# MASON: A New Multi-Agent Simulation Toolkit

Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, and Keith Sullivan
Department of Computer Science and Center for Social Complexity
George Mason University
4400 University Drive, Fairfax VA 22030
http://cs.gmu.edu/∼eclab/projects/mason/

## Abstract

*We introduce MASON, a fast, easily extendable, discrete-event multi-agent simulation toolkit in Java. MASON was designed to serve as the basis for a wide range of multi-agent simulation tasks ranging from swarm robotics to machine learning to social complexity environments. MASON carefully delineates between model and visualization, allowing models to be dynamically detached from or attached to visualizers, and to change platforms mid-run. We describe the MASON system, its motivation, and its basic architectural design. We then discuss five applications of MASON we have built over the past year to suggest its breadth of utility.*

## 1. Introduction

MASON is a single-process discrete-event simulation core and visualization toolkit written in Java, designed to be flexible enough to be used for a wide range of simulations, but with a special emphasis on "swarm" simulations of a very many (up to millions of) agents. The system is open-source and free, and is a joint effort of George Mason University's Computer Science Department and the George Mason University Center for Social Complexity. MASON may be downloaded at http://cs.gmu.edu/∼eclab/projects/mason/

MASON is not derived from any other toolkit, but rather was built from scratch from first principles. Our design philosophy was to build a fast, orthogonal, minimal model library to which an experienced Java programmer can easily add features, rather than one with many domain-specific, intertwined features which are difficult to remove or modify. To this we have added visualization and GUI facilities we have found useful for a variety of simulation tasks. We plan to position MASON as a core for new simulation libraries, and also as a toolkit sufficient for simple simulations.

The library was intended for researchers, such as ourselves, who needed to perform many simulation runs, possibly with large numbers of agents and interactions, with occasional visualization and modification of the runs. To this end MASON is fast, portable, capable of checkpointing and restarting models with or without visualization, able to migrate models across platforms, and capable of producing guaranteed-duplicatable results independent of platform. MASON models may be attached to a provided GUI toolkit which enables visualization and manipulation of the model in both 2D and 3D (using Java3D), and which can produce screenshots and movies.

MASON does not presently provide high-level model-building tools for inexperienced programmers; nor does it provide domain-specific features such as physics models, robot sensors, built-in charts and graphs, or data-import from geographic information systems. Instead, we hope external modules for various functions will be created to extend MASON, and we will be creating some of these ourselves in the near future.

While there are a many similarities between MASON and existing popular multi-agent simulation toolkits, we believe that MASON's combination of architecture and features are unusual for a multi-agent simulation system. In this paper we will discuss the motivation and architectural design of the system, and then detail five applications of MASON presently under way.

## 2. Motivation and Design Goals

We began work on MASON because we needed a simulation toolkit which made it relatively easy for us to create a very wide range of multi-agent and other simulation models, and to run many such models efficiently in parallel on back-end cluster machines. Domains to which we intended to apply the simulator ran the gamut from robotics, machine learning and artificial intelligence to multi-agent models of social systems (political science, historical development, land use, economics, etc.).

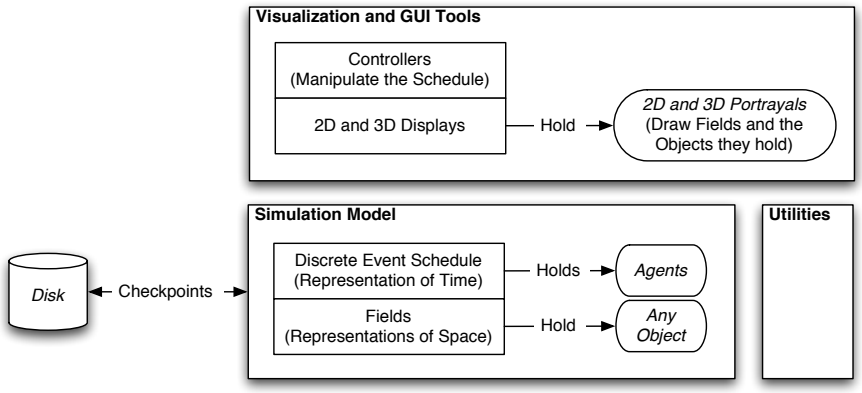


**Figure 1. Basic elements of the MASON model and visualization layers.**

Our previous research in these areas had either relied on a heavily modified robotics simulator (notably TeamBots[1]), a compiled social complexity toolkit such as SWARM[2], Ascape[3], or RePast[4], or an interpreted rapid-development library such as StarLogo[5], NetLogo[6], or Breve[7].

We typically needed to run many (>100,000) simulation runs to optimize model parameters or perform machine learning in a multi-agent problem domain. In such cases we had to "cook" the simulations on multiple back-end servers (in Linux, Solaris, and MacOS X), while occasionally viewing the results on a front-end MacOS X or Windows workstation. This required speed, the ability to migrate a simulation run from platform to platform, and (for our purposes) guaranteed platform-independence. Further, we needed to be able to customize the simulation to a wide range of multi-agent simulation environments. Existing systems did not meet these needs well, either because they tied the model to the GUI too closely, could not guarantee platform-independent results, or being written in an interpreted language, were slow. Additionally, many such systems, particularly the robotics simulators, were by-and-large geared to a particular problem domain. Rather than remove special-purpose code from an existing system (potentially introducing bugs), we instead hoped to build on top of a simple, general-purpose simulator.

Given MASON's motivations in large-scale parallel simulation, easy "hackability", and domain-independence (within the aegis of multi-agent simulation), its design goals were as follows:

- A small, fast, easily understood, and easily modified core.
- Separate, extensible visualization in 2D and 3D.
- Production of identical results independent of platform.
- Checkpointing any model to disk such that it can be resumed on any platform with or without visualization.
- Efficient support for up to a million agents without visualization.
- Efficient support for as many agents as possible under visualization.
- Easy embedding into larger existing libraries, including having multiple instantiations of the system coexisting in memory.

There were three design goals we explicitly did *not* make for MASON. First, we did not intend to include parallelization of a single simulation across multiple networked processors. Such an architecture is radically different than a single-process architecture. Second, we intended the MASON core to be simple and small, and so did not provide built-in features special to social agents or robotics simulators. We felt such things were more appropriately offered as optional domain-specific modules in the future. Third, although we tried to be reasonably memory-efficient, this was not a priority.

## 3. Architecture

MASON is written in Java in order to take advantage of its portability, strict math and type definitions (to guarantee duplicatable results), and object serialization (to checkpoint out simulations). Java has an undeserved reputation for slowness; and our past experience in developing the ECJ

1 www.teambots.org
2 www.swarm.org
3 www.brook.edu/dybdocroot/es/dynamics/models/ascape
4 www.repast.org
5 education.mit.edu/starlogo
6 ccl.sesp.northwestern.edu/netlogo/
7 www.spiderland.org

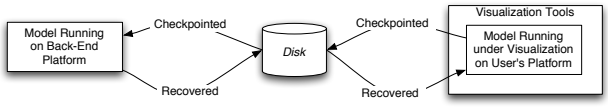


**Figure 2. Checkpointing and recovering a MASON model to be run standalone or under different kinds of visualization.**

evolutionary computation toolkit [5] suggested (correctly) that carefully-written Java code can be surprisingly fast.

The toolkit is written in a modular, layered architecture, as shown in Figure 1. At the bottom are a set of utility data structures which may be used for any purpose. Next comes the model layer, a small collection of classes consisting of a discrete-event schedule, a high-quality random number generator, and a variety of *fields* which hold objects and associate them with locations. This code alone is sufficient to write basic simulations running on the command line.

The visualization layer allows for display of fields and user control of the simulation. Some examples of visualized fields are shown in Figure 3. A bright line separates the model layer from the visualization layer: a suite of tools which allow runtime drawing and manipulation of the model. This allows us to treat the model as a self-contained entity. We may, at any time, separate the model from the visualization, checkpoint the model to disk, move it to a different platform and let it continue to run, or attach an entirely different visualization collection. Figure 2 shows this procedure.

All elements of MASON's model and visualization layers are self-contained and may be easily replaced or extended.

### 3.1. The Model Layer

MASON's model layer has no dependencies on the visualization layer and can be entirely separated from it. A MASON model is entirely contained within a single instance of a user-defined subclass of MASON's model class (`SimState`). This instance contains a discrete-event schedule and zero or more fields.

*Agents and the Schedule* MASON employs a specific usage of the term *agent*: a computational entity which may be scheduled to perform some action, and which can manipulate the environment. Note that we do not explicitly state that the agent is physically *in* the environment, though it may be; in this case we would refer to the agent as an *embodied agent*. Agents are brains, and do not need to be bodies. MASON does not schedule events on the schedule to send to an agent; rather it schedules the agent itself.

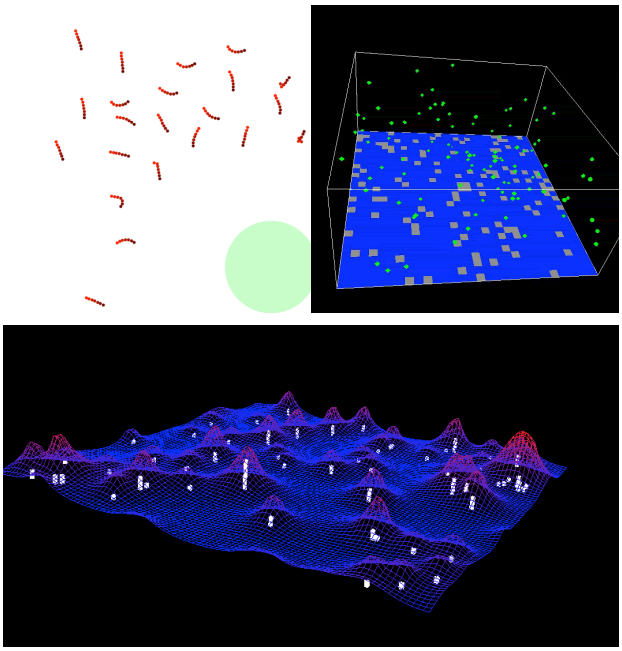

**Figure 3. Visualized fields in MASON, showing various forms of 2D and 3D continuous and discrete space.**

Scheduling an agent multiple times for different functions is easily done with an anonymous wrapper class.

MASON does not provide subschedules; instead, other facilities perform similar functions. Specifically, MASON provides various wrappers which can group agents together, iterate them, perform them in parallel on separate threads, etc. The schedule also allows for subdivisions of a single time tick.

*Fields* MASON's fields relate arbitrary objects or values with locations in some notional space. Many of these fields are little more than wrappers for simple 2D or 3D arrays. Others provide sparse relationships. An object may exist in multiple fields at one time (and, for some fields, in the same field more than once). The use of fields is entirely optional, and the user can add additional fields. MASON provides fields for:

- 2D and 3D arrays of objects, integers or doubles which are bounded or toroidal; and with hexagonal, triangular, or square layouts.
- 2D and 3D sparsely populated object grids which are bounded, unbounded, or toroidal; and with hexagonal, triangular, or square layouts.
- 2D and 3D sparse continuous (real-valued) space.
- Directed networks (graphs).

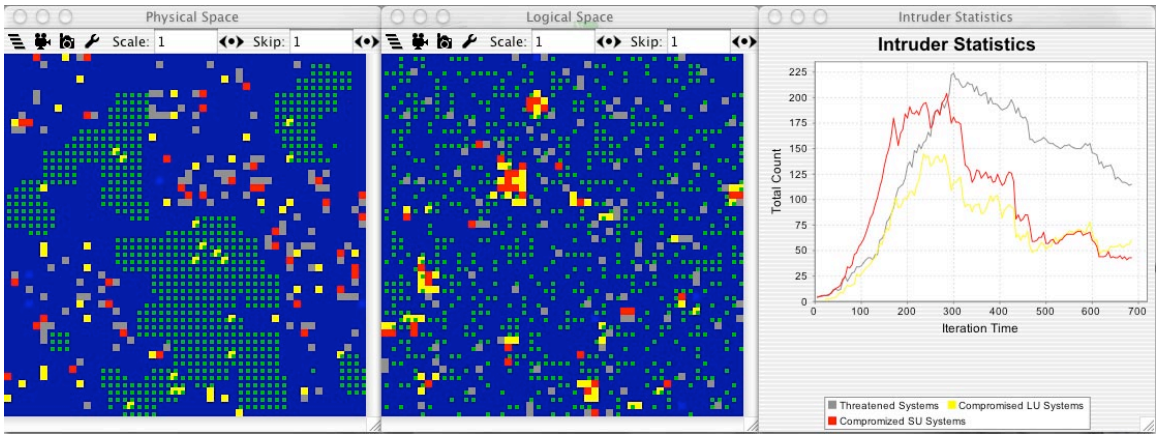


**Figure 4. Network Intrusion model: the physical (left) and logical (center) spaces, together with statistics on intrusions and compromised systems (right).**

### 3.2. The Visualization Layer

Objects in the visualization layer may examine model-layer objects only with the permission of a gatekeeper wrapper around the `SimState` called a `GUIState`. It is this class which can detach the model entirely and serialize the `SimState` to or from disk. As certain objects in the visualization world need to be scheduled (windows notably need to refresh themselves to reflect changes in the model), the `GUIState` also provides its own mini-schedule which is kept in sync with the model's underlying schedule. This allows the visualization layer to be entirely separate from the model.

MASON performs visualization through one or more *displays*, GUI windows which provide 2D and 3D views on underlying fields. Displays have a many-to-many relationship with fields, and 3D displays may visualize 2D fields as well. Each display holds one or more *field portrayals*, provided proxy objects responsible for drawing fields and allowing the user to inspect or change their contents. There is a many-to-one relationship between field portrayals and underlying fields.

In turn, the field portrayals portray their fields by calling up *simple portrayals* responsible for drawing or inspecting various kinds of objects stored in the fields. Objects may serve as their own portrayals but do not have to. There is a many-to-many relationship between simple portrayals and their underlying objects stored in fields. Portrayals also allow for *inspectors* (what SWARM would call "probes") of underlying model objects. The model itself also may have a global inspector. Inspection is done through Java's Bean Properties facility.

MASON provides an elaborate `Console`, a graphical interface widget which makes it easy for the user to start/stop/pause/step the schedule, to load and save serialized models, to show and hide displays, to load different simulations, and to view inspectors. The `Console` is not required and can be replaced with simpler implementations.

## 4. Using and Extending MASON

Because of the separation, MASON models are usually created in two stages. First, the author develops the model proper simply as a self-contained subclass of `SimState`, complete with a boilerplate command-line loop which starts the simulation, steps the schedule, and then closes down. After this code is completed, the MASON model should be able to run on the command line as a self-contained GUI-less application. Second, the author creates a `GUIState` to encapsulate the `SimState`, attaching portrayals and displays. At this point, the simulation can be also visualized.

MASON was specifically designed to be easily developed into a custom-purpose simulator, and only provides core tools common to most simulation needs. We have two branches of active development on the system. First, MASON does not provide tools such as graphing and charting or statistical facilities. Instead was have relied on well-established libraries such as ptplot or JClass Chart which are very easily integratable into MASON. We intend to keep the MASON core clean, and so our plan is to provide wrapper code for such tasks as a separate downloadable module.

Second, we are very interested in extending MASON to be used as a core in robotics or physical simulations similar to those in Breve or Player/Stage. We are planning to define as a field an existing Java physics engine, and to provide 3D visualization of the field.

## 5. Applications

MASON has existed for a little over a year at GMU; but we have already used it for a number of simulation tasks ranging from micro-air vehicle coordination to models of virus propagation. Here we will mention a few of interest. Most such are discrete-grid simulation worlds common to SWARM and related simulators; but one (cooperative target observation) has more in common with robotics simulators such as TeamBots.

### 5.1. Network Intrusion and Countermeasures

NetInt is an agent-based model designed to study computer network security issues, originally developed in Ascape and then ported to MASON by an inexperienced MASON developer to test the difficulty and speed of porting to the new system (with, we felt, very positive results). The current version models a network of 2500 computer systems connected via two overlaid topologies: IP address space (or physical space), and remote login space. In this space live two kinds of agents: computer systems and one or more hackers. Each computer system contains a set of security policies implemented when the system is believed to be compromised. Different computers may have different levels of security. The hacker agents have various levels of ability to break into systems at different rates. Initially, a hacker starts in control of a single computer.

A computer may be classified as secure (recall that there are several levels of security possible). Alternatively, computers may be classified as insecure, and there are several types that fall into this category. First, a computer system may be threatened, in the sense that a nearby computer (in either physical or logical space) has been compromised. Second, the system may be compromised at a lower-user level, in which case the attacker does not have (yet) too many privileges. Third and last, a computer may be compromised at the super-user level.

The parameters of the model allow one to understand the effects of changes in security policies as well as the effects of changes in hacker behavior. Figure 4 shows a snapshot of a simulation. The left and center panels show the physical and the logical spaces of computer systems. The rightmost panel shows the number of intrusions detected as the simulation progresses. For the sets of policies used in this experiment, we can observe a rapid increase in the number of threatened and compromised systems, peaking at 200 affected systems. At this point, the security policies start identifying, isolating and fixing the affected computers, leading to a decrease both in the number of threatened and compromised systems.

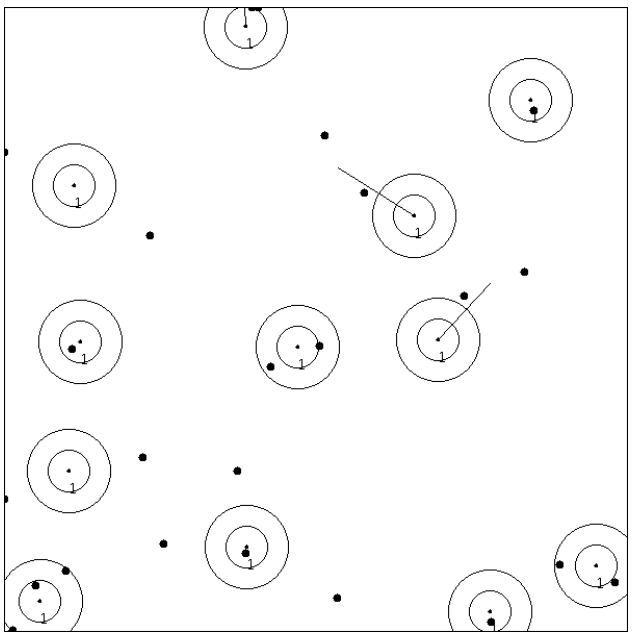

**Figure 5. Cooperative Target Observation model. Small doubly-circled dots are observers. Outer circles are their observation ranges. Large dots are targets. Straight lines connect observers with newly-chosen desired destinations.**

### 5.2. Cooperative Target Observation in Unmanned Aerial Vehicles

Unmanned aerial vehicles (UAVs) present a variety of interesting problems in cooperative robotics. In recent experiments [6] we examined the effectiveness of various algorithms in performing cooperative target observation (CTO). In our problem formulation, mobile UAV agents (called *observers*) collectively attempt to stay within an "observation range" of as many targets as possible. The targets wander randomly and are slower than the observers. The environment is bounded and clear of obstacles. Observers know the positions of all other observers and targets in the environment.

The CTO environment is shown in Figure 5. We used this environment to examine "tunably decentralized" cooperative algorithms, whereby changing a parameter we could gradually shift the algorithm from one global decision-making procedure to separate per-agent procedures. We examined two such algorithms for controlling the observers, based on K-means clustering and hill-climbing respectively, under combinations of decentralization, target velocity, rate of decision-making, and observation sensor range, yielding 4050 simulation runs all told.

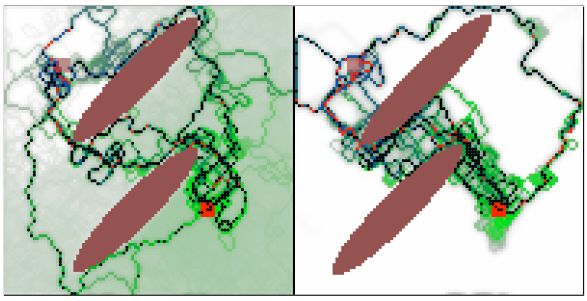

**Figure 6. Ant foraging with two obstacles: early (left) and late (right) snapshots of the simulation**

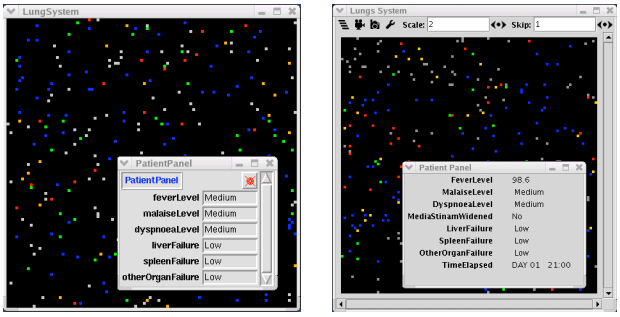


**Figure 7. Panels from the Anthrax Propagation model: (left) original SWARM model and (right) MASON replication.**

As both algorithms are tunably decentralized in a similar fashion, we expected that both would degrade in the same way: but this was not at all the case. Surprisingly, hill-climbing was sensitive to the degree of decentralization, but K-means was not. This was the case even though neither algorithm was uniformly superior to the other across all problem settings. We also considered the two in combination: K-means clustering followed by hill-climbing, which produced results as good as either of the two separately.

### 5.3. Ant Foraging

Swarm behavior algorithms are increasingly popular approaches to clustering and foraging tasks in multi-agent and robotics research [1, 4], and have served as inspiration for new kinds of population-oriented optimization methods [2]. We have recently examined how to augment swarm behaviors with pheromones to perform foraging tasks. Specifically our agents perform ant-like "central place food foraging", whereby agents leave a nest to search for food, then return to the nest laden with food. To assist them in their task, the agents deposit pheromones on the ground and respond to pheromone signals in various ways.

We are interested in adapting the concept of pheromones to various artificial-agent tasks. Our model differed from previous biologically-realistic approaches in that our "ants" used two pheromones: one to repeatedly find the food source, and another to locate the nest. When searching for food, the ant lays down a pheromone gradient to the nest while following a to-food gradient set up by the other pheromone. When returning to the nest, the opposite action occurs. The resulting algorithm is formal and efficient.

Figure 6 shows a typical 100x100 cell environment with one thousand ants, a nest (bottom right), a food source (top left), and two large elliptic obstacles. The ants cooperatively discover and optimize to a minimum-length trail. Our experiments in this environment [8] suggested that pheromones bear a strong resemblence to utility value functions found in dynamic programming and reinforcement learning. Indeed, the best pheromone-update functions we have discovered have an unusual kinship to value iteration and TD-learning equations.

Using these methods, we have developed ant trails which perform rapid local optimization, global search, and dynamic updating as food sources move or deplete or new obstacles appear in the environment. With more pheromones, the ants can also learn complex tours with multiple waypoints and self-intersecting paths.

We have also experimented with letting the computer search for and optimize these behaviors on its own. For this purpose, we connected MASON to the ECJ evolutionary computation system [5]. ECJ handled the main evolutionary loop: an "individual" (a candidate solution) took the form of a set of ant behaviors that was applied to each ant in the colony. To assess the quality of an individual, ECJ spawned a MASON simulation with the specified ant behaviors. The simulation was run for several hundred timesteps. At the end of the simulation, the amount of food foraged indicated the individual's fitness. More details on these experiments are reported in [7].

### 5.4. Anthrax Propagation in the Human Body

The interaction between pathogens and infected hosts is usually investigated using laboratory and live studies. But for some diseases, like inhalation anthrax, live studies are not possible due to their deadly effects. After examining laboratory studies we developed an agent-based model that would help researchers to simulate the spread of one such pathogen (anthrax) through various human organs, to test intervention strategies, and to explore what-if scenarios.

The onset, duration and outcome of inhalation anthrax is a complex dynamic process. We modeled the disease as a

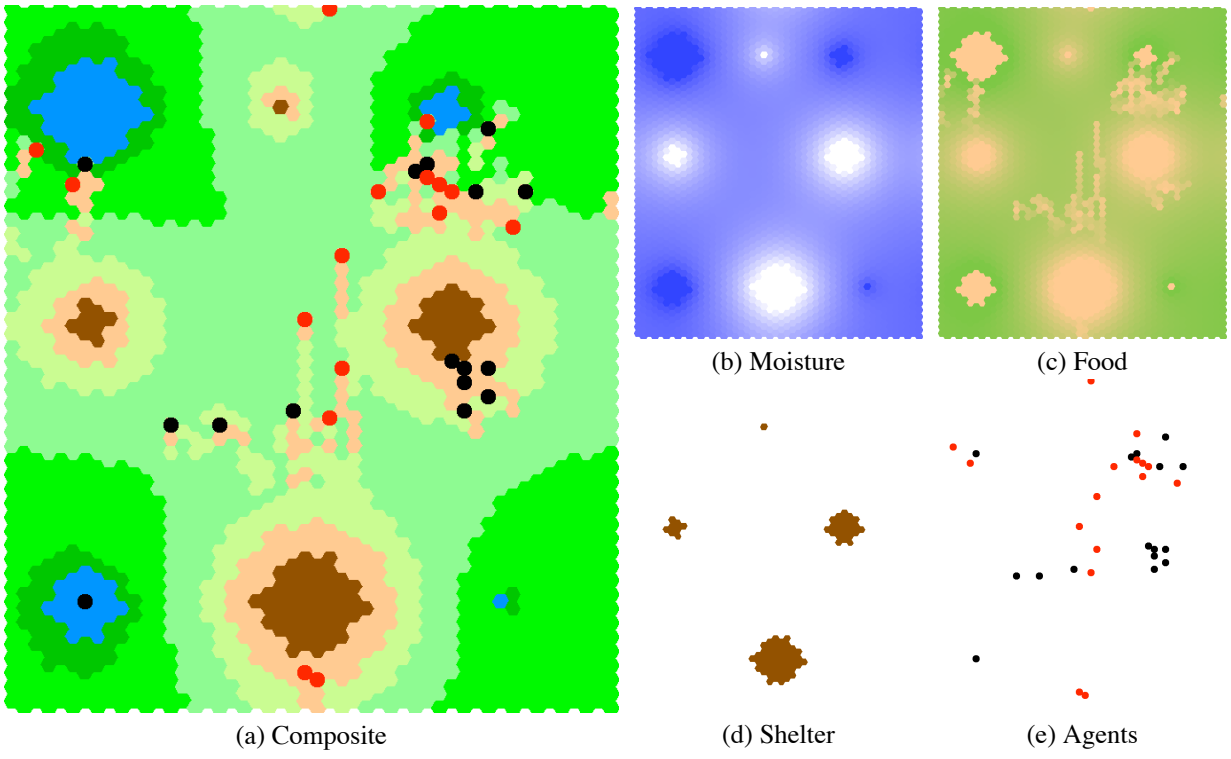


**Figure 8. Wetlands initial visualization and layers. Composite visualization (a) consisting of moisture layer (b), food sites layer (c), shelter sites layer (d), and agents layer (e). Agents are mobile but all environmental components are fixed in Wetlands 1.1 (though environmental components may change in value).**

series of discrete events that map out a time course for infection in the human body. Different systems in the human body which play a role in inhalation anthrax are modeled as spatial entities to show how the anthrax disease flows through the body. Each sytem has its own properties and interacts with anthrax spores that invade their space as well as with the other systems connected to them. The systems modeled are the lungs, primary and media stinam lymph nodes, liver, spleen, and circulatory system.

The dynamics of these interactions are visually displayed in the form of agents interacting with the systems. Each different colored square represent a type of agent in a system (anthrax spores, various cells, etc.). The system also displays statistics on the interactions of the systems and on the patient's health and disease state.

The Anthrax model was developed originally using SWARM in Objective-C, but was rewritten in its entirety in MASON in order to take advantage of various MASON control and inspection features. The individual performing the port had no previous knowledge of MASON at all, but reported that the port was was fairly easy as MASON has a similar scheduling mechanism as SWARM. Figure 7 shows before-and-after screenshots of two of the many Anthrax panels.

### 5.5. Wetlands: a Model of Memory and Primitive Social Behavior

How does group memory affect sociality? Most computational multi-agent social simulation models are designed with agents lacking explicit internal information-processing structure in terms of basic cognitive elements. In particular, memory is usually not explicitly modeled. In the MASON Wetlands model, (earlier called Floodland [9]), we presented initial results from memory experiments designed to investigate the effect of group memory structures and interaction situations on emergent patterns of sociality or collective intentionality. Specifically, we used the Wetlands model to carry out initial computational experiments conducted on culturally-differentiated agents endowed with finite and degradable memory that simulate bounded mnemonic function and forgetfulness. Our main initial findings thus far are that memory capacity and engram retention both pro-

mote sociality among groups, probably as nonlinear (inverse) functions [3].

Groups of agents look for food, which is generated by a moisture layer in the simulated landscape, and seek shelter when they get too wet. However, since agents consume energy as they live and move around, this creates a dynamic system of forces towards sources of food and shelter. In addition, groups of the same culture share information about food and shelter location, in order to mimic some minimal social in-group vs. out-group behaviors.

We are planning a number of future experiments: the memory structure of agents can be designed with richer structure and functionality, to mimic group memory. Group-level effects, such as groupthink and risky shift, are possibilities we will explore.

## 6. Conclusion

In this paper we presented MASON, a multi-agent simulation library written in Java. MASON is fast, portable, has a small core, and produces guaranteed replicable results. MASON is also designed to completely separate the model from the visualization dynamically, or to reattach it, to migrate the simulation to another platform in the middle of a run, and to provide visualization in 2D or in 3D. We also showed five applications of MASON highlighting the wide applicability of the toolkit. Two of the applications are ports of previous simulation models from Ascape and SWARM.

We plan to position MASON as principled foundation for future multi-agent simulation systems to build upon. MASON is free open source under a BSD-style license, and is available at http://cs.gmu.edu/∼eclab/projects/mason/

## Acknowledgements

Our thanks to Ken De Jong and Jayshree Sarma for their assistance in the development of the paper. Thanks also to MASON developers: Gabriel Catalin Balan wrote much of the 3D code, and Daniel Kuebrich wrote applications and Quicktime support. Thanks also to application writers for their assistance: the Network Intrusion model was written by Elena Popovici, the Anthrax model was written by Jayshree Sarma and Elena Popovici, the CTO model was written in part by Gabriel Catalin Balan, and the Wetlands model was written by Sean Paus.

## References

[1] R. Beckers, O. E. Holland, and Jean-Louis Deneubourg. From local actions to global tasks: Stigmergy and collective robotics. In *Artificial Life IV: Proceedings of the International Workshop on the Synthesis and Simulation of Living Systems , third edition*. MIT Press, 1994.

[2] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Santa Fe Institute Studies in the Sciences of Complexity. Oxford University Press, 1999.

[3] Claudio Cioffi-Revilla, Sean Paus, Sean Luke, James Olds, and Jason Thomas.

[4] J. L. Deneubourg, S. Goss, N. Franks, A. Sendova-Franks, C. Detrain, and L. Chretien. The dynamics of collective sorting: Robot-like ants and ant-like robots. In *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, pages 356–363. MIT Press, 1991.

[5] Sean Luke. ECJ 11: A Java evolutionary computation library. http://cs.gmu.edu/∼eclab/projects/ecj/, 2004.

[6] Sean Luke, Keith Sullivan, Gabriel Catalin Balan, and Liviu Panait. Tunably decentralized algorithms for cooperative target observation. Technical Report GMU-CS-TR-2004-1,Department of Computer Science, George Mason University, 2004.

[7] Liviu Panait and Sean Luke. Evolving ant foraging behaviors. In *Proceedings of the Ninth International Conference on the Simulation and Synthesis of Living Systems (ALIFE9)*, 2004.

[8] Liviu Panait and Sean Luke. A pheromone-based utility model for collaborative foraging. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS-2004)*, 2004.

[9] Sean Paus. Floodland: A simple simulation environment for evolving agent behavior. Technical report, Department of Computer Science, George Mason University, Fairfax, 2003.