double. Some fields may have no "position" per se of objects.

sim.portrayal.FieldPortrayal2D Methods

public void setObjectPosition(Object object, Point2D.Double position, DrawInfo2D fieldPortrayalInfo)
Attempts to move the object to a new location in the Field to reflect a new position as provided. The default implementation does nothing.

public Point2D.Double getObjectPosition(Object object, DrawInfo2D fieldPortrayalInfo)
Returns the position of the given object, given a DrawInfo2D currently appropriate for the Field Portrayal as a whole. Returns null if the object does not exist or has no location which corresponds to a position.

public Point2D.Double getRelativeObjectPosition(Object location, Object otherObjectLocation, DrawInfo2D otherObjectInfo)

Returns the position of an object on-screen, using *another* object's location and the DrawInfo2D set up for that second object to draw itself. This is used in unusual cases when there is no DrawInfo2D available for the field as a whole.

public Object getObjectLocation(Object object, GUIState state)
Returns the location of the Object in the field, or null if this is not appropriate for the field or if the object does not exist. The default implementation simply returns null.

public Object getPositionLocation(Point2D.Double position, DrawInfo2D fieldPortrayalInfo)
Returns the position on-screen of a given location in the field, given a DrawInfo2D currently appropriate for the Field Portrayal as a whole. If locations are not appropriate for the field, returns null. The default implementation simply returns null.

public Point2D.Double getLocationPosition(Object location, DrawInfo2D fieldPortrayalInfo)
Returns the location in the field corresponding to a position on-screen, given a DrawInfo2D currently appropriate for the Field Portrayal as a whole, If locations are not appropriate for the field, returns null. The default implementation simply returns null.

Some "fast" Field Portrayals draw grids of rectangles. There are two ways this can be done:

- Draw each rectangle separately.
- Create an image the size of the grid. Poke pixels into the image, one per rectangle. Then stretch the image to fit in the given space. This is known as the *buffer* method.

In some operating systems (notably MacOS X), the second approach is *much* faster. As such, these "fast" Field Portrayals have the option of doing either, or of using the "default" form chosen by MASON appropriate to the operating system being run. The three possibilities, defined as variables in FieldPortrayal2D, are:

```
public static final int DEFAULT;
public static final int USE_BUFFER;
public static final int DONT_USE_BUFFER;
```

sim.portrayal.FieldPortrayal2D Methods

public void setBuffering(int val)
Sets the grid drawing approach to one of the three values above.

public int getBuffering()
Returns the current drawing approach, one of the three above.

9.2.6 Standard Field Portrayals

MASON provides at one standard Field Portrayal for each of its fields: and special standard Field Portrayals for hexagonal representations of grid fields. They are:

9.2.6.1 Field Portrayals for Object Grids

sim.portrayal.grid.ObjectGridPortrayal2D portrays fields of the form sim.field.grid.ObjectGrid2D. ObjectGridPortrayal2D generates wrappers by fixing the Object but allowing it to change location. If the object has moved, the wrapper will look nearby (no more than 3 units in any direction) to find the new location, and report this new location. If the object has moved further than this, it's too expensive to track and the wrapper will simply report that the object's location is "unknown".

sim.portrayal.grid.ObjectGridPortrayal2D Methods

public LocationWrapper getWrapper(Object object, Int2D location)
Produces a wrapper which allows the object to change but which loses the location of the object (it becomes "unknown") if it's moved more than 3 grid cells away at any one time.

---

sim.portrayal.grid.HexaObjectGridPortrayal2D also portrays fields of the form sim.field.grid.ObjectGrid2D under the assumption that they have been laid out as hexagonal grids. It uses the same wrapper facility as ObjectGridPortrayal2D.

9.2.6.2 Field Portrayals for Sparse Grids

sim.portrayal.grid.SparseGridPortrayal2D portrays fields of the form sim.field.grid.SparseGrid2D. Such grids allow Objects to pile up at the same location. To draw Objects in this situation requires a **draw policy**: a stipulation of which Objects should be drawn on top of which other Objects, and which to draw at all.

You can use the default draw policy (arbitrary ordering of Objects) or you can create your own. To do this, you'll need to implement a sim.portrayal.grid.DrawPlicy, which contains a single method:

sim.portrayal.DrawPolicy Methods

public boolean objectToDraw(Bag fromHere, Bag addToHere)
Potential objects to draw are provided in the bag *fromHere*. Places into the Bag *addToHere* those objects which should be drawn, and in the order they should be drawn, and returns true. Alternatively if all objects from *fromHere* are to be used in *addToHere* and in the given order, nothing is added to *addHere* and false is returned (more efficient).

SparseGridPortrayal2D also makes its own LocationWrappers: in this case, the Object stays fixed but its location may change. Note that the location isn't provided in the method, as SparseGrid2D can look it up efficiently. Unlike ObjectGridPortrayal2D, these LocationWrappers won't lose track of an object unless it has actually left the field.

sim.portrayal.grid.SparseGrid2D Methods

public LocationWrapper getWrapper(Object object)
Produces a wrapper which allows the location to change but which fixes the object.

public void setDrawPolicy(DrawPolicy policy)
Sets the draw policy of the field portrayal.

public DrawPolicy getDrawPolicy()
Returns the draw policy of the field portrayal.

---

sim.portrayal.grid.HexaSparseGridPortrayal2D also portrays fields of the form sim.field.grid.SparseGrid2D under the assumption that they have been laid out as hexagonal grids. It uses the same wrapper facility and draw policy mechanism as SparseGridPortrayal2D.

9.2.6.3 Field Portrayals for Grids of Bags of Objects

sim.portrayal.grid.DenseGridPortrayal2D portrays fields of the form sim.field.grid.DenseGrid2D. Just as in SparseGridPortrayal2D, these grids allow Objects to pile up at the same location, and so require a draw policy. But DenseGridPortrayal2D's LocationWrappers aren't like those in SparseGridPortrayal2D: they're instead exactly like those in ObjectGridPortrayal2D, and so can't track objects if they've moved too far away.

sim.portrayal.grid.DenseGrid2D Methods

public LocationWrapper getWrapper(Object object, Int2D location)
Produces a wrapper which allows the object to change but which loses the location of the object (it becomes "unknown") if it's moved more than 3 grid cells away at any one time.

public void setDrawPolicy(DrawPolicy policy)
Sets the draw policy of the field portrayal.

public DrawPolicy getDrawPolicy()
Returns the draw policy of the field portrayal.

---

sim.portrayal.grid.HexaDenseGridPortrayal2D also portrays fields of the form sim.field.grid.DenseGrid2D under the assumption that they have been laid out as hexagonal grids. It uses the same wrapper facility and draw policy mechanism as DenseGridPortrayal2D.

9.2.6.4 Field Portrayals for Grids of Numbers

sim.portrayal.grid.ValueGridPortrayal2D portrays fields of the form sim.field.grid.IntGrid2D and sim.field.grid.DoubleGrid2D. You can use various kinds of SimplePortrayals with this class, but the default portrayal (which is an instance of sim.field.grid.ValuePortrayal2D) works fine and draws each grid cell with a color corresponding to the value in the cell. You specify this value with a **color map** (a class discussed later in Section 12), which maps colors to values. This map can be specified with the method setMap(...).

ValueGridPortrayal2D also generates custom LocationWrappers, where its "objects" are actually numerical values. Unlike other LocationWrappers, these do not change location: but they change "object" (the value at that location) as time passes.

Last but not least, the ValueGridPortrayal2D portrayal must give a name to the numbers it's displaying (for purposes of inspection or tooltips): for example "Temperature" or "Population Density".

Relevant methods:

sim.portrayal.grid.ValueGridPortrayal2D Methods

public LocationWrapper getWrapper(double value, Int2D location)
Produces a wrapper which allows the value to change (it's the "object") but fixes the location.

public void setMap(ColorMap map)
Sets the color map used by the default simple portrayal (a ValuePortrayal2D).

public ColorMap getMap()
Returns the color map used by the default simple portrayal (a ValuePortrayal2D).

public void setValueName(String name)
Sets the name used to describe the values in the grid.

public String getValueName()
Returns the name used to describe the values in the grid.

ValueGridPortrayal2D also has a special constructor which sets the value name:

sim.portrayal.grid.ValueGridPortrayal2D Constructor Methods

public ValueGridPortrayal2D(String valueName)
Constructs the ValueGridPortrayal2D, setting the name used to describe the values in the grid.

Important Note: Hexagonal Fields There is also a Hexagonal version of ValueGridPortrayal2D called sim.portrayal.grid.HexaValueGridPortrayal2D. This field portrayal shares more in common with "fast" Field Portrayals: it does not use a SimplePortrayal. Instead it simply draws its values as hexagons using the given ColorMap for color. Note that there is *still* a "fast" version of this portrayal! It's called sim.portrayal.grid.FastHexaValueGridPortrayal2D, and it's fast (indeed *much* faster) because instead of drawing hexagons, it draws rectangles organized like bricks: this allows it to pull off the same image-stretching tricks discussed later in Section 9.2.7.

Why doesn't HexaValueGridPortrayal2D use a SimplePortrayal?

Because it's just unacceptably slow. Drawing and computing hexagons is slow enough. Calling forth a subsidiary object to do it for you is slower still.

9.2.6.5 Field Portrayals for Continuous Space

sim.portrayal.continuous.ContinuousPortrayal2D portrays fields of the form sim.field.continuous.Continuous2D. The ContinuousPortrayal2D class is similar in many ways to SparseGridPortrayal2D: objects may occupy the same location, and the LocationWrappers can track an object as it changes location (and look up its location efficiently). However ContinuousPortrayal2D does *not* use a DrawPolicy, unlike SparseGridPortrayal2D.

Objects in continuous space, unlike objects in grids, can wrap around if the field is considered to be toroidal. Thus ContinuousPortrayal2D has the option of displaying an overlapping object on *both* sides in which it overlaps to create the illusion of wrapping around in a toroidal fashion.

It's also not quite clear where the boundary *is* in a continuous space, particularly if the Display2D's clipping has been turned off. So ContinuousPortrayal2D has the option of drawing a *frame* around the boundary of the field.

sim.portrayal.continuous.ContinuousPortrayal2D Methods

public void setFrame(Paint p)
Causes the portrayal to draw a frame around the bounds of the field with the given paint. If the paint is null, nothing is drawn.

public Paint getFame()
Returns the paint with which the portrayal is drawing the frame.

public void setDisplayingToroidally(boolean val)
Causes the portrayal to display wrap-around objects on both sides (or if in a corner, potentially four times), or clears this feature.

public boolean isDisplayingToroidally()
Returns whether or not the portrayal is displaying objects toroidally.

public LocationWrapper getWrapper(Object obj)
Produces a wrapper which allows the location to change but which fixes the object.

A Note on Mixing Continuous and Grid Space In an $n \times m$ continuous space, an object at $\langle 0, 0\rangle$ is drawn so that its center lies on the origin of the bounding rectangle of the space. But in a grid, an object at $\langle 0, 0\rangle$ is centered on the first grid square. This is a different location, as shown in the Figure at right.

When you overlap continuous and grid field portrayals, you'll want to be mindful of this. You probably will want to translate one or the other so that these two locations line up. The easiest way to do this is to translate the continuous portrayal by $\frac{1}{2n}$ and $\frac{1}{2m}$ when it's attached to the Display2D (see Section 9.1, page 151).

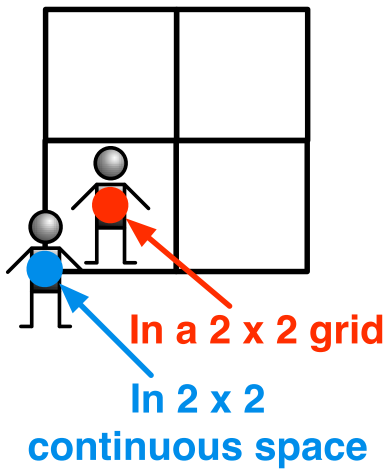


Figure 9.3 Centers of objects at $\langle 0, 0\rangle$ in a grid and in continuous space.

9.2.7 Fast Field Portrayals

For each of the grid Field Portrayals, there is often a **"fast" Field Portrayals** which does the same thing but much faster. The trade-off is flexibility: a "fast" Field Portrayal doesn't draw objects using a SimplePortrayal, but rather just draws them as a grid of colored rectangles. The color is determined using a **Color Map** (discussed in Section 12, a basic object which maps values to colors.

Fast Field Portrayals also take advantage of FieldPortrayal's **immutable field** feature (Section 9.2.4: instead of requerying the field, they may, if useful, simply re-draw the same thing over and over again. If your field changes very slowly, you can keep it immutable usually but force a redraw occasionally with a well-timed setDirtyField(true).

Most Fast Field Portrayals draw their grid of rectangles either by drawing separate rectangles one by one, or by poking pixels in an image, then stretching the image to fill the region (each pixel thus stretches into a rectangle). Which technique is faster depends on the operating system: on OS X, for example, it's much faster to poke pixels in an image. At the very end of Section 9.2.5 (2D Field Portrayals) we discussed the setBuffering(...) and getBuffering(...) methods, which determine what technique is used (or if MASON is free to pick one on its own appropriate to the operating system).

The most common Fast Field Portrayal is sim.display.grid.FastValueGridPortrayal2D, which draws grids of numbers as colored rectangles.

sim.portrayal.grid.FastValueGridPortrayal2D Constructor Methods

public FastValueGridPortrayal2D(String valueName, boolean immutableField)
Constructs the FastValueGridPortrayal2D, setting the name used to describe the values in the grid, and whether it's immutable.

public FastValueGridPortrayal2D(String valueName)
Constructs the FastValueGridPortrayal2D, setting the name used to describe the values in the grid. The field is assumed to not be immutable.

public FastValueGridPortrayal2D(boolean immutable)
Constructs the FastValueGridPortrayal2D, using a default name to describe the values in the grid, and and specifying whether it's immutable.

public FastValueGridPortrayal2D()
Constructs the FastValueGridPortrayal2D, using a default name to describe the values in the grid. The field is assumed to not be immutable.

FastValueGridPortrayal2D shares the same methods with ValueGridPortrayal2D for setting the Color Map:

sim.portrayal.grid.FastValueGridPortrayal2D Methods

public void setMap(ColorMap map)
Sets the color map used by the default simple portrayal (a ValuePortrayal2D).

public ColorMap getMap()
Returns the color map used by the default simple portrayal (a ValuePortrayal2D).

There is also a hexagonal version, sim.portrayal.grid.HexaFastValueGridPortrayal2D (try saying *that* three times fast!). It has exactly the same constructors and issues as FastValueGridPortrayal2D. It draws values as rectangles rather than as hexagons.

Another Fast Field Portrayal is sim.portrayal.grid.FastObjectGridPortrayal2D. This class represents Objects in the ObjectGrid2D as colored rectangles. FastObjectGridPortrayal2D is quite unusual because it uses *another* "fast" FieldPortrayal (specifically FastValueGridPortrayal2D) to actually handle its drawing.

Drawing works roughly like this. First, FastObjectGridPortrayal2D translates objects in the ObjectGridPortrayal into numbers. These numbers are then stored in a private DoubleGrid2D. FastObjectGridPortrayal2D then calls on its own private FastValueGridPortrayal2D to draw this DoubleGrid2D.

In order to be converted into numbers, the Objects in the ObjectGrid2D must be either instances of java.util.Number or they must implement the sim.util.Valuable interface (see Section 3.5). If they're neither Valuable nor Numbers, the Objects are assumed to be 1.0, unless they are null, in which they are assumed to be 0.0.

Alternatively, you can override the following method to convert the Objects as you see fit:

sim.portrayal.grid.FastObjectGridPortrayal2D Methods

public double doubleValue(Object obj)
Returns the double value associated with the given object in the field. The default implementation returns the number value if the Object is a Number or is sim.util.Valuable. Else if the Object is null, 0.0 is returned, else 1.0 is returned. Customize this as you see fit if necessary.

Once they're numbers, ObjectGrid2D uses exactly the same Color Map methods as FastValueGridPortrayal2D to convert them into colors.

FastObjectGridPortrayal2D has the following constructors:

sim.portrayal.grid.FastObjectGridPortrayal2D Constructor Methods

public FastObjectGridPortrayal2D(boolean immutable)
Constructs the FastObjectGridPortrayal2D, specifying whether it's immutable.

public FastValueGridPortrayal2D()
Constructs the FastObjectGridPortrayal2D. The field is assumed to not be immutable.

Again, there is also a hexagonal version, sim.portrayal.grid.HexaFastObjectGridPortrayal2D. It has exactly the same constructors and issues as FastObjectGridPortrayal2D.

9.2.8 Field Portrayals for Networks

Field Portrayals for Networks aren't what you expect. Rather than portray the edges and nodes in a network, in fact, they just **portray the edges alone**. Instead of drawing nodes, Network Field Portrayals let other Field Portrayals draw the nodes for them.

Why do this? Because elements in graphs don't have locations in space per se. If you draw a graph structure, you need only to specify the location of the nodes, and the edges are drawn according to that. But how are your nodes embedded in space? Are they in a continuous space? In a grid world? In a hexagonal environment?

In 2D, the Field Portrayal for Networks is, not surprisingly, sim.portrayal.network.NetworkPortrayal2D. It draws no nodes, only edges. Thus to draw the network you'll need to embed all of the Network's nodes in a sim.field.grid.SparseGrid2D or a sim.field.continuous.Continuous2D field, and then draw the nodes with one of the following:

How about hexagonal grids?

Right now NetworkPortrayal2D can only handle non-hexagonal SparseGrid2D and Continuous2D, because it computes the math itself rather than querying the underlying Field Portrayal. Perhaps later we'll retool it to do hexagonal if there's demand.

- sim.portrayal.grid.SparseGridPortrayal2D
- sim.portrayal.continuous.ContinuousPortrayal2D

Hint: When attaching the NetworkPortrayal2D and the node's Field Portrayal above, attach the NetworkPortayal2D first, so it draws first and the edges appear under the nodes. It generally looks better that way.

NetworkPortrayal2D doesn't just draw the edges in isolation: it still needs to know *where* the nodes are located on the screen. The way it does this is by querying each node's Field as to where it drew the node, then using that information to draw the edge.

The problem here is that NetworkPortrayal2D thus needs *two* fields: the Network and either a SparseGrid2D or Continuous2D field. But the setField(...) method only passes in *one* field. NetworkPortrayal2D gets around this by inventing a new field, called a sim.portrayal.network.SpatialNetwork2D, which simply holds the other two fields. You then pass the SpatialNetwork2D field into NetworkPortrayal2D.

A SpatialNetwork2D is constructed like this:

sim.portrayal.network.SpatialNetwork2D Constructor Methods

public SpatialNetwork2D(SparseField2D field, Network network)
Constructs the field with the given sparse field (either SparseGrid2D or Continuous2D) and a Network.

In fact, if you're careful, you can embed the **from** nodes in one sparse grid or continuous field, and embed the **to** nodes in *another* field. This might be useful, for example, for modeling a set of observers who observe a set of targets.

To set the **auxillary field** (the field of "to" nodes, if you want to differentiate them from the "from" nodes), and to get a given Object's location and the dimensions of the field, you have the methods:

sim.portrayal.network.SpatialNetwork2D Methods

public setAuxillaryField(SparseField2D field)
Sets the auxiliary field (the field for the "to" nodes) if it is different than the primary SparseField2D.

public Double2D getDimensions()
Returns the width and height of the primary field.

Drawing edges isn't the same as drawing objects at single locations: edges require a start point and an endpoint. Thus NetworkPortrayal2D sends to its SimplePortrayals a special subclass of DrawInfo2D, called sim.portrayal.network.EdgeDrawInfo2D, which adds the second point (the endpoint):

```
public Point2D.Double secondPoint;
```

The EdgeDrawInfo2D class has several constructors which extend the standard DrawInfo2D constructors:

sim.portrayal.EdgeDrawInfo2D Constructor Methods

public EdgeDrawInfo2D(GUIState gui, FieldPortrayal fieldPortrayal, RectangularShape draw, RectangularShape clip, Point2D.Double other)
Builds a DrawInfo2D from the given draw and clip rectangles. precise and selected are both set to false and location is set to null. The second point is provided by "other".

public EdgeDrawInfo2D(DrawInfo2D other, double translateX, double translateY, Point2D.Double other)
Builds a DrawInfo2D with its draw and clip rectangles translated by a certain factor from another DrawInfo2D. precise is copied from the other, but selected is set to false and location is set to null. The second point is provided by "other".

public EdgeDrawInfo2D(DrawInfo2D other, Point2D.Double other)
Builds a DrawInfo2D with its draw and clip rectangles copied from another DrawInfo2D. precise is copied from the other, but selected is set to false and location is set to null. The second point is provided by "other".

public EdgeDrawInfo2D(EdgeDrawInfo2D other)
Builds a DrawInfo2D with its draw and clip rectangles copied from another DrawInfo2D, as well as the second point. precise is copied from the other, but selected is set to false and location is set to null.

EdgeDrawInfo2D has the following method:

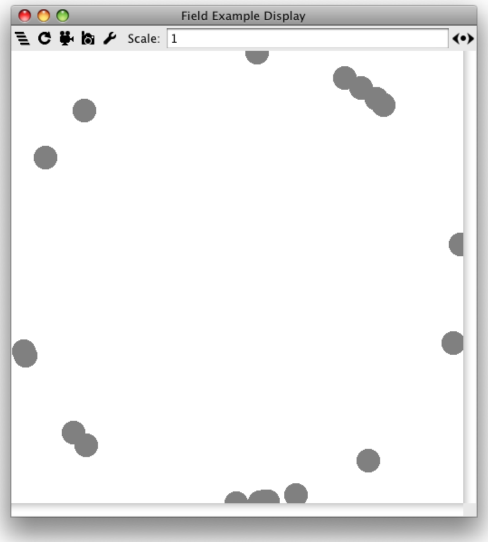
sim.portrayal.EdgeDrawInfo2D Methods

public String toString())
Produces a String describing the EdgeDrawInfo2D, essentially an extension to DrawInfo2D.toString().

9.2.9 Implementing a Field Portrayal

Fields are easy to implement: but Field Portrayals are not. They're complicated because of the various drawing, selection, and hit-testing tasks they must perform. But don't despair: you don't have to do *all* of that stuff. For example, you could write a Field Portrayal which just draws: this is a lot easier. Then you can add hit testing, selection, and translation at your leisure.

That's what we'll do here. In the following example, we'll create a Field Portrayal for the 1D Sparse Field we built in Section 5.3.1.2. The 1D Sparse Field allowed the user to associate objects with real-valued numbers ranging from 0.0 to 1.0 inclusive. Our Field Portrayal will draw these objects in circle. The Figure at right shows the general idea of what it'll look like in the end.



Note Most 2D Fields have a natural width and height which the Field Portrayal naturally exploits to scale to the Draw rectangle of the Display. Ours does not have a natural width and height (what's the "height" of a range from 0.0 to 1.0? Is it 1? If so, then what's the "width"?). So we're going to define an arbitrary width and height for our circle in"field units": 20 by 20. Since individual objects in a Field are generally displayed to roughly fill a 1 $\times$ 1 square, this will make nice big objects on our circle.

9.2.9.1 Drawing

Let's start by implementing a Field Portrayal which only does drawing. No hit-testing or selection, no mouse handling, no translation of positions to locations. Let's start with the boilerplate (which will include some classes we'll need for later sections too):

```
import sim.portrayal.FieldPortrayal2D;
import sim.portrayal.SimplePortrayal2D;
import sim.portrayal.Portrayal;
import sim.portrayal.LocationWrapper;
import sim.portrayal.DrawInfo2D;
import sim.util.Double2D;
import java.awt.Graphics2D;
import sim.util.Bag;
import java.awt.geom.Point2D;
import sim.portrayal.simple.OvalPortrayal2D;
import java.awt.geom.Rectangle2D;
import java.util.HashMap;

public class BoundedRealSparseFieldPortrayal2D extends FieldPortrayal2D
     {
```

Next we'll override the setField(...) method so that it verifies that the field is of the proper class.

```
    public void setField(Object field)
         {
         if (field instanceof BoundedRealSparseField)
              super.setField(field);
         else throw new RuntimeException("Invalid field: " + field);
         }
```

The Field Portrayal needs to override a single abstract method, getDefaultPortrayal(), to return a Simple Portrayal to use when none else can be determined for a given Object. Here we'll just use a sim.portrayal.simple.OvalPortrayal2D(), which in its default form, simply draws a gray filled circle.

```
SimplePortrayal2D defaultPortrayal = new OvalPortrayal2D();

public Portrayal getDefaultPortrayal()
     {
     return defaultPortrayal;
     }
```

Next we'll handle the getScale(...) method. This method takes a DrawInfo2D for the FieldPortayal as a whole and returns a sim.util.Double2D containing the width and height, in pixels of 1 unit of width and height in the field coordinate space. The FieldPortrayal provided has as its Draw rectangle the expected bounds for the Field as a whole. So usually the scale is computed by simply dividing the FieldPortrayal's Draw width and height by the underlying Field's width and height.

However we can't do that in this case: our Field (a 1-dimensional range of numbers) doesn't have a "width" or "height". What we're planning to do is draw, on a ring, all the objects in the Field: objects stored at 0.0 would be at the 12-o'clock position, objects at 0.5 would be at the 6-o'clock position, and so on. So let's imagine a width and height for our ring: 20.0 "units", so to speak, in Field coordinates. 20 is a nice number for another reason: objects will be drawn to roughly fill a 1×1 space of field coordinates, so our field will have nice big objects being drawn (1/20 of the "height" of the field).

Thus we have:

```
public Double2D getScale(DrawInfo2D fieldPortrayalInfo)
     {
     double boundsx = 20.0;    // our pretend "width"
     double boundsy = 20.0;    // our pretend "height"
     double xScale = fieldPortrayalInfo.draw.width / boundsx;
     double yScale = fieldPortrayalInfo.draw.height / boundsy;
     return new Double2D(xScale, yScale);
     }
```

Note that in many more "real" cases the getScale(...) method accesses the field directly to get some of this information, and so must synchronize on the schedule.

Next we're going to do the actual drawing, by overriding the draw(...) method (duh). The way we draw is as follows:

1. Compute the scale of the field.
2. Determine if any object is presently being selected (if so, this makes drawing a bit slower).
3. Compute (as startx, starty, endx, endy) the portion of the field which is being shown on-screen. This is done by scaling the Clip rectangle into the field's coordinate space.
4. Create a DrawInfo2D to hand to SimplePortrayals to tell them where to draw. I'll have our clip rectangle but we'll change the draw rectangle to tell the SimplePortrayals to draw themselves in different places.
5. For each object in the Field
   (a) Compute where it should be drawn (on our little ring)
   (b) Determine if where it's being drawn falls within the clip region
   (c) If so, call forth a SimplePortrayal, set up the DrawInfo2D, set the object as selected or not, and have the SimplePortrayal draw it.

For our purposes this will suffice: but obviously for a Field with lots of objects, you'll want to have a more sophisticated way of whittling down which objects get drawn on-screen: instead of doing the $O(n)$ process of going through every object on the Field, you might extract objects only from that portion of the Field which overlap with the Clip rectangle.

You'll notice in the code below the use of a "slop" variable. What's going on here is: objects can have their centers off-screen but still overlap somewhat on-screen. Since objects generally are drawn filling a 1×1

square in field coordinates, a 0.5 slop in our drawn bounds should be sufficient to guarantee objects will be drawn. If you have objects taking up a larger area than this, you may have to make the slop bigger in this example. A slop of this size will also nicely work when doing hit-testing later on.

```
public void draw(Object object, Graphics2D graphics, DrawInfo2D info)
    {
    final BoundedRealSparseField field = (BoundedRealSparseField)getField();
    if (field == null) return;

    Double2D scale = getScale(info);
    boolean someObjectIsPresentlySelected = !selectedWrappers.isEmpty();

    // compute the (startx, starty) and (end, endy) endpoints of the clip rectangle
    // in the field's bounding region.
    double startx = (info.clip.x - info.draw.x) / scale.x;
    double starty = (info.clip.y - info.draw.y) / scale.y;
    double endx = (info.clip.x - info.draw.x + info.clip.width) / scale.x;
    double endy = (info.clip.y - info.draw.y + info.clip.height) / scale.y;

    // Build a DrawInfo2D which uses the old clip rectangle.
    // We'll reuse it for various objects
    DrawInfo2D newinfo = new DrawInfo2D(new Rectangle2D.Double(0, 0, scale.x, scale.y), info.clip);
    newinfo.fieldPortrayal = this;

    // hit or draw each object
    Bag objs = field.getAllObjects();
    int len = objs.size();
    for(int i = 0; i < len; i++)
        {
        Object obj = objs.get(i);
        double loc = ((Double) (field.getObjectLocation(obj))).doubleValue();

        // we'll display 'em in a ring!  Remember our origin is (0,0) and width and
        // height is 20.0.
        double locx = Math.cos(2*Math.PI*loc) * 10.0 + 10.0;
        double locy = Math.sin(2*Math.PI*loc) * 10.0 + 10.0;

        // is it within the clip region?  Give it a slop of at least 0.5 in each direction
        // for two reasons.  First, this creates a 1.0 x 1.0 rectangle around the object
        // which will intersect with clip rects nicely for hit-testing.  Second, it gives
        // objects whose centers are off-screen but whose bodies still peek into the
        // slip region a chance to draw themselves [recall that objects are supposed to be
        // roughly 1.0 x 1.0 field units in size]  If you have objects that draw bigger than
        // this, you may wish to have a larger slop.  In this example, we don't.

        final double slop = 0.5;
        if (locx >= startx - slop && locx <= endx + slop && locy >= starty - slop && locy <= endy + slop)
            {
            // get the SimplePortrayal
            Portrayal p = getPortrayalForObject(obj);
            if (!(p instanceof SimplePortrayal2D))         // uh oh
                throw new RuntimeException("Unexpected Portrayal " + p + " for object " + obj);
            SimplePortrayal2D portrayal = (SimplePortrayal2D) p;

            // load the DrawInfo2D
            newinfo.draw.x = (info.draw.x + scale.x * locx);
            newinfo.draw.y = (info.draw.y + scale.y * locy);

            // Set selected (or not) and draw
            newinfo.selected = someObjectIsPresentlySelected && selectedWrappers.get(obj) != null;
            portrayal.draw(obj, graphics, newinfo);
            }
        }
    }
```

9.2.9.2 Hit Testing

The above is sufficient for drawing our objects in a ring: but if we want to select or inspect them, we'll need to perform **hit testing**. It turns out that hit-testing is nearly identical in code to drawing. But before we get to that, we need to start by discussing the concept of a **Stable Location**.

MASON's inspectors, discussed in Section 10, allow the user to inspect both the properties of an object and its location in its field. In some kinds of fields, the object may change but the location stays the same (the inspector **fixes on the location**. In other fields — the more common situation — the object stays the same but the location may change. We're in this second category.

The problem is that the location is an immutable object such as sim.util.Double2D, and so when an object moves about in the field, its location is constantly changed to new instances of Double2D. This causes MASON's inspectors to have to rebuild themselves each time, which is expensive. But we can get around it by creating a special "location" object, called a **stable location**, which queries the underlying field for the current location of the object, then changes its properties to reflect that location. A stable location will never change: but its property values will. The stable location will be provided to LocationWrappers in lieu of the actual location.

StableLocations implement the sim.portrayal.inspector.StableLocation interface, which defines a single method (though you'll need to implement more methods than this):

sim.portrayal.inspector.StableLocation Methods

public String toString()
Returns the current location as a String.

Additionally, a StableLocation will provide various Java Bean Properties (get/set methods) for each of the features of the location. For example, MASON provides the StableLocation sim.portrayal.inspector.StableDouble2D, which provides three properties: the present or last known X value, the present or last known Y value, and whether or not the object presently exists in the field.

In our case, we have a single value as our location, and a boolean indicating whether the object exists in the field. Here's how I'd write our class:

```
import sim.portrayal.inspector.StableLocation;

public class StableDouble1D implements StableLocation
    {
    double x = 0;      // our current value
    boolean exists = false;   // is the object in the field?
    Object object;
    BoundedRealSparseField field;

    public StableDouble1D(Object object, BoundedRealSparseField field)
        {
        this.object = object;
        this.field = field;
        }

    void update()   // re-gather information about our location and existence
        {
        Double pos = null;
        if (field != null) pos = field.getObjectLocation(object);
        if (pos == null) { exists = false; }  // don't update x so it stays the same
        else { x = pos.doubleValue(); exists = true; }
        }

    public double getValue() { update(); return x; }
    public boolean getExists() { update(); return exists; }

    public void setValue(double val)
        {
```

```
        if (field!=null) field.setObjectLocation(object, new Double(val));
        x = val;
        exists = true;
        }

    public String toString()
        {
        update();
        if (!exists) return "Gone";
        else return "" + x;
        }
    }
```

MASON provides some StableLocation objects for you already:

- sim.portrayal.inspector.StableDouble2D wraps around Double2D and expects a Continuous2D field.
- sim.portrayal.inspector.StableDouble3D wraps around Double3D and expects a Continuous3D field..
- sim.portrayal.inspector.StableInt2D wraps around Int2D and expects a SparseGrid2D field.
- sim.portrayal.inspector.StableInt3D wraps around Int3D and expects a SparseGrid3D field.

These classes are essentially the same implementation as the one above: they have a constructor which takes the Object and some kind of SparseField, plus various get and set methods for Inspectors, and a toString() method.

Armed with a StableLocation, we can now create our LocationWrapper. Back to the BoundedRealSparseFieldPortrayal2D class:

```
    public LocationWrapper getWrapper(final Object obj)
        {
        final BoundedRealSparseField field = (BoundedRealSparseField)this.field;
        final StableDouble1D loc = new StableDouble1D(obj, field);

        return new LocationWrapper(obj, null, this) // don't care about location, we're updating it below
            {
            public Object getLocation()
                {
                // always call update just in case
                loc.update();
                return loc;
                }

            public String getLocationName()
                {
                return getLocation().toString();
                }
            };
        }
```

As you can see, we created a LocationWrapper which takes an object and a location but ignores the location. Instead, we override the location-related methods to return the current underlying location of the object (as it moves about), using a StableDouble1D as our location which never changes.

Now we can add our hit-testing code. Ordinarily we'd add it to the method hitObject(...), which corresponds to the method draw(...). But as mentioned, this code is very close to identical to the drawing code in Field Portrayals, and so nearly all Field Portrayals instead override a method which does both of them at the same time, to save some replication. That method is called hitOrDraw(...).

So: delete the draw(...) method in the code and replace it with:

```
    protected void hitOrDraw(Graphics2D graphics, DrawInfo2D info, Bag putInHere)
        {
```

```
final BoundedRealSparseField field = (BoundedRealSparseField)getField();
if (field == null) return;

Double2D scale = getScale(info);

boolean someObjectIsPresentlySelected = !selectedWrappers.isEmpty();

// compute the (startx, starty) and (end, endy) endpoints of the clip rectangle
// in the field’s bounding region.
double startx = (info.clip.x - info.draw.x) / scale.x;
double starty = (info.clip.y - info.draw.y) / scale.y;
double endx = (info.clip.x - info.draw.x + info.clip.width) / scale.x;
double endy = (info.clip.y - info.draw.y + info.clip.height) / scale.y;

// Build a DrawInfo2D which uses the old clip rectangle.
// We’ll reuse it for various objects
DrawInfo2D newinfo = new DrawInfo2D(new Rectangle2D.Double(0, 0, scale.x, scale.y), info.clip);
newinfo.fieldPortrayal = this;

// hit or draw each object
Bag objs = field.getAllObjects();
int len = objs.size();
for(int i = 0; i < len; i++)
    {
    Object obj = objs.get(i);
    double loc = ((Double) (field.getObjectLocation(obj))).doubleValue();

    // we’ll display ’em in a ring!  Remember our origin is (0,0) and width and
    // height is 20.0.
    double locx = Math.cos(2*Math.PI*loc) * 10.0 + 10.0;
    double locy = Math.sin(2*Math.PI*loc) * 10.0 + 10.0;

    // is it within the clip region?  Give it a slop of at least 0.5 in each direction
    // for two reasons.  First, this creates a 1.0 x 1.0 rectangle around the object
    // which will intersect with clip rects nicely for hit-testing.  Second, it gives
    // objects whose centers are off-screen but whose bodies still peek into the
    // slip region a chance to draw themselves [recall that objects are supposed to be
    // roughly 1.0 x 1.0 field units in size]  If you have objects that draw bigger than
    // this, you may wish to have a larger slop.  In this example, we don’t.

    final double slop = 0.5;
    if (locx >= startx - slop && locx <= endx + slop &&
        locy >= starty - slop && locy <= endy + slop)
        {
        // get the SimplePortrayal
        Portrayal p = getPortrayalForObject(obj);
        if (!(p instanceof SimplePortrayal2D)) // uh oh
            throw new RuntimeException("Unexpected Portrayal " + p + " for object " + obj);
        SimplePortrayal2D portrayal = (SimplePortrayal2D) p;

        // load the DrawInfo2D
        newinfo.draw.x = (info.draw.x + scale.x * locx);
        newinfo.draw.y = (info.draw.y + scale.y * locy);

        // draw or hit
        if (graphics == null)  // hit
            {
            if (portrayal.hitObject(obj, newinfo)) putInHere.add(getWrapper(obj));
            }
        else // draw.  Be sure to set selected first
            {
            newinfo.selected = someObjectIsPresentlySelected && selectedWrappers.get(obj) != null;
            portrayal.draw(obj, graphics, newinfo);
            }
        }
        }
    }
```

As you can see, precious little new code. Our hit code is simply testing if the SimplePortrayal hit the object, and if so, creating a LocationWrapper for it and dumping it in a Bag.

To finish up, the above code needs a (presently empty) HashMap called selectedWrappers, where it checks to see if the object being drawn is "selected" (that is, a member of the HashMap). We'll make more use of this HashMap in a moment:

```
HashMap selectedWrappers = new HashMap();
```

9.2.9.3 Selection

Next we're going to add selection. You'll notice that we had a little selection code in the previous sections. In 2D Field Portrayals, selection works like this:

1. The user selects an object or objects
2. Display2D tells the appropriate Field Portrayals to *deselect* all currently selected objects and to *select* the objects in question.
3. The Display2D maintains a list of all currently selected objects.
4. When a Field Portrayal is supposed to draw an object, it first sets the "selected" flag, as appropriate, in the DrawInfo2D passed to the SimplePortrayal
5. The SimplePortrayal responds to the selected flag to draw the object in a special way.

How does the Field Portrayal remember what objects were selected so it can inform the SimplePortrayals later on? This is up to individual Field Portrayals, but the standard code is the one we're going to use: just maintain a HashMap. The HashMap in question is the selectedWrappers map added as a dummy in the previous section.

```
HashMap selectedWrappers = new HashMap();
public boolean setSelected(LocationWrapper wrapper, boolean selected)
    {
    if (wrapper == null) return true;
    if (wrapper.getFieldPortrayal() != this) return true;

    Object obj = wrapper.getObject();
    boolean b = getPortrayalForObject(obj).setSelected(wrapper,selected);
    if (selected)
        {
        if (b==false) return false;
        else selectedWrappers.put(obj, wrapper);
        }
    else
        {
        selectedWrappers.remove(obj);
        }
    return true;
    }
```

setSelected(...) is called both for objects which have been selected and for ones which have been recently deselected. As you can see, this code just maintains a record of which have been selected and which have not. We then use that in the drawing code to set the "selected" flag prior to drawing.

9.2.9.4 Translation

Last but not least, 2D Field Portrayals typically implement four optional methods which translate between locations in the field and positions on-screen. This is not necessary unless you want to do things like drag objects about the screen with the mouse. But they're not that tough to write. The methods are:

- getObjectLocation(...) returns the location of an object in the underlying field.
- getPositionLocation(...) returns the position on-screen corresponding to a given location.
- getLocationPosition(...) returns the location in the field corresponding to a position on-screen.
- setObjectPosition(...) Changes the location of an object, if possible, to correspond to a new position on-screen.

One additional method we don't have to define, as it's just a composition of the getObjectLocation(...) and getLocationPosition(...) methods:

- getObjectPosition(...) returns the position-onscreen of an object in the field.

One important note: some of these methods access the field directly, and so must synchronize on the schedule.

Here are the implementations:

```
public Object getObjectLocation(Object object, GUIState gui)
    {
    synchronized(gui.state.schedule)
        {
        final BoundedRealSparseField field = (BoundedRealSparseField)getField();
        if (field == null) return null;
        return field.getObjectLocation(object);
        }
    }

public Object getPositionLocation(Point2D.Double position, DrawInfo2D fieldPortrayalInfo)
    {
    Double2D scale = getScale(fieldPortrayalInfo);

    // Convert the point to our point in (0,0 ... 20, 20) space
    double locx = (position.getX() - fieldPortrayalInfo.draw.x) / scale.x;
    double locy = (position.getY() - fieldPortrayalInfo.draw.y) / scale.y;

    // what point on our ring is closest to this?
    double val = Math.atan2(locy - 10.0, locx - 10.0) / (2*Math.PI);
    if (val < 0) val += 1.0;

    return new Double(val);
    }

public Point2D.Double getLocationPosition(Object location, DrawInfo2D fieldPortrayalInfo)
    {
    double loc = ((Double)location).doubleValue();
    if (loc < 0 || loc > 1) // uh oh
        return null;

    Double2D scale = getScale(fieldPortrayalInfo);

    // cast the number to our ring location
    double locx = Math.cos(2*Math.PI*loc) * 10.0 + 10.0;
    double locy = Math.sin(2*Math.PI*loc) * 10.0 + 10.0;

    // convert to a position on-screen
    double x  = (fieldPortrayalInfo.draw.x + scale.x * locx);
```

```
        double y = (fieldPortrayalInfo.draw.y + scale.y * locy);
        return new Point2D.Double(x,y);
        }

    public void setObjectPosition(Object object, Point2D.Double position, DrawInfo2D fieldPortrayalInfo)
        {
        synchronized(fieldPortrayalInfo.gui.state.schedule)
            {
            Object loc = getPositionLocation(position, fieldPortrayalInfo);
            final BoundedRealSparseField field = (BoundedRealSparseField)getField();
            if (field != null) field.setObjectLocation(object, (Double)loc);
            }
        }
```

You'll notice some similarity between this code and the drawing code. Perhaps this is an opportunity for merging code, though in MASON's implementations they're broken out to guarantee inlining speed.

9.3 2D Simple Portrayals

The final part of the visualization puzzle, after 2D Displays and 2D Field Portrayals, are **simple portrayals**. These are subclasses of sim.portrayal.Portrayal whose job is to draw individual objects or values stored a Field. Simple Portrayals are registered with Field Portrayals to draw specific objects; or objects of a certain class; or all objects in the Field, or null values, etc. Objects in a field may also automatically serve as their own Simple Portrayals.

There's no sim.portrayal.SimplePortrayal class?

Not at present, no. Though there is a sim.portrayal.SimplePortrayal2D class for 2D Simple Portrayals and a sim.portrayal.SimplePortrayal3D class for 3D Simple Portrayals.

Not all Field Portrayals use Simple Portrayals. Recall that, as discussed in Section 9.2.7, so-called "fast" Field Portrayals bypass the Simple Portrayal mechanism entirely and directly draw the objects in their fields, usually as rectangles colored using a sim.util.gui.ColorMap (discussed in Section 12) to translate values into colors.

For 2D visualization, Simple Portrayals subclass the sim.portrayal.SimplePortrayal2D class, which provides basic facilities. Unlike the 2D Field Portrayal facility, implementing a 2D Simple Portrayal is pretty easy if you have basic knowledge of Java's AWT or Java2D graphics.

There are several special kinds of 2D Simple Portrayals:

- **Basic Simple Portrayals** draw objects, often as ovals or rectangles.
- **Value Simple Portrayals** draw numbers as ovals or rectangles or hexagons: these Simple Portrayals commonly color their shapes using a sim.util.gui.ColorMap to translate between number values and colors.
- **Edge Simple Portrayals** draw edges in networks. These are designed to work with sim.portrayal.network.NetworkPortrayal2D and with a special subclass of DrawInfo2D called sim.portrayal.network.EdgeDrawInfo2D.
- **Wrapper Simple Portrayals** "wrap" around subsidiary Simple Portrayals to add additional gizmos to them. For example. to add a label, or to circle an object when it's selected, or to enable rotation or dragging or a trail, simply create a wrapper portrayal around your basic portrayal and submit the wrapper portrayal as the Simple Portrayal to the Field Portrayal. Wrapper portrayals can wrap other wrapper portrayals, creating a sequence of "wraps" around a basic portrayal.

We'll cover each of these in turn.

9.3.1 Basic Simple Portrayals

The most "basic" of the basic simple portrayals is sim.portrayal.SimplePortrayal2D itself. This simple portrayal refuses to draw the object it's given, and also doesn't respond to hit-testing or selection, etc. If you want your object to be invisible, this is the Simple Portrayal to use.

Four provided Basic Portrayals draw their objects as either filled or outlined shapes:

- sim.portrayal.simple.RectanglePortrayal2D draws its object as a rectangle.
- sim.portrayal.simple.OvalPortrayal2D draws its object as an oval.
- sim.portrayal.simple.ShapePortrayal2D draws its object as a 2D java.awt.Shape, which you provide. You can also provide, as two arrays, the X and Y points for a polygon.
- sim.portrayal.simple.HexagonalPortrayal2D draws its object as a hexagon. HexagonalPortrayal2D is a subclass of ShapePortrayal2D. HexagonalPortrayal2D is often used for drawing objects in hexagonal field portrayals.

These classes all have the same basic internal variables you can set, and constructors to match.

```
public Paint paint;     // the paint with which to fill or draw the shape outline
public double scale;    // how much to scale the object when drawing
public boolean filled;  // whether to fill or draw the shape outline
```

ShapePortrayal2D has an additional variable you can set:

```
public Stroke stroke;   // the Stroke with which to draw the shape
```

If this variable is set to null, ShapePortrayal2D will use a default java.awt.geom.BasicStroke() instead.

These classes are pretty easily customized. One standard trick is to subclass a Basic Portrayal to cause it to (say) change its color in response to some aspect or property of the Object: for example, if it's hungry or not. To do this, you'd override its draw(...) method to change some feature first, then call the superclass. For example:

```
public void draw(Object object, Graphics2D graphics, DrawInfo2D info)
    {
    MyObject obj = (MyObject) object;
    boolean val = obj.isHungry();
    if (val) paint = Color.red;
    else paint = Color.blue;
    super.draw(object, graphics, info);
    }
```

Here again, a sim.util.gui.ColorMap (Section 12) might be of use to you: you could store one permanently as an instance variable.

A final Basic SimplePortrayal provided in MASON is sim.portrayal.simple.ImagePortrayal2D. This class draws a small bitmap image to represent the object. The image can be transparent or semitransparent if you wish. The image is scaled as follows: if the image is taller than it is wide, then the image is scaled so that its width is exactly info.draw.width. If the image is wider than it is tall, then the image is scaled so that its height is exactly info.draw.height.

ImagePortrayal2D is a subclass of RectanglePortrayal2D, so it has an paint, scale, and filled variable. Only the scale variable is used: you can the image it further by changing (or passing in) the scale variable.

ImagePortrayal2D has a number of unusual constructors, so it's worthwhile explaining them here.

sim.portrayal.simple.ImagePortrayal Constructor Methods

public ImagePortrayal(javax.swing.ImageIcon icon)
Creates an ImagePortrayal at a scale of 1.0, and using the image from the provided ImageIcon.

public ImagePortrayal(javax.swing.ImageIcon icon, double scale)
Creates an ImagePortrayal at the provided scale, and using the image from the provided ImageIcon.

public ImagePortrayal(Class cls, String resourceName)
Creates an ImagePortrayal by loading an image resource of the name *resourceName*, which must be located right next to the class file (".class") of the given class. That is, uses getClass().getResource(...). The scale is 1.0.

public ImagePortrayal(Class cls, String resourceName, double scale)
Creates an ImagePortrayal by loading an image resource of the name *resourceName*, which must be located right next to the class file (".class") of the given class. That is, uses getClass().getResource(...). The scale is provided.

public ImagePortrayal(Image image)
Creates an ImagePortrayal from the given image, and a scale of 1.0.

public ImagePortrayal(Image image, double scale)
Creates an ImagePortrayal from the given image, using the given scale.

You could override ImagePortrayal2D to change its image to reflect a change in status of the underlying object, but it's not very efficient. Instead I'd use a special wrapper portrayal, sim.portrayal.simple.FacetedPortrayal2D instead. See Section 9.3.4.2.

9.3.2 Value Simple Portrayals

Some Field Portrayals draw fields which consist of arrays of numbers rather than objects. One particular Field Portrayal, sim.portrayal.grid.ValuePortrayal2D, relies on a SimplePortrayal to help it draw those objects.

What about HexaValuePortrayal2D?

Recall that HexaValuePortrayal2D just draws the objects directly as hexagons because of efficiency concerns. So it's only ValuePortrayal2D of interest here.

So if SimplePortrayal's draw(...) and hitObject(...) methods take Objects to draw/hit, what's passed in if there's no Object, but just a number? Answer: a sim.util.MutableDouble (Section 3.5). This object is little more than a wrapper around the number in question.

ValuePortrayal2D displays its number as a rectangle with a given color, using the ValueGridPortrayal2D's Color Map. As such, ValuePortrayal2D subclasses from RectanglePortrayal2D and inherits all of its features, except that the paint variable is set each time to reflect the desired color value.

ValuePortrayal2D, or a subclass of it, will *only* work with ValueGridPortrayal2D or a subclass of it.

9.3.3 Edge Simple Portrayals

Edges in graphs are different from simple objects. Rather than being drawn at a single location, edges are drawn *from* a given location *to* another location. For this reason, NetworkPortrayal2D requires a special kind of SimplePortrayal2D which understands how to get these two points and draw itself accordingly.

Whoa, inconsistency in naming convention.

Yeah. SimpleEdgePortrayal2D is the only 2D Simple Portrayal, besides SimplePortrayal2D, with the word "Simple" in front of it. And it's also the only one outside the sim.portrayal.simple package. This is mostly historical.

The Simple Portrayal in this category is sim.portrayal.network.SimpleEdgePortrayal2D, a special subclass of SimplePortrayal2D which expects a special subclass of DrawInfo2D called sim.portrayal.network.EdgeDrawInfo2D (Section 9.2.8) to provide it with both the "from" and "to" locations.

SimpleEdgePortrayal2D is not so simple. It can draw itself in several ways:

- Undirected edges can be drawn as thin lines of a specified color.

- Directed edges can be drawn as thin lines broken into two intervals, a "from" interval and a "to" interval, each with its own color.
- Directed edges can also be drawn as a triangle whose thick end is at the "from" node and whose point is at the "to" node. The triangle has a single specified color.
- Edges of all kinds can be drawn with a label.

To do this, SimpleEdgePortrayal2D has several public variables, some of which may be reminiscent of the Basic Simple Portrayals earlier. Here are the first four:

```
public Paint fromPaint;
public Paint toPaint;
public Paint labelPaint;
public Font labelFont;
```

The "paint" of the edge is *fromPaint*. When two paints are required (as in directed edges drawn with a line of two colors), then the *toPaint* is additionally used. The *labelPaint* is the paint of the label: if it is null (the default) then no label is drawn. The *labelFont* is the label's unscaled Font.

You can set all these in a constructor:

sim.portrayal.network.SimpleEdgePortrayal2D Constructor Methods

public SimpleEdgePortrayal2D(Paint fromPaint, Paint toPaint, Paint labelPaint, Font labelFont)
Creates a SimpleEdgePortrayal with the given from-paint, to-paint, label paint, and label font. If the label paint is null, no label will be drawn.

public SimpleEdgePortrayal2D(Paint fromPaint, Paint toPaint, Paint labelPaint)
Creates a SimpleEdgePortrayal with the given from-paint, to-paint, and label paint. A default font will be used for the label. If the label paint is null, no label will be drawn.

public SimpleEdgePortrayal2D(Paint edgePaint, Paint labelPaint)
Creates a SimpleEdgePortrayal with the given edge paint (used for both the from-paint and the to-paint) and label paint. A default font will be used for the label. If the label paint is null, no label will be drawn.

public SimpleEdgePortrayal2D()
Creates a SimpleEdgePortrayal which draws using black as its edge color, and no label.

Two other variables control how the edge is drawn:

```
public double baseWidth;
public int shape;
```

The *shape* variable determines whether the SimpleEdgePortrayal2D draws itself as a line or as a triangle. It can be set to one of the two constants:

```
public static final int SHAPE_LINE;
public static final int SHAPE_TRIANGLE;
```

Finally, the *baseWidth* works as follows. By default it is 0.0. The baseWidth determines either the **width of the line** or the **width of the "from" end of the triangle**. If you create a triangle, you'll need to change the base width, ideally to 1.0.

When drawing as a triangle or a line of non-zero width, you have the option of specifying how the triangle or line width changes when you zoom in or out. This is called the **scaling** of the edge, and it's one of three constants:

```
public static final int NEVER_SCALE;
public static final int SCALE_WHEN_SMALLER;
public static final int ALWAYS_SCALE;
```

The default is ALWAYS_SCALE, which instructs the edge to get bigger or smaller when you zoom in and out as if you're examining it closer. This is *probably* what you want. An alternative is SCALE_WHEN_SMALLER which only scales when you're really zoomed out: if you're zoomed in, it stays small so as not to crowd the environment. You probably don't want to set scaling to NEVER_SCALE, which keeps the edges at their standard thickness regardless.

Various methods can be used to set these values:

sim.portrayal.network.SimpleEdgePortrayal2D Methods

public int getShape()
Returns the edge shape, one of SHAPE_LINE and SHAPE_TRIANGLE.

public void setShape(int shape)
Sets the edge shape, one of SHAPE_LINE and SHAPE_TRIANGLE.

public double getBaseWidth()
Returns the base width of the edge (0.0 by default).

public void setBaseWidth(double width)
Sets the base width of the edge. This must be ≥ 0.0. You must set the base width of a triangle in order to see a triangle.

public int getScaling()
Returns the scaling of the edge, one of NEVER_SCALE, SCALE_WHEN_SMALLER, or ALWAYS_SCALE (the default).

public void setScaling(int scaling)
Returns the scaling of the edge to one of NEVER_SCALE, SCALE_WHEN_SMALLER, or ALWAYS_SCALE (the default).

Edge Labels and Weights Underlying edges in graphs often have associated data (labels or weights), provided by their info instance variables. SimpleEdgePortrayal2D can portray this information in one of two ways. First, it can draw a **label**: a string of text describing the edge. Second, if the data is a weight, that is, it takes the form of a numerical value, SimpleEdgePortrayal2D can adjust the *thickness* of the line by multiplying the base width by the absolute value of this weight.

Let's handle the first one first. Labels are drawn if you specify the label color, as discussed earlier. You can also specify a label font, though that's less common. So how do you specify what the label *is?* SimpleEdgePortrayal2D queries the method getLabel(...), which you can override, to provide the label of an Edge.

If you set setAdjustsThickness(true), you can also turn on SimpleEdgePortrayal2D's ability to automatically adjust the thickness of the edge to reflect the underlying edge.info value interpreted as a number. You must also set the baseWidth as well: I recommend setting it to 1.0.

To determine the numerical value of the edge weight, SimpleEdgePortrayal2D calls a method called getPositiveWeight(...), which you can override if you like. Here are the methods in question:

sim.portrayal.network.SimpleEdgePortrayal2D Methods

public String getLabel(Edge edge, EdgeDrawInfo2D info)
Returns a label for the edge. The default implementation returns the empty String if the edge.info is null, else it calls edge.info.toString().

public double getPositiveWeight(Edge edge, EdgeDrawInfo2D info)
Returns a positive weight for the edge. The default implementation returns the absolute value of the edge.info object if it is a java.lang.Number or if it is sim.util.Valuable, else it returns 1.0.

public boolean getAdjustsThickness()
Returns whether or not the portrayal is adjusting the edge width to reflect the edge label interpreted as a weight.

public void setAdjustsThickness(boolean val)
Sets whether or not the portrayal is adjusting the edge width to reflect the edge label interpreted as a weight.

If you'd like to (say) change the edge *color* to reflect a weight or label, an easy way is to subclass the edge in a manner similar to the example given for the Basic Simple Portrayals. For example, the following code might change color based on whether or not the edge.info value is null:

```
public void draw(Object object, Graphics2D graphics, DrawInfo2D info)
    {
    Edge edge = (Edge) object;
    if (edge.info == null) { toPaint = fromPaint = Color.red; }
    else { toPaint = fromPaint = Color.blue; }
    super.draw(object, graphics, info);
    }
```

9.3.4 Wrapper Simple Portrayals

I notice all these names are adjectives

Yep. Wrapper portrayals are given adjectives describing their added functionality, and other simple portrayals are given nouns as names.

MASON provides a whole bunch of **wrapper portrayals** which greatly enhance your 2D Simple Portrayal's capabilities without you having to write any more code. These portrayals can wrap around another Simple Portrayal to enhance it: and you can wrap a wrapper portrayal in another wrapper portrayal to stack on the enhancement fun. Here are the wrapper portrayals presently provided:

- sim.portrayal.simple.LabelledPortrayal2D adds a textual label to the portrayed object, and can be (typically) set up to only do this when the object is selected.
- sim.portrayal.simple.CircledPortrayal2D adds a highlighting circle around the portrayed object, and can be (typically) set up to only do this when the object is selected.
- sim.portrayal.simple.FacetedPortrayal2D chooses among several subsidiary SimplePortrayals to portray the object based on some current feature of the object, or to portray with all of them at once.
- sim.portrayal.simple.OrientedPortrayal2D Adds an orientation compass marker indicating the direction of the object.
- sim.portrayal.simple.TransformedPortrayal2D Modifies the size, orientation, or translation of the underlying portrayed object using a java.awt.geom.AffineTransform.
- sim.portrayal.simple.TrailedPortrayal2D Adds a physical trail behind the object, showing where it's been in the recent past. TrailedPortrayal can be (typically) set up to only do this when the object is selected.
- sim.portrayal.simple.MovablePortrayal2D Allows you to move the object by dragging it with the mouse.
- sim.portrayal.simple.AdjustablePortrayal2D Allows you to change the orientation of, or the scale of, the object by dragging a provided handle with the mouse. The handle only appears when the object is selected.

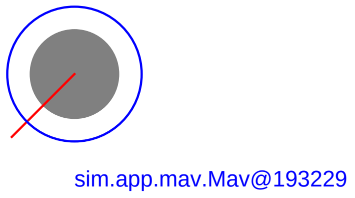
As you can see, the wrapper portrayal concept can do quite a lot of things.

Subsidiary Portrayals Wrapper portrayals all take one or more **subsidiary portrayals** (or **children**), usually in their constructors. When the wrapper portrayal is asked to draw, or hit-test, etc. an object, it typically (also) calls the equivalent method on its subsidiary, plus adding its own goodness. Furthermore, a wrapper portrayal can also take another wrapper portrayal, which takes a wrapper portrayal, which takes a basic portrayal (for example), thus forming a chain of wrapper portrayals.

What if your object portrays itself? No problem: just use null as the child, and the wrapper portrayal will assume the object itself is a SimplePortrayal and treat it, effectively, as the child.

9.3.4.1 Labeling, Hilighting, and Showing Orientation

MASON provides three default wrapper portrayals for adding more visual information to an existing SimplePortrayal: sim.portrayal.simple.LabelledPortrayal2D adds a textual label, sim.portrayal.simple.CircledPortrayal2D highlights an object by drawing a circle around it, and sim.portrayal.simple.OrientedPortrayal2D adds a compass orientation marker. The figure at right shows all three in action at once, wrapping a simple, gray OvalPortrayal2D. (The label wasn't particularly well chosen).



Labelling sim.portrayal.simple.LabelledPortrayal2D Adds an optional label to a portrayed object. The origin of the label is directly below the object by a small amount. There are a lot of options. First, you you may specify the **paint** and **font** of the label, the **text alignment** (left, right, center) of the label, and whether or not it's only **displayed when the object is selected**. You an indicate whether the size of the font **changes when you zoom in**.

And you can specify the location of the label in two ways. First, you can state where the origin of the label is to be placed in model units. For example, assuming the object is typically to be drawn within a 1×1 rectangle in model units, you can specify that the label is to be drawn directly to the right of the object by placing the location at $\langle 1.5, 0\rangle$. As the user zooms in in the Display2D, the label changes its distance proportionally. Second, you can add an additional **offset**, in pixels, to the label location. The default label location is ten pixels down from right at the expected lower edge of the object, horizontally dead center:

```
public static final double DEFAULT_SCALE_X = 0;
public static final double DEFAULT_SCALE_Y = 0.5;
public static final double DEFAULT_OFFSET_X = 0;
public static final double DEFAULT_OFFSET_Y = 10;
```

The baseline (the bottom of the letters) of the text is drawn relative to this point. You can further specify whether the text is to be left-aligned (the default), right-aligned, or centered with regard to the point. The constants are:

```
public static final int ALIGN_CENTER;
public static final int ALIGN_LEFT;
public static final int ALIGN_RIGHT;
```

The constructors should now be clear:

sim.portrayal.simple.LabelledPortrayal2D Constructor Methods

public LabelledPortrayal2D(SimplePortrayal2D child, String label)
Creates a LabelledPortrayal2D which always draws its label at the default location, color and font, and left-alignment. If the child is null, then LabelledPortrayal2D assumes the object can portray itself and uses it as the (effective) child.

LabelledPortrayal2D(SimplePortrayal2D child, String label, Paint paint, boolean onlyLabelWhenSelected)
Creates a LabelledPortrayal2D with default location values, left-aligned, with the given paint and label, and whether or not to only draw the label when the object has been selected. If the child is null, then LabelledPortrayal2D assumes the object can portray itself and uses it as the (effective) child.

public LabelledPortrayal2D(SimplePortrayal2D child, double scaley, String label, Paint paint, boolean onlyLabelWhenSelected)

Creates a LabelledPortrayal2D with the given child, Y scale value to specify the location of the label (plus a 10 pixel y offset), left-aligned, and providing the text paint, label, and whether or not to only draw the label when the object has been selected. If the child is null, then LabelledPortrayal2D assumes the object can portray itself and uses it as the (effective) child.

public LabelledPortrayal2D(SimplePortrayal2D child, int offsetx, int offsety, double scalex, double scaley, Font font, int align, String label, Paint paint, boolean onlyLabelWhenSelected)
Creates a LabelledPortrayal2D with the given child, using the offset and scale values to specify the location of the label, and providing the font, text alignment (one of ALIGN_CENTER, ALIGN_LEFT, or ALIGN_RIGHT), text paint, label, and whether or not to only draw the label when the object has been selected. If the child is null, then LabelledPortrayal2D assumes the object can portray itself and uses it as the (effective) child.

How does LabelledPortrayal determine what the label is? By calling the getLabel(...) method, which you can override to provide more functionality. By default, the method works like this: f you provided a label, it'll use that. Otherwise, if the object is null, it'll say "null". Otherwise it'll call the toString() method on the object.

Label scaling works much like the scaling in EdgePortrayal2D. By default, the font never scales as you zoom in. Alternatively you can have it always scale in size, much like looking closer and closer at a map. Alternatively you can have it only get smaller when you zoom far away, but not get bigger when you're getting very close. The constants are:

```
public static final int NEVER_SCALE;
public static final int SCALE_WHEN_SMALLER;
public static final int ALWAYS_SCALE;
```

Here are LabelledPortrayal's (few) methods of interest:

sim.portrayal.simple.LabelledPortrayal2D Methods

public SimplePortrayal2D getChild(Object obj)
Returns the subsidiary portrayal. If the object can portray itself, it is returned as the portrayal.

public String getLabel(Object object, DrawInfo2D info)
Returns the label string to draw.

public boolean isLabelShowing()
Returns whether LabelledPortayal2D will show its label. If false, then LabelledPortrayal2D will not show its label no matter what its settings are regarding object selection. If true, then LabelledPortrayal2D may still not show its label depending on object selection settings.

public void setLabelShowing(boolean val)
Sets whether LabelledPortayal2D will show its label. If false, then LabelledPortrayal2D will not show its label no matter what its settings are regarding object selection. If true, then LabelledPortrayal2D may still not show its label depending on object selection settings.

public boolean getOnlyLabelWhenSelected()
Returns whether the label will be shown only if the object is selected.

public void setOnlyLabelWhenSelected(boolean val)
Sets whether the label will be shown only if the object is selected.

public int getLabelScaling()
Returns the label scaling setting, either NEVER_SCALE, SCALE_WHEN_SMALLER, or ALWAYS_SCALE.

public void setLabelScaling(int val)
Sets the label scaling setting, either NEVER_SCALE, SCALE_WHEN_SMALLER, or ALWAYS_SCALE.

Highlighting MASON also provides a class for doing simple highlighting as well: sim.portrayal.simple.CircledPortrayal2D. This class adds a simple circular ring around the object. It works in a similar fashion to the LabelledPortrayal2D: you can have it add the ring only when selected, and there is an option to switch it off regardless. The ring has similar positioning information too: you can specify a **scale** defining the radius of the ring in model coordinates, and a further **offset** to increase the radius by some number of pixels. The default values are:

```
public static final double DEFAULT_SCALE = 2.0;
public static final double DEFAULT_OFFSET = 0.0;
```

The constructors should be straightforward:

sim.portrayal.simple.CircledPortrayal2D Constructor Methods

public CircledPortrayal2D(SimplePortrayal2D child, int offset, double scale, Paint paint, boolean onlyCircleWhenSelected)

Creates a CircledPortrayal2D with the given child, and the size of the ring in both pixel offset and model scale. The ring is drawn with the given paint, and you can specify if it is to only be drawn when the object is selected. If the child is null, then CircledPortrayal2D assumes the object can portray itself and uses it as the (effective) child.

public CircledPortrayal2D(SimplePortrayal2D child, Paint paint, boolean onlyCircleWhenSelected)
Creates a CircledPortrayal2D with the given child, and default size settings. The ring is drawn with the given paint, and you can specify if it is to only be drawn when the object is selected. If the child is null, then CircledPortrayal2D assumes the object can portray itself and uses it as the (effective) child.

public CircledPortrayal2D(SimplePortrayal2D child)
Creates a CircledPortrayal2D with the given child, and default settings for the ring size and paint (blue). The ring is always shown. If the child is null, then CircledPortrayal2D assumes the object can portray itself and uses it as the (effective) child.

Methods of interest are likewise similar to LabelledPortrayal2D:

sim.portrayal.simple.CircledPortrayal2D Methods

public SimplePortrayal2D getChild(Object obj)
Returns the subsidiary portrayal. If the object can portray itself, it is returned as the portrayal.

public boolean isCircleShowing()
Returns whether CircledPortrayal2D will show its circle. If false, then CircledPortrayal2D will not show its circle no matter what its settings are regarding object selection. If true, then CircledPortrayal2D may still not show its circle depending on object selection settings.

public void setCircleShowing(boolean val)
Sets whether portrayal will show its circle. If false, then CircledPortrayal2D will not show its circle no matter what its settings are regarding object selection. If true, then CircledPortrayal2D may still not show its circle depending on object selection settings.

public boolean getOnlyCircleWhenSelected()
Returns whether the circle will be shown only if the object is selected..

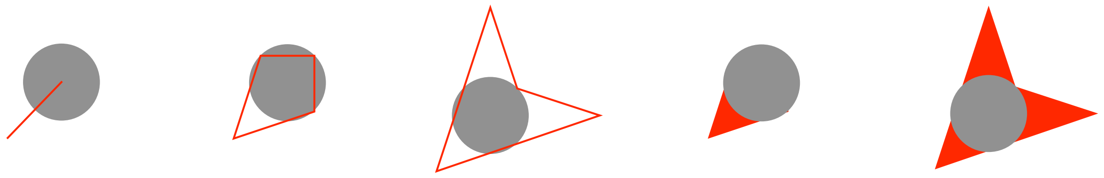


Figure 9.4 Five orientation markers for sim.portrayal.simple.OrientedPortrayal2D.

public void setOnlyCircleWhenSelected(boolean val)
Sets whether the circle will be shown only if the object is selected..

Showing Orientation Last but not least, MASON can add an **orientation marker** to your SimplePortrayal, using a wrapper portrayal called sim.portrayal.simple.OrientedPortrayal2D.

In order to display an orientation, OrientedPortrayal2D must know what the orientation *is*. To do this, it expects that the underlying object in the model implements the sim.portrayal.Oriented2D interface. This interface defines a single method:

sim.portrayal.Oriented2D Methods

public double orientation2D()
Returns the current orientation of the object, in radians.

If the object implements this method, then OrientedPortrayal2D queries it to determine how to specify the orientation of the object. If not, then the orientation is assumed to be 0.0.

Why isn't this method a proper Java Bean Property, that is, **get***Orientation2D()?*

Because MASON doesn't want to require the simulation designer to have this orientation appear in an Inspector when the object is inspected. You can always create a property called getOrientation2D() which calls orientation2D(). We won't claim this is a *good* reason: but it *is* a reason, so there.

Oriented2D then allows the object to be drawn using its subsidiary Simple Portrayal, and adds an orientation marker either above or below the portrayal's depiction. The five options at present are shown in Figure 9.4. When is the marker drawn on top? When it's either a single line or is an unfilled outline of a shape. If it's a filled shape, it's drawn underneath.

You specify the shape to draw in two ways. First, you specify the shape itself, and second, you specify if the shape should be drawn filled (the default). Lines obviously cannot be drawn filled regardless. The available shapes, as shown in Figure 9.4, are:

```
public static final int SHAPE_LINE;
public static final int SHAPE_KITE;
public static final int SHAPE_COMPASS;
```

The default is SHAPE_LINE.

You can also specify the *scale* of the orientation marker: how big it is relative to the underlying SimplePortrayal. This is done in exactly the same way as in CircledPortrayal2D: you provide a **scale** in the underlying model units, and an **offset** in pixels. The default settings for these are:

```
public static final double DEFAULT_SCALE = 0.5;
public static final int DEFAULT_OFFSET = 0;
```

Haven't I seen the Compass shape before?

Yep, it's the shape used for the Flockers demo example. That example uses a trick: the shape is all you see. This is done by wrapping a sim.portrayal.simple.SimplePortrayal, which if you recall doesn't draw anything. However it also doesn't respond to hit requests, so how is that done? It turns out that OrientedPortrayal2D responds to hit requests if the hit lands on the orientation marker shape. Since the shape is all there is, we're good.

Just like CircledPortrayal2D you can optionally have OrientedPortrayal2D only show its orientation shape if the object is selected, or force it off entirely.

One thing to be aware of is that Oriented2D responds to hit-testing on its orientation marker shape in addition to sending hit-testing results to its subsidiary SimplePortrayal. This can be turned off with the method setOrientationHittable().

OrientedPortrayal2D's constructors:

sim.portrayal.simple.OrientedPortrayal2D Constructor Methods

public OrientedPortrayal2D(SimplePortrayal2D child, int offset, double scale, Paint paint, int shape)
: Creates a OrientedPortrayal2D with the given child, shape, shape paint, and the size of the shape in both pixel offset and model scale. If the child is null, then OrientedPortrayal2D assumes the object can portray itself and uses it as the (effective) child.

public OrientedPortrayal2D(SimplePortrayal2D child, int offset, double scale, Paint paint)
: Creates a OrientedPortrayal2D with the given child, shape paint, and the size of the shape in both pixel offset and model scale. The shape is a line. If the child is null, then OrientedPortrayal2D assumes the object can portray itself and uses it as the (effective) child.

public OrientedPortrayal2D(SimplePortrayal2D child, int offset, double scale)
: Creates a OrientedPortrayal2D with the given child, and the size of the shape in both pixel offset and model scale. The shape is a line, and drawn in red. If the child is null, then OrientedPortrayal2D assumes the object can portray itself and uses it as the (effective) child.

public OrientedPortrayal2D(SimplePortrayal2D child, Paint paint)
: Creates a OrientedPortrayal2D with the given child and shape paint. The shape is a line, and drawn with the default scale and offset. If the child is null, then OrientedPortrayal2D assumes the object can portray itself and uses it as the (effective) child.

public OrientedPortrayal2D(SimplePortrayal2D child)
: Creates a OrientedPortrayal2D with the given child. The shape is a line, and drawn with the default scale and offset, and in red. If the child is null, then OrientedPortrayal2D assumes the object can portray itself and uses it as the (effective) child.

OrientedPortrayal2D has certain important methods needed to set features:

sim.portrayal.simple.OrientedPortrayal2D Methods

public SimplePortrayal2D getChild(Object obj)
: Returns the subsidiary portrayal. If the object can portray itself, it is returned as the portrayal.

public int getShape()
: Returns the shape being drawn. The default is SHAPE_LINE.

public void setShape(int shape)
: Sets the shape being drawn. The default is SHAPE_LINE.

public boolean isDrawFilled()
: Returns whether OrientedPortrayal2D will fill (versus draw the outline of) its orientation marker shape. Lines are never filled regardless of the setting of this property. The default is true (shapes are drawn filled).

public void setDrawFilled(boolean val)
: Sets whether OrientedPortrayal2D will fill (versus draw the outline of) its orientation marker shape. Lines are never filled regardless of the setting of this property. The default is true (shapes are drawn filled).

public boolean isOrientationShowing()
  Returns whether OrientedPortrayal2D will show its orientation marker shape. If false, then OrientedPortrayal2D will not show its shape no matter what its settings are regarding object selection. If true, then OrientedPortrayal2D may still not show its shape depending on object selection settings.

public void setOrientationShowing(boolean val)
  Sets whether portrayal will show its orientation marker shape. If false, then OrientedPortrayal2D will not show its shape no matter what its settings are regarding object selection. If true, then OrientedPortrayal2D may still not show its shape depending on object selection settings.

public boolean getOnlyDrawWhenSelected()
  Returns whether the orientation marker shape will be shown only if the object is selected.

public void setOnlyDrawWhenSelected(boolean val)
  Sets whether the orientation marker shape will be shown only if the object is selected.

public boolean isOrientationHittable()
  Returns whether the orientation marker shape will respond to hit testing in addition to the underlying portrayal.

public void setOrientationHittable(boolean val)
  Sets whether the orientation marker shape will respond to hit testing in addition to the underlying portrayal.

9.3.4.2 Using Multiple SimplePortrayal2Ds

sim.portrayal.simple.FacetedPortrayal2D takes some N subsidiary SimplePortrayals. Depending on the current value of the underlying object, it picks from among these SimplePortrayals to portray the object. Optionally, FacetedPortrayal2D can call upon *all* of the SimplePortrayals to portray the object in sequence.

FacetedPortrayal2D is different than other wrapper portrayals because it takes more than one SimplePortrayal. To create a FacetedPortrayal2D you need to provide all those children, plus whether you'd like to portray with *all* of them or with a particular one depending on the current object status.

sim.portrayal.simple.FacetedPortrayal2D Constructor Methods

public FacetedPortrayal2D(SimplePortrayal2D[] children)
  Creates a FacetedPortrayal2D with the given children, set up to select one at a time based on the current object value.

public FacetedPortrayal2D(SimplePortrayal2D[] children, boolean portrayAllChildren)
  Creates a FacetedPortrayal2D with the given children, and whether or not to portray all of them, or rather one selected at a time based on the current object value.

Portraying all SimplePortrayal2Ds at Once Why would you want to do this? Mostly to concatenate two shapes together in a simple way. For example: to portray an object as a circle plus a rectangle. I agree though, it's not going to be used much. Which leads us to...

Selecting a SimplePortrayal2D based on Value In this configuration, FacetedPortrayal2D picks the SimplePortrayal2D based on the object's numerical value. For this to work, the object must either be a Number or must be Valuable. Furthermore, the doubleValue() of the object must be an integer ≥ 0 and less than the number of subsidiary SimplePortrayals. This value will determine which SimplePortrayal is picked. If you find this too restrictive, you can instead override the getChildIndex(...) to return the index of the child to use.

You can use this, for example, to display different images depending on an object's value. In this case, create multiple ImagePortrayal2Ds, one for each image of interest, and attach them all to the FacetedPortrayal2D. Based on the object's value, different ImagePortrayal2Ds will be used.

Or you could use this to make a SimplePortrayal2D appear and disappear. Here, you attach your SimplePortrayal2D of choice, plus an unsubclassed instance of sim.portrayal.SimplePortrayal2D (which doesn't draw anything).

sim.portrayal.simple.FacetedPortrayal2D Methods

public int getChildIndex(Object object, int numIndices)
Returns the child index to use based on the given object. The index must be ≥ 0 but less than *numIndices*.

9.3.4.3 Applying a Geometric Transformation

This class doesn't feel fully baked

It's not.

sim.portrayal.simple.TransformedPortrayal2D is pretty straightforward: it allows you to rotate, translate, scale, shear, or otherwise transform the way the subsidiary portrayal is drawn using a java.awt.geom.AffineTransform. But there's a tradeoff: you're not allowed to perform hit-testing on the object, thus no selection or inspection.

A note about AffineTransforms: they change *everything*: the text, the thickness of the lines, the scale of the patterns in the paint being used. This class is rarely used. I'd instead just build a new SimplePortrayal2D.

At any rate, the constructors:

sim.portrayal.simple.TransformedPortrayal2D Constructor Methods

public TransformedPortrayal2D(SimplePortrayal2D child, AffineTransform transform)
Creates a TransformedPortrayal2D with the given child and affine transform.

And the methods of interest:

sim.portrayal.simple.TransformedPortrayal2D Methods

public SimplePortrayal2D getChild(Object obj)
Returns the subsidiary portrayal. If the object can portray itself, it is returned as the portrayal.

public boolean hitObject(Object object, DrawInfo2D range)
Always returns false.

9.3.4.4 Allowing the User to Move, Rotate, or Scale the Object

The wrapper portrayals sim.portrayal.simple.MovablePortrayal2D and sim.portrayal.simple.AdjustablePortrayal2D make it easy to move and make simple adjustments (rotation, size) to an underlying object in a field.

Let's start with MovablePortrayal2D. If you wrap a SimplePortrayal2D with this object, the user will be able to move the object by dragging it with the mouse. When you move an object, you also select it (and thus deselect other objects). If you use a MovablePortrayal2D in conjunction with an AdjustablePortrayal2D (discussed next), you must wrap the MovablePortrayal2D inside the AdjustablePortrayal2D, not the other way around. MovablePortrayal takes a single, simple Constructor:

sim.portrayal.simple.MovablePortrayal2D Constructor Methods

public MovablePortrayal2D(SimplePortrayal2D child)
Creates a MovablePortrayal2D with the given child.

How do you *prevent* objects from being moved? Ordinarily, without a MovablePortrayal2D the user can't move an object by dragging it. But consider the following example. All the objects in your field, even though they're different classes, use the same SimplePortrayal2D. You have wrapped this in a MovablePortrayal2D and installed the MovablePortrayal2D with setPortrayalForAll(). So now all of your objects are movable. But what if you didn't want some of them movable?

Of course, you could break your SimplePortrayal2Ds out by object or by class, some with MovablePortrayal2D and some without. This might be irritating though. But there's an alternative. You can make your objects sim.portrayal.Fixed2D. This interface declares a single method which controls whether the object may be moved by MovablePortrayal2D. You can also use this interface to keep track of where your object is being moved as the user drags it about, and to adjust the user's desired drag location (for example, to constrain the object to only be moved along a line). Fixed2D defines a single method:

sim.portrayal.Fixed2D Methods

public boolean maySetLocation(Object field, Object newObjectLocation)
Given the field and the proposed new location of the object in the field, dictates how a MovablePortrayal2D interacts with this object. If you don't want this object to be movable by a MovablePortrayal2D, simply return false. If you are fine with being moved, return true. If you would simply like to be informed of where you are being moved — for example, to update internal belief about your location — update this information using *newObjectLocation* and return true. If you would like to modify the proposed location, move the object yourself to the modified location in the field, then return false.

The sim.portrayal.simple.AdjustablePortayal2D wrapper portrayal gives the user an easy way to rotate and/or scale your object. You're responsible for defining what "rotation" and "scaling" mean: for example, you could use these hooks for something else, like changing the age and temperature of your object. But usually they're used for exactly what you'd expect.

Like MovablePortrayal2D, AdjustablePortrayal2D has a single straightforward constructor:

sim.portrayal.simple.AdjustablePortrayal2D Constructor Methods

public AdjustablePortrayal2D(SimplePortrayal2D child)
Creates a AdjustablePortrayal2D with the given child.

If you wrap a SimplePortrayal2D with an AdustablePortrayal2D, it'll add a ring-and-knob widget whenever the object is selected (see Figure 9.5 below). If the user rotates the knob on the ring, the object will be asked to reorient itself. If the user drags the knob *away* from the ring (the knob doesn't presently move off the ring, but MASON will understand) the object will be asked to rescale itself.

AdjustablePortrayal2D only bothers to ask objects to orient themselves if they implement the sim.portrayal.Orientable2D interface, which is an extension of the sim.portrayal.Oriented2D, an interface defined earlier in Section 9.3.4.1. The Orientable2D interface defines two methods:

Oriented2D doesn't define a proper Java Property!

We don't want to require the user to display orientation information in inspectors. If you'd like to do so, just implement an additional method, public double getOrientation2D(), which simply calls orientation2D().

sim.portrayal.Orientable2D Methods

public double orientation2D()
Returns the object's orientation.

public void setOrientation2D(double val)
Sets the object's orientation.

It's up to you to implement these methods in your object to reorient itself or provide orientation information (or not). You also have the option of just implementing the sim.portrayal.Oriented2D interface, which simply returns the object's orientation. In this case, the ring will rotate helpfully to reflect your object's current orientation, but the user won't be able to reorient the object. This could be used instead of sim.portrayal.simple.OrientedPortrayal2D to indicate the current orientation.

Speaking of OrientedPortayal2D: AdjustablePortrayal2D doesn't change how the underlying SimplePortrayal2D draws its object. So if you adjust the object's rotation, you may want something to reflect this, either with OrientedPortayal2D or a custom SimplePortrayal2D.

Last but not least, the user can *scale* (notionally resize) an object by dragging the knob. To do this, your object must implement the sim.portrayal.Scalable2D interface. This interface defines two methods which form a Java Bean Property:

Wait, why does Scalable2D form a property but not Orientable2D?

History. Look, I didn't say Orientatable2D's reason for not being a property was *good* reason...

sim.portrayal.Scalable2D Methods

public double getScale2D()
Returns the object's scale.

public void setScale2D(double val)
Sets the object's scale.

It's up to you to define how your object looks when it's rescaled.

A final note about interaction between AdjustablePortrayal2D and MovablePortrayal2D (discussed earlier). If you use a MovablePortrayal2D in conjunction with an AdjustablePortrayal2D, you must wrap the MovablePortrayal2D inside the AdjustablePortrayal2D, not the other way around.

9.3.4.5 Adding a Trail

It's often nice to display a **trail** dragging behind an object to show where it's been in the recent past. MASON provides a wrapper portrayal, sim.portrayal.simple.TrailedPortrayal2D, which provides exactly this.

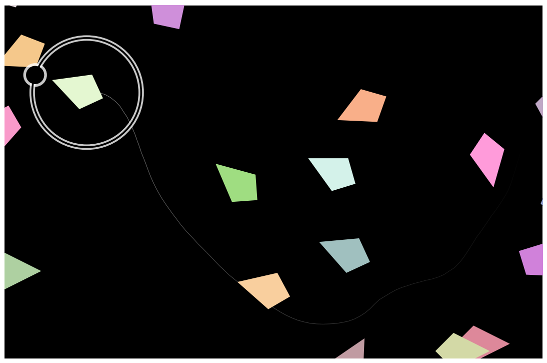


Figure 9.5 A flocker with a trail and an adjustment ring.

TrailedPortrayal2D generally requires not one but **two** FieldPortrayals: one to draw the object itself (via its subsidiary SimplePortrayal), and one to draw its trail. You provide the "trail" FieldPortrayal, and the subsidiary, in the TrailPortrayal2D's constructor. Then you add the TrailedPortrayal2D to *both* FieldPortrayals.

You should probably attach the "trail" FieldPortrayal to the Display2D before you attach the "object" FieldPortrayal: this way, the trail is drawn behind the object. The TrailedPortrayal2D won't be drawn in both FieldPortrayals: it'll recognize the "trail" FieldPortrayal and only draw itself there. In the other ("object") FieldPortrayal, it'll ask its subsidiary to draw.

You can further wrap the TrailedPortrayal (in a MovablePortrayal2D, for example) to move the object, but only add the MovablePortrayal to the "object" FieldPortrayal. Add the TrailedPortrayal2D to the "trail" FieldPortrayal.

TrailedPortrayal2D is complicated. Any examples?

Check out sim.app.flockers.FlockersWithUI.

Compatible Field Portrayals In theory TrailedPortrayal2D can be used with any continuous or Grid2D Field Portrayal. But in reality it works best only with field portrayals for fields in which an object may not appear at multiple locations simultaneously, and which have implemented the methods getObject-Location(...) and getRelativeObjectPosition(...). This means that realistically you should use it only with sim.portrayal.continuous.ContinuousPortrayal2D and sim.portrayal.grid.SparseGridPortrayal2D.

Memory Tradeoffs TrailedPortrayal2D needs to store the trail it's building for its underlying object. This has some implications if you've got more than one object in the field being represented by this class: which one will have the trail stored? Will all objects be displaying their trails, or only those selected? Will objects be building trails even when not displayed? And so on.

You have several options, in increasing memory usage:

- If you don't mind if **only one** object will draw a trail at a time, chosen from among the objects currently **selected**, and that the trail will be **grown only once the object is selected**, then you can get away with using a single TrailedPortrayal2D for your two FieldPortrayals: for example you can set it as their portrayalForAll(). In this case you must also call setOnlyGrowTrailWhenSelected(true) (otherwise the behavior will be undefined).
- If you want **every selected object** to draw its trail, and don't mind that the trail will be **grown only once the object is selected**, then you'll need to assign a unique TrailedPortrayal2D for every such object (that is, you'll have to use something like portrayalForObject()). You fortunately still can call setOnlyGrowTrailWhenSelected(true), which will save some memory.
- If you want **every selected object** to draw its trail, and each object must **retain knowledge of its trail** even when it's not being displayed, then you'll need to assign a unique TrailedPortrayal2D for every such object (that is, you'll have to use something like portrayalForObject()).
- If you want **every object to draw its trail at all times**, then you'll need to assign a unique TrailedPortrayal2D for every such object (that is, you'll have to use something like portrayalForObject()). You will have to call setOnlyShowTrailWhenSelected(false), which will potentially be *very* slow.

Note that this brings up two important properties: onlyGrowTrailWhenSelected (default false), which governs whether an object grows trails at all times, even when it's not being displayed, and onlyShowTrail-WhenSelected (default true), which governs whether an object only shows trails when it's currently selected. Furthermore, whether or not the TrailedPortrayal2D is assigned uniquely to a given object, or is being shared for multiple objects, will dictate certain options and the value of these two properties.

Lengths and Jumps Trails also have a *length*, measured in **model time**. Trail segments created earlier in the past than this length are automatically deleted. There is no default setting: the length is always specified in the constructor.

And trails have a concept of **maximum jump**, meant to make them look better in toroidal environments. The problem is that if an object disappears off the left side of the screen (say) and reappears on the right side, the trail thinks the object has zoomed across the screen and will draw a big trail line across the screen. This isn't what you wanted probably. So you can specify the largest expanse (as a percentage of the field width or height) allowed before a trail segment will not be displayed. The default setting is:

```
public static final double DEFAULT_MAXIMUM_JUMP = 0.75;
```

... That is, a trail segment will be shown unless its width is over 0.75 of the field width, or its height is over 0.75 of the field height. You'd have to have a pretty fast-moving object to violate this constraint.

Making a Custom Trail A trail is made up of **segments**, each segment representing a movement of the object from one previous location to another during a single timestep. You can provide a SimplePortrayal2D which draws these segments. TrailedPortrayal2D will call your SimplePortrayal2D's draw(...) method, passing in a special DrawInfo2D called sim.portrayal.simple.TrailedPortrayal2D.TrailDrawInfo2D. This class adds a variable called value which indicates where along the trail (in time) this segment is. If the segment is at the very beginning of the trail (that is, next to the object), the value will be 0.0. If it's at the very end of the trail, the value will be 1.0:

```
public double value; // this is in sim.portrayal.simple.TrailedPortrayal2D.TrailDrawInfo2D
```

TrailDrawInfo2D is a subclass of sim.portrayal.network.EdgePortrayal2D, meaning that you have access not to just one point but two points: these represent the start and end of your segment. So if you want to draw a line, you can get the points to draw just as you access the two points from EdgeDrawInfo2D. Alternatively you could just draw a single object (creating a sort of "dotted" trail), in which case you'd just draw at the draw origin like any SimplePortayal2D. Try using sim.portrayal.simple.OvalPortrayal2D as your trail portrayal some time.

Armed with this knowledge, you should now be able to understand the TrailedPortrayal2D constructors:

sim.portrayal.simple.TrailedPortrayal2D Constructor Methods

public TrailedPortrayal2D(GUIState state, SimplePortrayal2D child, FieldPortrayal2D fieldPortrayal, double length, Color minColor, Color maxColor)
Creates a TrailedPortrayal2D with the given child and "trail" Field Portrayal. The length of the trail is provided, as well as the min color (at the "start" of the trail, near the object) and the max color (at the far end of the trail). TrailedPortrayal2D will draw trail segments using lines.

public TrailedPortrayal2D(GUIState state, SimplePortrayal2D child, FieldPortrayal2D fieldPortrayal, double length)
Creates a TrailedPortrayal2D with the given child and "trail" Field Portrayal. The length of the trail is provided. The min color (at the "start" of the trail, near the object) is set to DEFAULT_MIN_COLOR (opaque gray), and the max color (at the far end of the trail) is set to DEFAULT_MAX_COLOR (fully transparent gray). TrailedPortrayal2D will draw trail segments using lines.

public TrailedPortrayal2D(GUIState state, SimplePortrayal2D child, FieldPortrayal2D fieldPortrayal, SimplePortrayal2D trail, double length)
Creates a TrailedPortrayal2D with the given child and "trail" Field Portrayal. The length of the trail is provided, as well as a custom SimplePortrayal2D responsible for actually drawing each trail segment.

The default min and max color are defined as:

```
public static final Color DEFAULT_MIN_COLOR = new Color(128,128,128,255); // opaque gray
public static final Color DEFAULT_MAX_COLOR = new Color(128,128,128,0);   // transparent
```

And the relevant methods:

sim.portrayal.simple.TrailedPortrayal2D Methods

public void setOnlyGrowTrailWhenSelected(boolean val)
Sets whether or not to begin growing the trail only when the object has been selected. The default value is FALSE.

public boolean getOnlyGrowTrailWhenSelected()
Returns whether or not to begin growing the trail only when the object has been selected. The default value is FALSE.

setOnlyShowTrailWhenSelected(boolean val)
Sets whether or not to show the trail only when the object has been selected. The default value is TRUE.

public boolean getOnlyShowTrailWhenSelected()
Returns whether or not to show the trail only when the object has been selected. The default value is TRUE.

public void setLength(double val)
Sets the length of the trail (in model time).

public double getLength()
: Returns the length of the trail (in model time).

public void setMaximumJump(double val)
: Sets the maximum jump of intervals of the trail, as a percentage of field width and field height. The default value is 0.75.

public double getLength()
: Returns the maximum jump of intervals of the trail, as a percentage of field width and field height. The default value is 0.75.

9.3.5 Objects Acting As Their Own Simple Portrayals

Objects in fields can also act as their own Simple Portrayals. It's pretty straightforward: you just have the object subclass the sim.portrayal.SimplePortrayal2D class and implement the methods themselves. None of the methods is *required*, though you'll probably want to at least implement the draw(...) method. Then you just don't bother registering a SimplePortrayal2D for the object. See the next section for hints on how to implement these methods.

If your object subclasses SimplePortrayal2D, you can still use wrapper portrayals even though there's no underlying SimplePortrayal2D registered for the object. Just pass in null as the chid to the wrapper portrayal, and it'll use the object itself as the child.

9.3.6 Implementing a Simple Portrayal

So you want to make our own custom SimplePortrayal. Just like a 2D Field Portrayal, a 2D Simple Portrayal handles several tasks:

- Drawing its object
- Doing hit-testing on the object
- Selecting or deselecting the object
- Handling mouse events on the object
- Providing Inspectors for the object
- Returning the object's "status" (a short textual description of the object).
- Returning the object's name

The only one you really have to implement is drawing: though it's not hard to implement the others. Let's get right to it.

9.3.6.1 Drawing

Drawing is expensive, and so Field Portrayals try hard to avoid drawing objects unless they must. But when a Field Portrayal determines that it is *likely* that a given object lies within the Display2D's clip region and ought to be drawn, it calls for the the Simple Portrayal and calls the following method on it:

sim.portrayal.SimplePortrayal2D Methods

public void draw(Object object, Graphics2D graphics, DrawInfo2D info)
: Draws the Object on-screen. The object should be centered at ⟨info.draw.x, info.draw.y⟩. One unit of width in the model's world is equivalent to info.draw.width pixels, and likewise one unit of height in the model's world is equivalent to info.draw.height pixels. info.clip provides the clip rectangle, in pixels, of objects which must be drawn: if the object does not fall within this rectangle, it need not be drawn. If info.precise is true, then the object should be drawn using high-quality floating-point operations; else (potentially faster) integer operations will suffice. If info.selected is true, the object should assume it has been "selected" and draw itself differently accordingly if it feels the need. object provides the object in question: and info.fieldPortrayal provides the field portrayal. In some cases, info.location may provide the location of the object in the field, though this is optional and should not be relied on.

What are all these draw and clip rectangles?

For a refresher, see Section 9.2.2.

MASON assumes that most objects are drawn roughly as 1×1 in model coordinates: thus when an object draws itself on-screen, it'll usually be drawn to approximately fill the rectangle from ⟨info.draw.x, info.draw.y⟩ to ⟨info.draw.x + info.draw.width, info.draw.y + info.draw.height⟩. If this rectangle doesn't intersect with the clip rectangle (in info.clip), then the Field Portrayal will likely not even bother to ask the SimplePortrayal2D to draw itself.

If drawing is really expensive, and the object's shape is unusual, SimplePortrayal2D can do more sophisticated clip testing once its draw(...) method is called to determine if the object is *really* intersecting with the info.clip rectangle. This is rare though.

When you're asked to draw, you're given a java.awt.Graphics2D class to draw with. Most MASON examples draw with Java AWT (integer) graphics primitives when they can, and Java2D (floating point) graphics primitives when they must. This is because in most cases AWT is still faster than Java2D in many implementations. This is changing though: it might be enough at this point for you to just always draw with Java2D

When *must* you draw with Java2D? When the DrawInfo2D's precise variable is set to true. This is often the case if MASON is asking you to draw not to the screen but in fact to a high-quality vector PDF which requires floating point accuracy.

Example Let's say we want to draw our object as a red stretched-out oval. We might implement the method as:

```
Ellipse2D.Double ellipse = new Ellipse2D.Double();

public void draw(Object object, Graphics2D graphics, DrawInfo2D info) {
    graphics.setColor(Color.RED);
    if (info.precise) {
        graphics.fillOval((int)(info.draw.x - info.draw.width / 2), (int)(info.draw.y - info.draw.height / 4),
                          (int)(info.draw.width), (int)(info.draw.height / 2));
    }
    else {
        ellipse.setFrame(info.draw.x - info.draw.width / 2.0, info.draw.y - info.draw.height / 4.0,
                         info.draw.width, info.draw.height / 2.0);
        graphics.fill(ellipse);
    }
}
```

9.3.6.2 Hit Testing

Hit testing is *very* similar to drawing. Instead of drawing a shape, we'll create the shape, then do an intersection test on it. The method is:

sim.portrayal.SimplePortrayal2D Methods

---

public boolean hitObject(Object object, DrawInfo2D range)
Returns true if the object intersected with the clip rectangle in the DrawInfo2D. The object is assumed to have be located on-screen centered at the origin of the DrawInfo2D's draw rectangle, and with a width and height specified by the DrawInfo2D's draw rectangle. It is possible in certain rare situations that the object may be null. Even if the object intersects with the clip rectangle, you may still return false if you don't wish the object to be hit (normally for selection, inspection, adjustment, or moving with the mouse). The default implementation simply returns false.

You'll generally find it more helpful to use Java2D rather than Java AWT graphics to do your hit testing.
It's also often helpful to cut the user some slack when your objects are small (zoomed out, say). So provide some slop in the hit testing, as shown below.

Example As you can see, very similar. Continuing our previous example, it's quite straightforward:

```
Ellipse2D.Double ellipse = new Ellipse2D.Double();        // we previously defined this in draw(...)

public boolean hitObject(Object object, DrawInfo2D range) {
    final double SLOP = 1.0;  // need a little extra area to hit objects
    ellipse.setFrame(info.draw.x - info.draw.width / 2.0 - SLOP, info.draw.y - info.draw.height / 4.0 - SLOP,
        info.draw.width + SLOP * 2, info.draw.height / 2.0 + SLOP * 2);
    return (ellipse.intersects(range.clip.x, range.clip.y, range.clip.width, range.clip.height));
}
```

9.3.6.3 Selecting an Object

MASON indicates that an object has been selected in two ways. First, it calls the setSelected() prior to drawing the object, indicating whether or not the object has been selected. If the SimplePortrayal2D does *not* want to be selected, it can return false at this point (you can't ask to be selected — you can only refuse if you like). For example, you could say:

```
public boolean setSelected(LocationWrapper wrapper, boolean selected) {
        return false;   // I don't ever want to be selected
        }
```

Then (depending on what was returned by this method), the variable selected is set to true or false in the DrawInfo2D passed into drawing. This indicates whether or not the object is presently selected. You can use this to (for example) change how your object looks:

```
Ellipse2D.Double ellipse = new Ellipse2D.Double();

public void draw(Object object, Graphics2D graphics, DrawInfo2D info) {
    if (info.selected) graphics.setColor(Color.RED);
    else graphics.setColor(Color.BLUE);
    if (info.precise) {
        graphics.fillOval((int)(info.draw.x - info.draw.width / 2), (int)(info.draw.y - info.draw.height / 4),
                          (int)(info.draw.width), (int)(info.draw.height / 2));
    }
    else {
        ellipse.setFrame(info.draw.x - info.draw.width / 2.0, info.draw.y - info.draw.height / 4.0,
                         info.draw.width, info.draw.height / 2.0);
        graphics.fill(ellipse);
    }
}
```

Or could use selection to inform the object in your model of its new "selected" status:

```
Ellipse2D.Double ellipse = new Ellipse2D.Double();

public void draw(Object object, Graphics2D graphics, DrawInfo2D info) {
    MyObject myobj = (MyObject)object;
    synchronized(info.gui.state.schedule) { myobj.iHaveBeenSelected(info.selected); }

    graphics.setColor(Color.RED);
    if (info.precise) {
        graphics.fillOval((int)(info.draw.x - info.draw.width / 2), (int)(info.draw.y - info.draw.height / 4),
                          (int)(info.draw.width), (int)(info.draw.height / 2));
    }
    else {
        ellipse.setFrame(info.draw.x - info.draw.width / 2.0, info.draw.y - info.draw.height / 4.0,
                         info.draw.width, info.draw.height / 2.0);
        graphics.fill(ellipse);
    }
}
```

Important Note on Synchronization Notice that we synchronized on the schedule before modifying the object. This is because the model thread could well be running and in charge of the object, and we don't want to create a race condition.

9.3.6.4 Getting the Object Status and Name

Displays and inspectors often want to display an object's name and a (very) short description of it. Often these are simply set to the same thing. The default implementations normally should suffice: the name just returns the toString() method applied to the given object, and the status just returns the name. But let's say you wanted to customize it in some insane way. You could say:

```
public String getStatus(LocationWrapper wrapper) { return "My object is always of HIGH status!"; }

public String getName(LocationWrapper wrapper)
    {
    if (wrapper == null) return "CRAZY NULL OBJECT";
    return "AWESOME OBJECT: " + wrapper.getObject();
    }
```

9.3.6.5 Customizing Mouse Events

It's rare to need to handle a mouse event on your own: this is the domain of classes like AdjustablePortrayal2D and MovablePortrayal2D. But if you like you can intercept mouse events on your object and do something special with them. The method for handling Mouse events is somewhat complex:

sim.portrayal.SimplePortrayal2D Methods

public boolean handleMouseEvent(GUIState gui, Manipulating2D manipulating, LocationWrapper wrapper, MouseEvent event, DrawInfo2D fieldPortrayalDrawInfo, int type)
Handles the given event, and either returns true (meaning that the event was handled and consumed) or false. Many mouse events are routable, a notable exception being scroll wheel events. Mouse events are sent at various times, indicated by *type*. If type is sim.portrayal.SimplePortraya2D.TYPE_SELECTED_OBJECT, then the mouse event is being called because it is a selected object, even if it's not being hit by the mouse. If the type is sim.portrayal.SimplePortraya2D.TYPE_HIT_OBJECT then the object is being sent a mouse event because it was hit by the mouse. The GUIState is provided, plus a Manipulating2D (likely a Display of some sort). The LocationWrapper provides the object in question, the field portrayal, and location of the object. The DrawInfo2D is the one provided to the field portrayal, not the simple portrayal: its draw field indicates the region of the field.

So let's say you'd like to have your SimplePortrayal2D print "ouch!" every time it's moused over. You could do it this way:

```
public boolean handleMouseEvent(GUIState gui, Manipulating2D manipulating, LocationWrapper wrapper,
                                MouseEvent event, DrawInfo2D fieldPortrayalDrawInfo, int type)
```

```
{
// need to lock on the model first!  We're accessing it potentially
// while the model is running underneath

synchronized(gui.state.schedule)
    {
    if (type == TYPE_HIT_OBJECT && event.getID() == event.MOUSE_MOVED)
        {
        System.err.println("ouch!");
        return true;
        }
    else return false;
    }
}
```

Note that you need to lock on the schedule: unlike various other operations, mouse events can and do occur right in the middle of the period of time that the model thread is in control of the model. If you don't lock on the schedule, you run the risk of a race condition as you access the model at the same time the model thread is messing around with it.

Certain SimplePortrayals2D wrapper classes which use the handleMouseEvent method also need to query the Display2D about certain things. This is done through the sim.display.Manipulating2D interface. At present this interface provides a single method:

sim.display.Manipulating2D Methods

public void performSelection(LocationWrapper wrapper)
Selects the Object represented by the given LocationWrapper by calling the relevant FieldPortrayal's setSelected(...) method

Chapter 10

Inspectors

An Inspector is a GUI widget that lets a user inspect, track, chart, and modify a specific model object, value, or the property of an object.[1] Inspectors are subclasses of the class sim.portrayal.Inspector, and with the exception of sim.portrayal.SimpleInspector, they're all found in the package sim.portrayal.inspector. Figure 10.1 shows a UML diagram of the primary Inspector classes.

Inspectors are generally *produced* by other Inspectors or widgets to inspect objects in the model as needed. Most inspectors are produced in one of the following five ways.

- When you double-click on an object, its Field Portrayal or Simple Portrayal may bring forth an inspector for that object or value.
- You can assign a single inspector for the model as a whole.
- You can assign inspectors for individual fields..
- Inspectors may produce other inspectors as you wander through them looking at aspects of the object or other objects it points to.
- Inspectors may contain within them other inspectors, much as a JPanel may have other JPanels within it.

There are two kinds of inspectors. Basic Inspectors inspect **objects** or **values**, often things found in a Field. The most common basic inspector is sim.portrayal.SimpleInspector. In contrast, **property inspectors** inspect not objects but Java Bean *properties* of objects. The difference is that an basic Inspector fixates on a specific object, but a Property Inspector's object or value can change based on the current setting of the property. Another difference is that basic Inspectors are usually provided programmatically for objects, but there can be many kinds of property inspectors (charts and graphs, data export, etc.), and they are loaded dynamically at run-time: they're basically plug-ins.

You can create your own Inspectors and it's perfectly fine to do so: after all, in its most basic form, an Inspector is little more than an ultimate subclass of javax.swing.JPanel. But it's more common to use the built-in Inspectors provided with MASON, particularly sim.portrayal.SimpleInspector.

10.1 Producing an Inspector from a Portrayal

The most common way an Inspector is produced is as a result of double-clicking on a visualized object. The Display gathers from the various Field Portrayals all objects and values which were hit by the double-click. It then asks the relevant Field Portrayals to produce Inspectors for these objects and values: "Fast" Field

[1]SWARM and Repast call these things *probes*. MASON gets the term *inspector* from the same concept found in the NeXTSTEP and Mac OS X GUI libraries.

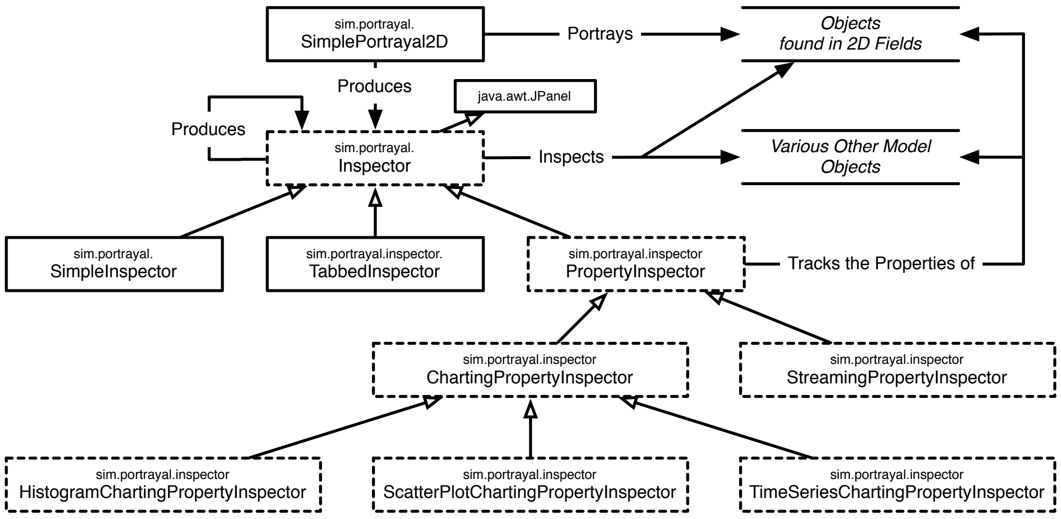

sim.portrayal. SimplePortrayal2D
Portrays
Objects found in 2D Fields
Produces
java.awt.JPanel
Produces
sim.portrayal. Inspector
Inspects
Various Other Model Objects
sim.portrayal. SimpleInspector
sim.portrayal.inspector. TabbedInspector
sim.portrayal.inspector PropertyInspector
Tracks the Properties of
sim.portrayal.inspector ChartingPropertyInspector
sim.portrayal.inspector StreamingPropertyInspector
sim.portrayal.inspector HistogramChartingPropertyInspector
sim.portrayal.inspector ScatterPlotChartingPropertyInspector
sim.portrayal.inspector TimeSeriesChartingPropertyInspector

Figure 10.1 UML diagram of MASON's Inspector facility.

Portrayals may respond to this by providing Inspectors directly, but most Field Portrayals respond by calling forth the appropriate Simple Portrayal to produce the Inspector instead. At any point a Field Portrayal or Simple Portrayal is free to refuse to provide an Inspector.

Display2D and Display3D both produce Inspectors in this way. This is done as follows:

1. The user double-clicks on a region in the Display.
2. The Display collects all the objects and values hit by the mouse, and their respective FieldPortrayals (for example Display2D does this by calling objectsHitBy(...)).
3. The Display tells each FieldPortrayal to build an Inspector for the object, by calling createInspector(...).
4. In some cases the FieldPortrayal produces the Inspector itself (usually if it's a "fast" FieldPortrayal). In other cases the FieldPortrayal calls forth the appropriate SimplePortrayal and tells it to produce the Inspector (by calling createInspector(...) on the SimplePortrayal).
5. The Display gathers all the received Inspectors, then "wraps" each Inspector in an outer Inspector which adds additional information about the location of the object. This is why you can inspect both the properties of an object and its location when you double-click on it.
6. The Display submits all "wrapped" Inspectors to the Console, where they appear under the **Inspectors** tab.

Display3D does items 2, 3, 5, and 6 internally, and a few details will be mentioned later in Section 11. But Display2D does these items by calling a specific method on itself:

sim.display.Display2D Methods

public void createInspectors(Rectangle2D.Double rect, GUIState simulation)
Builds inspectors for objects and values intersecting with the given rectangle, and submits them to the Console.

You could override this method but it'd be awfully unusual to do so.

So how is item 4 done? All Portrayals, whether FieldPortrayals or SimplePortrayals, whether 2D or 3D, respond to the same method:

sim.portrayal.Portrayal Methods

public Inspector getInspector(LocationWrapper wrapper, GUIState state)
Returns an Inspector for the object or value provided in the given wrapper.

FieldPortrayals often implement this method by calling the same method on the SimplePortrayal. The default implementation of this method in SimplePortrayal is to produce a sim.portrayal.SimpleInspector. This Inspector presents a tabular list of Java Bean Properties of the given object or value, using the sim.util.Properties facility discussed in Section 3.4. By customizing those properties as discussed in that Section, you can provide a bit of customization to the SimpleInspector. But if you want a more custom Inspector, you'll need to make one yourself and have the SimplePortrayal return it from this method.

These kinds of Inspectors are fleeting: they're generated on request, and are stored in the **"Inspectors" tab**. The user can detach them into separate windows and can also destroy them at any time. When the simulation is stopped and restarted, all such inspectors are automatically destroyed.

10.2 Producing an Inspector for a Model

An Inspector is often commonly provided for the model as a whole, where it will appear permanently in the optional **"Model" tab** of the Console.

You can specify the Inspector in question in several ways. First, you can indicate which object should be Inspected via the method getSimulationInspectedObjet: most commonly you'd provide the model itself, but you can give a proxy object of some sort if you like. Second, you can specify a sim.util.Properties object which provides the properties for a SimpleInspector. This is rare to do. Third, you can provide a custom Inspector of your own design. Here are the GUIState methods in question:

sim.display.GUIState Methods

public Object getSimulationInspectedObject()
Returns the object to inspect to provide model features, or null. Often you may override this method to return the GUIState itself, if you'd like the GUIState to be inspectable.

public Properties getSimulationProperties()
Returns a sim.util.Properties object for dynamic display of model properties. Overriding this method is extremely rare.

public Inspector getInspector()
Returns an inspector customized to display model features. The default implementation first calls getSimulationInspectedObject() and returns a sim.portrayal.SimpleInspector (Section 10.5) to inspect the returned object. If the object was null, the default implementation then attempts to call getSimulationProperties() and builds a SimpleInspector to display those properties. If *this* method returns null, then null is returned (and no model features are displayed by the Controller).

The most common approach is to just tell MASON to inspect the model directly, using a SimpleInspector (which will examine the model's Java Bean Properties), by overriding the appropriate GUIState method like this:

```
public Object getSimulationInspectedObject() { return state; }  // return the model
```

Volatility Inspectors may or may not be *volatile*, meaning that they may be updated every simulation iteration. Inspectors produced by Simple Portrayals and Field Portrayals are generally volatile. But model inspectors often are not. More often than not, a model inspector displays a set of model parameters which the user can change, rather than various variables which are updated each timestep to keep the user informed.

Volatile inspectors are expensive because they redraw themselves all the time. For this reason, by default, the inspector produced by getSimulationInspectedObject() is *not* volatile. If you want to change this, you should also override the volatility flag in the Inspector generated, like this:

```
public Inspector getInspector()
    {
    Inspector insp = super.getInspector();  // builds an inspector from getSimulationInspectedObject
    insp.setVolatile(true);
    return insp;
    }
```

By default, the inspector produced by getSimulationProperties() is volatile.

10.3 Producing an Inspector for a Field or Other Permanent Object

In addition to the model Inspector, can create additional custom "permanent" Inspectors for whatever purpose you like: most commonly, to provide some inspection of a Field. To do this, you simply attach the Inspector of your choice to a Display, using the following method:

sim.display.Display2D and sim.display.Display3D Methods

public void attach(Inspector inspector, String name)
    Attaches an inspector to the Display, assigning it the given name.

This will cause a menu entry to appear in the Display's **"Layers" button menu**. Choosing this menu will reveal the Inspector in its own window.

Unlike the Model inspector mechanism, this approach doesn't produce a default Inspector: you'll need to construct an Inspector yourself.

10.4 Producing an Inspector from Another Inspector

Last but not least, Inspectors can be generated from objects which appear as properties in other Inspectors. In some cases the user can create another sim.portrayal.SimpleInspector to inspect the object. In other cases you can call forth **property inspectors** which don't inspect a specific *object* or *value*, but rather the current *setting of a Java Bean property*. For example, you could chart the change of a numerical property of an object.

This stuff is automatically handled internally by Inspectors, and there's no API for it. However, you can register a property inspector of your creation to be able to be produced for any property. Property Inspectors are registered with MASON using a file and loaded at runtime. See Section 10.6 for more information.

10.5 Basic Inspectors

Basic Inspectors inspect objects or values, whereas Property Inspectors inspect the properties of objects. Basic Inspectors are much easier to implement. But why make one? After all, the class sim.portrayal.SimpleInspector defines a simple but very functional inspector which forms the default for much of MASON. If you do not create custom inspectors, SimpleInspector is likely the inspector to be used.

A basic Inspector is just a subclass of JPanel with an additional required method called updateInspector(). This method is called by MASON whenever it wants the inspector to revise itself to reflect changes in the inspected object. This method is abstract and if you create your own Inspector you'll have to override it.

sim.portrayal.Inspector Methods

public abstract void updateInspector()
    Updates the Inspector to reflect new information about its underlying inspected object.

If you implement this method, you'll probably want to repaint your Inspector window after doing updates. However on some systems Swing is a bit, shall we say, tempermental: your repaint can get lost if

MASON is running quickly and pounding Swing with update requests. One simple way to ensure that your window is repainted is to put an event in the Swing Event Queue which then itself turns around and puts a repaint event in the queue. This seems to solve most of Swing's issues. The magic code looks like this:

```
SwingUtilities.invokeLater(new Runnable() { public void run() { repaint(); }});
```

This code essentially replaces a repaint() call. I'd use it in Inspectors, but it's not necessary in places like Displays.

A Note on Synchronization When your custom Inspector modifies an object in response to the user making a change, it's possible (indeed likely) that the underlying model thread is running. This means that it is critical that you first synchronize on the schedule, something long the lines of:

```
synchronized(gui.state.schedule) {
do your code which changes the object
```

Due to a design oversight, the GUIState is not provided in updateInspector() — you'll have to hold onto it some other way, perhaps passed in via a constructor;.

10.5.1 Volatility

As discussed in Section 10.2, basic Inspectors can be **volatile** (or not). The definition is simple: a volatile inspector has its updateInspector() method called by MASON every iteration of the model. A non-volatile inspector does not: instead updateInspector() is only called in special circumstances.

Ordinarily you'd want a volatile inspector. But it's expensive to constantly update that Inspector. So for situations (such as a Model Inspector) where the Inspector cannot be closed, you have the option of setting it to be non-volatile. This is helpful particularly for Inspectors where the underlying object data is not expected to ever change except by the user as he modifies the Inspector.

Inspectors are by default volatile. Here are the methods for changing this:

sim.portrayal.Inspector Methods

public void setVolatile(boolean val)
Sets whether or not an Inspector is volatile.

public boolean isVolatile()
Returns whether or not an Inspector is volatile.

MASON will respect the setting you make here and change its policy for calling updateInspector() accordingly.

If you make a non-volatile inspector, you may need to provide a "refresh button" to allow the user to manually update it. In fact, sim.portrayal.SimpleInspector does this automatically. It's easy to create a JButton all set up for you, with a method called (not surprisingly) makeUpdateButton(). Pressing this button in turn calls updateInspector().

sim.portrayal.Inspector Methods

public Component makeUpdateButton()
Returns a suitable button which manually updates the inspector when pressed. This is done by calling updateButtonPressed().

This button has a specific refresh icon which looks like: ⟳ The icon comes in two forms, an "ordinary" icon and a "I've been pressed" look. Additionally, another icon (and its "pressed" look) are used here and there to indicate inspection: 🔍 sim.portrayal.Inspector defines these icons as:

```
public static final ImageIcon INSPECT_ICON; // inspect icon
public static final ImageIcon INSPECT_ICON_P; // "pressed" inspect icon
public static final ImageIcon UPDATE_ICON; // refresh icon
public static final ImageIcon UPDATE_ICON_P; // "pressed" refresh icon
```

10.5.2 Inspectors, Steppables, and Windows

MASON ordinarily updates volatile inspectors by scheduling a Steppable on the GUIState's minischedule. This steppable is then called by the minischedule each time the simulation iterates. MASON obtains this Steppable by calling the following method:

sim.portrayal.Inspector Methods

public Steppable getUpdateSteppable()
Builds a suitable Steppable which will update the Inspector each time its step(...) method is called.

MASON schedules this Steppable in a repeating fashion, which produces a Stoppable. MASON then passes this Stoppable to the Inspector and uses the one returned to instead.

sim.portrayal.Inspector Methods

public Stoppable reviseStopper(Stoppable stopper)
Given a provided Stoppable, produces a new Stoppable which, when stopped, stops the original and perhaps does other things as well. MASON will used the revised Stoppable to stop the Inspector, such as at the end of a simulation run.

Why would this be useful? Mostly to give the Inspector a chance to create a hook to be informed of when its underlying Steppable is being stopped (usually when the simulation is over). The general approach is to create an anonymous subclass of Stoppable which, when stopped, stops the original, and additionally informs the Inspector in a manner of your choosing.

Basic Inspectors can be separated from the Console's inspector list and placed in their own windows. Inspector provides two utility functions which MASON uses to perform this. You might wish to override one or both:

sim.portrayal.Inspector Methods

public String getTitle()
Returns the string which should fill the title bar of the Inspector's window. By default returns the empty String.

public JFrame createFrame(Stoppable stopper)
Produces a scrollable JFrame, with the Inspector in it, which will stop the Stoppable when the JFrame is closed.

10.5.3 SimpleInspector

sim.portrayal.SimpleInspector is the most common inspector: it's a basic inspector which exposes to the user all the properties defined by sim.util.Properties on a given object. SimpleInspector also respects the extra gizmos that sim.util.Properties provides. For example, if an object has defined a domain, SimpleInspector will draw either a slider or a pull-down menu. If an object has a proxy, SimpleInspector will show that instead. And so on.

SimpleInspector's default getTitle() calls toString() on the underlying object and returns that. However the SimpleInspector's list of properties can be bordered and given a name (like "Properties"). This name is passed in in the constructor.

SimpleInspector is largely defined by its constructors:

sim.portrayal.SimpleInspector Constructor Methods

public SimpleInspector(Object object, GUIState state, String name)
Builds a SimpleInspector based on the provided object, using the provided name as the title of the SimpleInspector's list of properties.

public SimpleInspector(Object object, GUIState state)
Builds a SimpleInspector based on the provided object, with no title for the SimpleInspector's list of properties.

public SimpleInspector(Properties properties, GUIState state, String name)
Builds a SimpleInspector based on the provided properties, using the provided name as the title of the SimpleInspector's list of properties.

Note the third method: you don't *have* to inspect an actual object. You can have the SimpleInspector actually use a Properties collection of your own devising. This makes it possible to inspect dynamic or virtual objects, so to speak, which exist in name only.

If SimpleInspector is inspecting a list or array, it will only display so many elements per page so as not to crowd the screen. That value is set to:

```
public static final int MAX_PROPERTIES = 25;
```

10.5.4 TabbedInspector

sim.portrayal.inspector.TabbedInspector provides an easy[2] way to break out an Inspector into multiple tabs to save window space. It's fairly straightforward: you create a TabbedInspector, then add Inspectors to it, each with a tab name. Then you provide the TabbedInspector to the system. Only one sub-Inspector will be displayed at a time. TabbedInspector is just a utility most easily used with custom sub-inspectors of your design.

TabbedInspector requires that its sub-Inspectors have exactly the same volatility as TabbedInspector itself: in other words, you can't have a mix of volatile and non-volatile sub-Inspectors. However you have one additional option. If TabbedInspector is updated, you can have it update either *all* of the sub-Inspectors or just the sub-Inspector which is *presently being displayed*.

TabbedInspector's constructors:

sim.portrayal.inspector.TabbedInspector Constructor Methods

public TabbedInspector()
Creates a volatile TabbedInspector.

public TabbedInspector(boolean volatile)
Creates a TabbedInspector with the given volatility.

TabbedInspector's methods are likewise obvious:

sim.portrayal.inspector.TabbedInspector Methods

[2]Well, not *very* easy.

public void setUpdatingInspectors(boolean val)
Sets whether or not the TabbedInspector is updating all inspectors or just the frontmost one.

public boolean isUpdatingInspectors()
Returns whether or not the TabbedInspector is updating all inspectors or just the frontmost one.

public void addInspector(Inspector insp, String tab)
Adds an inspector, with a tab with the given name.

public void remove(Inspector insp)
Removes an inspector.

public void clear()
Removes all inspectors.

10.5.5 Inspecting Values

Basic inspectors don't always inspect objects: sometimes they inspect values (usually numbers). For example, when you double-click on a *value* in ValueGrid2D, FastValueGrid2D, or similar field, up will come an inspector for the value at that position. What is being shown you?

Answer: ValueGrid2D — or more properly, ValuePortrayal2D — created a special SimpleInspector which inspects a custom object which displays the value. When SimpleInspector modifies the properties of the object, the value will be changed in the field.

This object is called a **value filter**. ValuePortrayal2D has two value filter, one for integers (sim.portrayal.simple.ValuePortrayal2D.IntFilter) and one for doubles (sim.portrayal.simple.ValuePortrayal2D.DoubleFilter). Both are subclasses of the abstract class sim.portrayal.simple.ValuePortrayal2D.Filter.

Value filters have a simple constructor, Here's the one for Filter:

sim.portrayal.simple.ValuePortrayal2D.Filter Constructor Methods

public Filter(LocationWrapper wrapper)
Creates a filter for the value stored in the given wrapper.

If you think about it, the object in the LocationWrapper is really immaterial for our purposes — it's just a dummy (a MutableDouble) holding the current number value. For us, what matters is the *location* stored in the LocationWrapper, which the Filter uses to extract and/or change the current value in the field.

IntFilter and DoubleFilter both have getters and setters for the value. Here's the ones for DoubleFilter:

sim.portrayal.simple.ValuePortrayal2D.DoubleFilter Methods

public void setValue(double val)
Sets the current value in the field.

public double getValue()
Returns the current value in the field.

You probably will never need to use these classes, but it's useful to understand how they work in case you need to do a similar thing.

10.6 Property Inspectors

All Property Inspectors are subclasses of the abstract class sim.portrayal.inspector.PropertyInspector. A Property Inspector is not like a basic Inspector. While a basic Inspector inspects a given *object* (or in some cases *value*), a Property Inspector inspects a *property of an object*. While objects can't change, their properties can change.

Property Inspectors are nearly always volatile, and unlike basic Inspectors, they are **dynamically loaded at runtime**: they're basically plug-ins to MASON. You can create property inspectors and add them to MASON and they'll appear everywhere for all applications. Last, though Property Inspectors are JPanels, like all Inspectors, it's often the case that this JPanel is not displayed, or the Property Inspector creates a window which is shared with other Property Inspectors (such as multiple time series sharing the same chart). All this means that Property Inspectors are quite a bit more complicated than basic Inspectors. So get ready!

10.6.1 How Property Inspectors are Created

Property Inspector construction is somewhat complex. Each Property Inspector class in MASON is specified in a file called sim/portrayal/inspector/propertyinspector.classes. This file has the full name for each property inspector class defined one on a line. Blank lines are also acceptable, as are comments, which are full lines starting with a pound sign (#). Here is a typical MASON propertyinspector.classes file:

```
# This is the propertyinspector.classes file.  Add your
# PropertyInspector subclass to the list below if you wish
# it to be included for consideration in that menu.
# Empty lines in this file are ignored, as
# is any part of a line which starts with a pound sign.

sim.portrayal.inspector.StreamingPropertyInspector
sim.portrayal.inspector.TimeSeriesChartingPropertyInspector
sim.portrayal.inspector.HistogramChartingPropertyInspector
sim.portrayal.inspector.ScatterPlotChartingPropertyInspector
```

When a MASON simulation is fired up, these classes are loaded from the file and queried about what **types** of properties they are capable of examining. Not all Property Inspectors can inspect all properties. For example, a Property Inspector designed to chart numerical values wouldn't make sense being registered for properties which return Strings. The Inspectors are also queried regarding their **name**, a String which will appear in the pop-up menu by which a Property Inspector is chosen. These aren't really names per se, but rather menu options. For example the name for sim.portrayal.inspector.HistogramChartingPropertyInspector is "Make Histogram".

These are defined in the static methods:

sim.portrayal.inspector.PropertyInspector Methods

public static String name()
  Returns the PropertyInspector's preferred name.

public Class[] types()
  Returns an array consisting of the kinds of types the PropertyInspector is capable of inspecting. Property types can be class names, or they can be numerical or boolean types, or a mix of the two. In the case of numerical or boolean types, use the TYPE constant for that numerical type: for example for doubles, use the constantjava.lang.Double.TYPE. If null is returned, this PropertyInspector is assumed to accept all data types. If an empty array is returned, it is assumed to accept no data types at all (not very useful!).

At this point MASON may display, for various properties of an object, a pop-up menu of PropertyInspector choices, by calling getPopupMenu(...). For example, this is done when the user clicks on the 🔧 icon next to a property in a SimpleInspector. This method goes through each of the registered PropertyInspector classes, determines which ones have valid types, then calls getMenuNameForPropertyInspectorClass(...), which produces the menu text for the PropertyInspector, typically by calling name() on the relevant class.

When the user chooses one of these menu options, MASON will create a PropertyInspector for that property. This is done by calling a factory method, makeInspector(...), to build the PropertyInspector. makeInspector(...) first constructs the PropertyInspector using Java Reflection. It then calls isValidInspector() on this Inspector to determine if the inspector is **valid** (it's invalid if the user cancelled during construction, or an error occurred). If it's valid, makeInspector(...) returns the inspector, else it returns null.

sim.portrayal.inspector.PropertyInspector Methods

public static PropertyInspector(PropertyInspector makeInspector(Class inspectorClass, Properties properties, int index, Frame parent, GUIState gui)

Produces a PropertyInspector ready to display, or returns null if the user cancelled the PropertyInspector creation or some event or error occurred which prevented the PropertyInspector from being constructed. The PropertyInspector's class is provided, along with the Properties of the object to inspect, and the index of the desired property.

public static String getMenuNameForPropertyInspectorClass(String classname)

Determines the name of the PropertyInspector class to be displayed in the pop-up menu.

public static JToggleButton getPopupMenu(Properties properties, int index, GUIState state, JPopupMenu pop)

Produces a popup menu attached to a JToggleButton which shows all valid PropertyInspectors for a given property. The property is specified by the provided index. If a JPopupMenu is provided, it will be used and added to. Otherwise a new one will be created.

protected void setValidInspector(boolean val)

Sets whether or not the constructed PropertyInspector is valid. Note that this method is protected: only instances may set themselves as valid.

public boolean isValidInspector()

Returns whether or not the constructed PropertyInspector is valid.

This multi-step process allows PropertyInspectors to, among other things, pop up modal dialog boxes to more specifically determine user preferences, or cancel his option. For example, sim.portrayal.inspector.StreamingPropertyInspector pops up a window asking the user where he'd like to stream to (a file? a window?). And sim.portrayal.inspector.HistogramChartingPropertyInspector pops up a window asking the user if he'd like to create the histogram on a new chart, or add it to an existing chart owned by another HistogramChartingPropertyInspector.

Some PropertyInspectors want to be automatically displayed in windows: others do not, but rather do their work hidden (perhaps creating their own windows). MASON will query the PropertyInspector regarding this using shouldCreateFrame() and display the PropertyInspector (or not) accordingly. By default shouldCreateFrame(returns true. Override it to return false if you like.

Why is shouldCreateFrame() overridden but isValidInspector() is set?

Oh, you saw that, did you? The excuse for this seeming disparity is that PropertyInspector classes nearly always have a fixed value to return for shouldCreateFrame, but the value returned by isValidInspector() varies from instance to instance.

Finally, after the PropertyInspector has had its reviseStopper(...) method called, the revised Stoppable is again passed to it via setStopper(...). This method stores the Stoppable in the PropertyInspector to be accessed later with getStopper(). This is used by some PropertyInspectors to cancel themselves in some way other than closing the window (recall not all PropertyInspectors have windows).

Why have both reviseStopper(...) and set/getStopper(...)?

Sure, you could have done this with just reviseStopper(...). But it's convenient to have a separate method. Think of it this way: reviseStopper(...) is mostly used to revise the Stoppable so as to remain *informed* that the Inspector is being stopped. Whereas set/getStopper(...) are used to enable the PropertyInspector to conveniently *stop itself*. What's unexpected is the fact that Inspector doesn't have set/getStopper(...). Perhaps we'll refactor it one day (it's less useful for basic Inspectors).

Collectively, these methods are:

sim.portrayal.inspector.PropertyInspector Methods

public boolean shouldCreateFrame()
: Returns true (the default) if the PropertyInspector, once created, should be placed in a JFrame and displayed. Override this to return false if the PropertyInspector handles its own windows.

public void setStopper(Stoppable stopper)
: Sets the Stoppable for the PropertyInspector.

public Stoppable getStopper()
: Returns the Stoppable for the PropertyInspector.

10.6.2 Charting Property Inspectors

> *Charting Property Inspectors are all well and good, but I want a permanent Charting display for my model.*
>
> With some elbow grease, you can get charting in a Display quite nicely. For details, see the file docs/howto.html, under the title "How to Display a Chart Programmatically".

MASON provides three Property Inspectors which produce charts using the JFreeChart[3] library. The Property Inspectors are subclasses of the abstract superclass sim.portrayal.inspector.ChartingPropertyInspector. These Property Inspectors rely on **chart generators**, utility classes discussed later in Section 12. Chart generators manage the complexity of interfacing with JFreeChart. Charts also use the Preferences facility (Section 8.3), with the Preferences key:

```
public final static String chartKey = "sim.portrayal.inspector.ChartingPropertyInspector";
```

The current Charting Property Inspectors are:

- sim.portrayal.inspector.TimeSeriesChartingPropertyInspector produces time series describing the progression of a numerical value over time. It is capable of inspecting any boolean, byte, short, int, long, float, double, java.lang.Number, or sim.util.Valuable.
- sim.portrayal.inspector.ScatterPlotChartingPropertyInspector produces scatter plots: the scatter plot changes each iteration. This class is capable of inspecting (and extracting its data from) any array of sim.util.Double2D or sim.util.Int2D. Each Double2D or Int2D represents a point on the scatter plot.
- sim.portrayal.inspector.HistogramChartingPropertyInspector produces histograms which change each iteration. This class is capable of inspecting (and extracting its data from) any array of booleans, bytes, shorts, ints, longs, floats, doubles, java.lang.Number, or sim.util.Valuable. It is also capable of inspecting a sim.util.IntBag or sim.util.DoubleBag. Each element in the array or Bag represents a sample in the histogram. Histograms by default have the following number of bins, which can be changed at any time by the user:

  ```
  public static final int DEFAULT_BINS = 8;
  ```

Each PropertyInspector is created using the same constructors: the superclass versions are shown here:

sim.portrayal.inspector.ChartingPropertyInspector Constructor Methods

public ChartingPropertyInspector(Properties properties, int index, GUIState gui, ChartGenerator generator)
: Produces a ChartingPropertyInspector with the given (possibly shared) generator and the property defined by the given index in the properties list. The generator is checked for validity.

[3]http://jfree.org

public ChartingPropertyInspector(Properties properties, int index, Frame parent, GUIState gui)
Produces a ChartingPropertyInspector with a potentially new ChartGenerator and the property defined by the given index in the properties list. The user may be queried to determine which ChartGenerator to use.

These constructors in turn may call the method validChartGenerator(...) to determine if a provided ChartGenerator is valid, or they may call createNewGenerator() to produce a new one from scratch. The generator is then stored internally but may be accessed with getGenerator(). These classes also maintain a list of all current charts on-screen, in order to query the user as to which chart he'd like the ChartingPropertyInspector to draw on (or create a new one). This list is defined by getCharts(...).

sim.portrayal.inspector.ChartingPropertyInspector Methods

public boolean validChartGenerator(ChartGenerator generator)
Returns true if the given generator can be used by the ChartingPropertyInspector.

public ChartGenerator createNewGenerator()
Returns a brand-new ChartGenerator.

public ChartGenerator getGenerator()
Returns the current ChartGenerator.

protected Bag getCharts(GUIState gui)
Returns all the presently on-screen charts associated with the given simulation.

Series, Series Attributes, and Global Attributes Each ChartingPropertyInspector stores a *series* of data. Multiple series may be displayed on the same chart, and so multiple ChartingPropertyInspectors may share the same chart.

A ChartingPropertyInspector's series is updated each iteration from (ultimately) the updateInspector() method. This in turn will call the method updateSeries(...) to give the ChartingPropertyInspector the chance to load new series data from the model before it is drawn on-screen.

Each series has a set of **series attributes**: for example, the color or line style of the series when drawn on-screen (to distinguish it from others). These are defined by a subclass of sim.util.media.chart.SeriesAttributes special to the ChartGenerator being used. Different SeriesAttributes objects have different features. You can get the current SeriesAttributes object with getSeriesAtributes(), though you'll need to know what subclass it is.

Certain ChartingPropertyInspectors can also control how often the data on a given chart is updated, and if there is too much data, how the data should be compacted (which samples should be removed). This is defined by an instance of a protected class called sim.util.media.chart.ChartingPropertyInspector.GlobalAttributes, of which there is a protected instance:

```
protected GlobalAttributes globalAttributes;
```

This class defines at least the following variables:

```
public long interval; // how long (in milliseconds) to wait before aggregating
public int aggregationMethod; // how to aggregate
public int redraw; // how to wait before redrawing
```

Aggregation works by breaking the time series into intervals, then selecting one sample from that interval to retain (the others are eliminated). Typically each interval contains two data points. Aggregation methods include:

```
protected static final int AGGREGATIONMETHOD_CURRENT; // don't aggregate
protected static final int AGGREGATIONMETHOD_MAX; // retain the latest sample
protected static final int AGGREGATIONMETHOD_MIN; // retain the earliest sample
protected static final int AGGREGATIONMETHOD_MEAN; // replace all samples with their mean
```

Redrawing also has certain options:

```
protected static final int REDRAW_ALWAYS; // always redraw
protected static final int REDRAW_TENTH_SEC; // redraw once every 1/10 second
protected static final int REDRAW_HALF_SEC; // redraw once every 1/5 second
protected static final int REDRAW_ONE_SEC; // redraw every second
protected static final int REDRAW_TWO_SECS; // redraw every two seconds
protected static final int REDRAW_FIVE_SECS; // redraw every five seconds
protected static final int REDRAW_TEN_SECS; // redraw every ten seconds
protected static final int REDRAW_DONT; // never redraw
```

Some ChartingPropertyInspectors (such as for histograms) don't do data aggregation at all, and indicate as such by returning false for includeAggregationMethodAttributes().

sim.portrayal.inspector.ChartingPropertyInspector Methods

protected abstract void updateSeries(double time, double lastTime)
Updates the series in the ChartingPropertyInspector. The current simulation time is provided, along with the last time when this method was called (to determine if enough time has elapsed to do another update, for example).

public SeriesAttributes getSeriesAttributes()
Returns the current SeriesAttributes object for the ChartingPropertyInspector.

public GlobalAttributes getGlobalAttributes()
Returns the current GlobalAttributes object for the chart on which the ChartingPropertyInsepctor is displaying its series.

protected boolean includeAggregationMethodAttributes()
Returns true if the ChartingPropertyInspector uses data aggregation.

10.6.3 Streaming Property Inspectors

MASON also provides a Property Inspector which streams property values out to a stream as time passes. This class is called sim.portrayal.inspector.StreamingPropertyInspector.

Compared to the Charting property inspectors, StreamingPropertyInspector is fairly simple. The user can choose from four streaming options:

- Stream to a file, overwriting the original file if there was one.
- Stream to a file, appending to the original file if there was one.
- Stream to a scrolling JTextArea in a window.
- Stream to Standard Out (System.out).

Additionally, StreamingPropertyInspector can be overridden to provide a custom streaming location.

Nearly everything of relevance is handled by StreamingPropertyInspector's two constructors: one meant for user options, and one for custom streaming:

sim.portrayal.inspector.StreamingPropertyInspector Constructor Methods

public StreamingPropertyInspector(Properties properties, int index, Frame parent, GUIState gui)

Produces a new StreamingPropertyInspector on the given property index, with the user choosing how the data is to be streamed out.

public StreamingPropertyInspector(Properties properties, int index, Frame parent, GUIState gui, PrintWriter stream, String streamName)

Produces a new StreamingPropertyInspector on the given property index, which writes to the provided stream, using the given name to describe the stream.

Chapter 11

Visualization in 3D

Yet to come.

11.1 3D Field Portrayals

11.2 3D Simple Portrayals

Chapter 12

GUI and Media Utilities

Yet to come.

Index