NetLogo: Design and Implementation of a Multi-Agent Modeling Environment

Seth Tisue
seth@tisue.net

Uri Wilensky
uri@northwestern.edu

Center for Connected Learning and Computer-Based Modeling
Northwestern University, Evanston, Illinois

Presented at SwarmFest, Ann Arbor, May 9–11, 2004

1 Introduction

In this paper we examine design and implementation choices we made when creating NetLogo [Wilensky, 1999], a multi-agent programming language and modeling environment for simulating complex phenomena [Wilensky, 2001]. Although NetLogo is used across a wide range of education levels from grade school up, we focus here on NetLogo as a tool for research and for teaching at the undergraduate level and higher.

Our earlier paper on NetLogo as a research tool [Tisue & Wilensky, 2004] is recommended background reading, especially if you are not already familiar with NetLogo. It includes a history of NetLogo's origins, a tour of the NetLogo interface, an introduction to the NetLogo language, evidence of acceptance of NetLogo in the research community, and a list of ongoing NetLogo-centered projects in our research group. That paper is less technical, this paper more so.

2 Audience

NetLogo derives from our experience with our earlier environment, StarLogoT [Wilensky, 1997]. Even though the original incarnation of StarLogo [Resnick & Wilensky, 1993, Resnick, 1994] was on a supercomputer, it had always been primarily intended for use in schools.[1] But StarLogoT became very popular among researchers. So with NetLogo, we now aim more explicitly to satisfy the needs of both audiences.

All the multi-agent Logos have adopted design principles from the Logo language [Papert, 1980]. A central principle is "low threshold, no ceiling." Low threshold means new users, including those who never programmed before, should find it easy to get started. No ceiling means the language shouldn't be limiting for advanced users. We wanted NetLogo to be just as popular with researchers as StarLogoT had been, so that meant devoting significant attention to the "no ceiling" side of the principle. Logo's reputation as a language for schools doesn't do justice to its ample power, as demonstrated in [Harvey, 1997].

We believe researchers should care about "low threshold" too. Even for such users, NetLogo's inheritance from educational languages brings several benefits. First, in universities there is substantial overlap between teaching and research, and if a single tool can serve both needs synergy should result.

[1] There were several different early implementations of StarLogo in the first part of the 1990's. The supercomputer version was Connection Machine StarLogo. Later came MacStarLogo [Begel, 1999], of which StarLogoT is a superset.

Second, when code is easier to write and easier to read, everyone benefits. Models become easier to build—often researchers can write models themselves when otherwise they would have to hire a programmer. And models become more easily understood by others; this is vitally important in order for researchers to effectively communicate their results to others, verify each other's results, and build upon each other's work. The goals of scientific modeling are compromised if programs are long, cryptic, and platform-specific. A NetLogo model is less likely to suffer these problems than one written in common general-purpose languages like Java and C++.

NetLogo is its own programming language, embedded in an integrated, interactive modeling environment. The integrated approach to multi-agent modeling originates with StarLogo, was refined in StarLogoT and NetLogo, and has also been followed by other all-in-one agent-based modeling solutions such as AgentSheets [Repenning, Ioannidou & Zola, 2000] and Breve [Klein, 2002]. "Toolkits" or libraries such as Swarm [Minar, Burkhart, Langton & Askenazi, 1996] and Repast [Collier & Sallach, 2001] take a different approach; they make simulation facilities available to programs written in a general-purpose language such as Java.

We see the integrated approach as essential to achieving our "low threshold" goal. The difficulty of programming in Java or C++ isn't due only to the language itself. It's also due to the complication of the environments (whether command line based or GUI based) in which programming in those languages is normally done. When you add in the added complexity of getting the environment to talk to a modeling library or toolkit, the initial barrier for entry for new programmers becomes quite high—even before they start dealing with the difficulties of the languages themselves.

In contrast, the NetLogo environment allows a smooth, almost unnoticeable transition from exploring existing models into programming. NetLogo's user interface makes no firm distinction between using a model and editing it. Even the smallest amount of knowledge of the language is immediately useful in creating buttons and monitors or typing commands into the command center, in order to better inspect and control an existing model. Altering the model's rules is only as far away as a click on the Procedures tab.

3 StarLogoT

StarLogoT succeeded in attracting a large user base from a range of disciplines, but it had important technical limitations that we wanted to address.

The biggest limitation of StarLogoT was that it only ran on Macintosh computers. At the time development on StarLogoT's precursors began, the introduction of Java had not yet brought cross-platform development of GUI applications within easy reach. Also, the target audience was schools, so the software needed to be compact and fast enough to run even on hardware that by today's standards was absurdly underpowered. Putting thousands of agents on such machines was only possible if the underlying engine was written in assembly language, which is of course platform-specific.

The need to be fast and small caused other limitations as well. StarLogoT's decimal arithmetic was fixed point, not floating point, with only a few digits of precision. Many arbitrary limits were imposed in order for crucial data structures to fit within a small, fixed number of bits. For example, a model couldn't have more than 16,384 turtles, or a patch grid bigger than 251x251, or a stack depth of more than 64.

StarLogoT's language design was constrained as well by what could reasonably be implemented. The need for efficiency led StarLogoT's architecture to become quite complicated. It included three different virtual machines for our three agent types (observer, turtles, and patches). Different agent types had different capabilities and different rules for acting in parallel; this was confusing to users and some of the restrictions placed on user programs were severe.

4 Starting over

Because of these limitations, we chose to start over and write our new environment, NetLogo, from

scratch in Java. We bet that Java would permit us to write a cross-platform application that was reasonably fast. Java doesn't always completely live up to its "write once, run anywhere" promise, but it does so enough of the time that it brought cross-platform development within reach for our small development team. We knew that Java was slower than assembly language, but hoped that on newer, faster machines it wouldn't matter too much. (See below for a fuller discussion of speed.)

Using Java offered the additional benefit that individual NetLogo models could be embedded in web pages and run in a browser, without the end user needing to download and install an application. (Initially, we even allowed the full NetLogo environment to run as an applet in a web browser, although we later abandoned this option as not worth the extra development effort.)

Since we were starting from scratch anyway, we took the opportunity to redesign the language to further both our "low threshold" and "no ceiling" goals. Sometimes we had to weigh tradeoffs between those two goals; in other cases, we could reduce barriers to novice entry yet also make the language more expressive and powerful. In doing so, we also tried to be compatible with standard, popular Logo implementations whenever possible and reasonable. In particular, we tried not to stray too far from StarLogoT, so our existing user base wouldn't find the transition too painful.

NetLogo's design was also driven not only by the need to support the construction of models, but also to support what we call "participatory simulations" [Wilensky & Stroup, 1999a], in which a group of students acts out the behavior of a system, each student playing the role of an individual element of the system. To enable this, NetLogo includes a technology called HubNet [Wilensky & Stroup, 1999b], which enables communication between a NetLogo model operating as a server and a set of clients, which may be handheld devices or computers running HubNet client software.

We began development in 1999. Since then averaged we've averaged two to three new releases per year. The first beta version came in 2000, the first numbered version (1.0) in early 2002, version 2.0 at the end of 2003, and version 2.0.1 in spring 2004. Version 2.0.1 is mature, stable, and reliable. Even though our user base has expanded greatly, the rate of incoming bug reports has slowed to a trickle.

5 Language

As a language, NetLogo adds agents and concurrency to Logo. Logo, as originally developed by Seymour Papert and Wally Feurzeig in 1968, is derived from Lisp, but has a friendlier syntax. Logo was designed as a programming language usable by children as well as adults and is still popular today for that purpose. It is a powerful general-purpose computer language. Although there is no single agreed upon standard for the Logo language, NetLogo shares enough syntax, vocabulary, and features with other Logos to earn the Logo name.

Our earlier paper [Tisue & Wilensky, 2004] outlines the basics of the NetLogo language. We offer additional details here.

Some important differences from most Logos include:

- We have no symbol data type. Eventually, we may add one, but since it is seldom requested, it may be that the need doesn't arise much in agent-based modeling. In most situations where traditional Logo would use symbols, we simply use strings instead.

- Control structures such as `if` and `while` are special forms, not ordinary functions. You can't define your own special forms.

- As in most Logos, functions as values are not supported. Most Logos provide similar functionality, though, by allowing passing and manipulation of fragments of source code in list form. NetLogo's capabilities in this area are presently limited. A few of our built-in special forms use UCBLogo-style "templates" to accomplish a similar purpose, for example, `sort-by [length ?1 < length ?2] string-list`. In some circumstances, using `run` and `runresult` instead is workable, but they operate on strings, not lists.

There are several reasons for those omissions. They are partly due to NetLogo's descent from StarLogoT, which as discussed above needed to be very lean. Many of StarLogoT's limitations have already been addressed in NetLogo (for example, NetLogo has agentsets and double-precision floating point math), but some of the "leanness" remains. This leanness is not only historical, though. Efficiency is always a vital goal for multi-agent systems, since many modelers want to do large numbers of long model runs with as many agents as they can. It is easiest to construct a fast engine for a simple language, and, from a language design perspective, omitting advanced language features and prohibiting the definition of new special forms may actually be desirable for a language in which readability and sharing of code is paramount. We weigh these tradeoffs carefully as we continue to expand the language.

For further information on the NetLogo language, consult the NetLogo User Manual [Wilensky, 1999], particularly the Programming Guide and Primitives Dictionary sections.

6 Java upgrade

NetLogo is written in Java. Java was chosen because both the core language and the GUI libraries are cross-platform, and because modern Java virtual machines have use JIT (just in time) compiler technology to achieve relatively high performance.

NetLogo 1.3 supported earlier Java versions going back to Java 1.1, but for NetLogo 2.0 we decided to require Java 1.4. The major reasons for choosing Java 1.4 for the new version were as follows:

- The new language version includes much richer libraries. It was increasingly difficult to find developers used to working within the limitations of the antiquated version.
- More recent VM's are higher quality. Before we abandoned Java 1.1, constantly working around bugs in the various 1.1 VM's was a serious drag on our development efforts. For instance, we were never able to get the interface builder fully working on Linux.
- Unlike Java 1.1, Java 1.4 offers "strict" math libraries which guarantee identical, reproducible results cross-platform.
- Leaving Java 1.1 behind allowed us to switch GUI toolkits, from the old AWT toolkit to the newer Swing toolkit, which has numerous advantages, including better look & feel (Figure 1).
- After a long wait, Apple finally released a high quality Java 1.4 implementation for Mac OS X.
- Even with the new VM, Apple's support for AWT-based applications on Mac OS X was poor. Mac support is important to us, but a high quality implementation on the Mac was simply impossible without switching to Swing.
- Since Java 1.4 is available for all the major platforms for which 1.3 is also available (not counting Mac OS X 10.0 and 10.1), it seemed unnecessary to be backwards compatible with Java 1.3.

Regrettably, switching to Java 1.4 meant dropping support for users of Windows 95 and MacOS 8 and 9, since no Java 1.4 implementation is available for those operating systems. However, we continue to offer support and bugfixes for NetLogo 1.3, so those users aren't left out in the cold.

7 Speed

Early versions of NetLogo were slow, but especially since version 1.3, models run much faster. Most of our users now find NetLogo fast enough for most purposes. Nonetheless, we plan to continue to improve NetLogo's speed, since as mentioned above agent-based modeling is a field in which users always want more speed.

StarLogoT was partially written in assembly language and was highly performance tuned. NetLogo is written in Java and the NetLogo language is much more flexible and feature rich than StarLogoT. Therefore, you would expect NetLogo to be slower. Surprisingly, that isn't always or even usually true. Which environment is faster depends on the nature of the

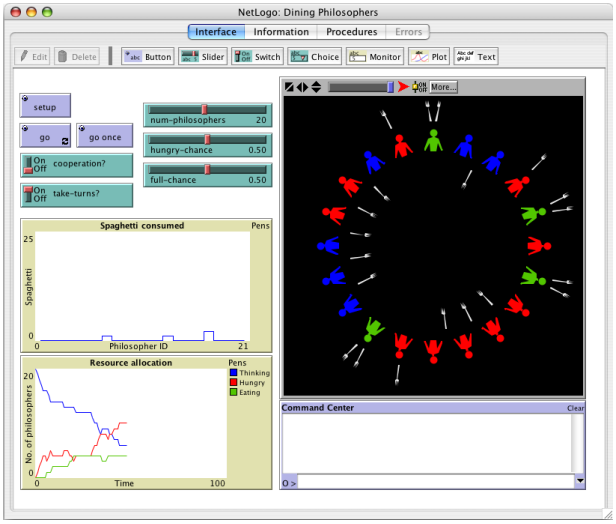

Figure 1: Our new, Swing-based user interface. Also illustrates new graphics features.

model. In general, StarLogoT is still faster for models with simple code and large numbers of agents. But NetLogo is usually faster for models with complex code and smaller numbers of agents.

The surprising fact that StarLogoT is not always faster can be accounted for by reference to StarLogoT's unique architecture. As mentioned above, the StarLogoT engine was divided into three virtual machines: one for the observer, written in Lisp, and two for the turtles and patches, written in assembly language. The turtle and patch machines were extremely fast, but crossing the boundaries between the different machines was slow. With simpler code and more turtles and patches, overall speed benefited more from the speed of the turtle and patch virtual machines. In contrast, NetLogo's internal architecture is much more uniform. A single virtual machine handles all three agent types. Therefore, there is no special penalty associated with complex code and no special benefit associated with large numbers of agents.

NetLogo is a hybrid compiler/interpreter. To improve performance, we don't interpret the user's code directly. Instead, our compiler analyzes, annotates, and restructures it into a form that can be interpreted more efficiently.

Earlier versions of NetLogo (1.0 and 1.1) compiled user code into a form suitable for execution by a virtual machine which was stack-based. However, we discovered through profiling that making the virtual machine stack-based actually hurt performance rather than helping it. So, in our current compiled representation, each command is tree-structured so that intermediate results are stored on the Java VM's own stack instead of our stack. (We still have a stack, but it is used by only a few commands.) This change resulted in an approximately twofold performance gain. Other, smaller engine performance gains since NetLogo 1.0 came from profiling the engine code and addressing inefficiencies in object creation, memory usage, and other areas.

If we want to further increase NetLogo's speed in the future, the most promising approach, relative to the likely development effort required, seems to be to compile NetLogo code to Java byte code instead of our own custom intermediate representation. Informal tests indicate that this would likely result in at least a twofold improvement in speed. We also have considered replacing the Java-based engine with a native one, perhaps written in C. However, general opinion recently is that JITted Java code isn't always slower than C code anymore, so we're not certain if this approach would be fruitful.

So far we have been discussing the speed of NetLogo's core computational engine. But NetLogo's overall performance doesn't depend only on engine speed. There's also graphics speed to consider. Whether engine speed or graphics speed dominates varies widely from model to model—some are 90% engine, others are 90% graphics. The latter kind of model can always be sped up by using NetLogo's graphics "control strip" to temporarily shut off graphics altogether, but that doesn't mean graphics performance is unimportant.

Switching our GUI framework from AWT to Swing raised problems for graphics performance. Prior to NetLogo 2.0, graphics window updates were "incremental," that is to say, only agents that moved or changed were redrawn. Incremental painting on-screen, instead of to an offscreen buffer, is not supported under Swing, and on Mac OS X, the perfor-

mance of painting offscreen was unacceptable. As an experiment, we switched from incremental painting to always redrawing the complete contents of the graphics window every time, fearful that the change would hurt performance. We were pleasantly surprised; on Macs graphics performance actually increased, and on Windows, the speed penalty was negligible.

Abandoning incremental updates freed NetLogo's graphics capabilities enormously. Previously, in order to make incremental updates possible, the graphics window was limited in several important respects. Even though NetLogo's world is continuous, turtles in the graphics window were always the same size and appeared centered on their patches, like chess pieces. Since patches did not overlap, it was possible to redraw each patch incrementally and separately. But if incremental updates are no longer performed, then there is no longer any reason to align turtles with the grid. So now, in NetLogo 2.0, turtles can be any size and shape and be positioned anywhere. Turtles and patches can also be labeled with text. Turtle shapes are vector-based to ensure smooth appearance at any scale. (These features had been available in earlier NetLogo versions, but were slow and buggy. Now they are fast and reliable.) These changes have led to dramatic visual enhancement of models (Figure 1, Figure 2).

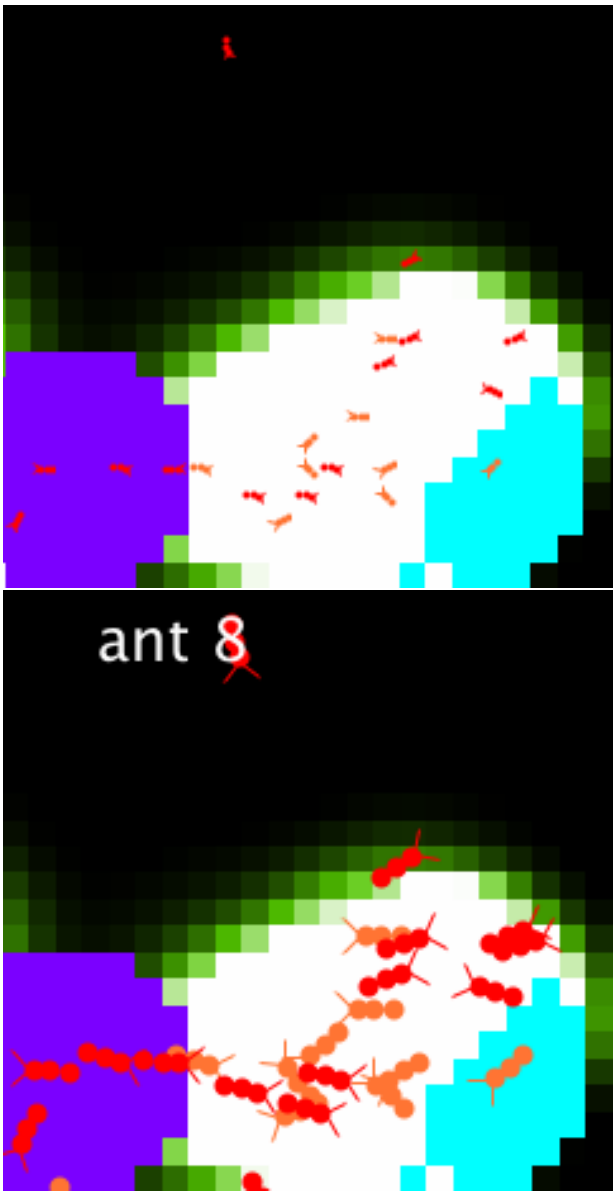

Figure 2: The Ants model, with and without new graphics features.

8 Concurrency

In many respects the engine is an ordinary interpreter. But it also has some unusual features because of the need to support concurrent processes. Concurrency in NetLogo has two sources.

The first kind of concurrency we support is concurrency among agents. If you use the command `forward 20` to ask a set of turtles to move forward 20 steps, we don't want one turtle to win the race before the others have even left the starting line. So, we have all the turtles take one step together, then they all take another step, and so forth. Ultimately, the NetLogo engine is single-threaded, so the turtles must move one at a time in some order; they can't really move simultaneously. So the engine "con-

text switches" from agent to agent after each agent has performed some minimal unit of work, called a "turn." Because the timing of context switches is deterministic, the overall behavior of the model remains deterministic. We only update the screen after all the agents have had a turn; this visually preserves the illusion of simultaneity. The NetLogo User Manual contains a more detailed discussion of the timing of context switches between agents. We provide a command, `without-interruption`, which the programmer can use to prevent unwanted switching.

The second kind of concurrency we support is concurrency among the different elements of the NetLogo user interface which can initiate the execution of code. Currently these are: buttons, monitors, and the Command Center. Buttons and monitors contain code entered by the model author, and the user may enter commands into the Command Center at any time. In all three cases, a "job" is created and submitted to the engine to request that some code be executed by some agents. Jobs are akin to what operating systems call "threads" or "processes." We use the word "job" to avoid confusion. At the operating system level, the NetLogo application is *one* process, and the NetLogo engine is *one* thread within that process.

When multiple jobs are active, the engine must switch between them, just as it switches between the agents within a job. The rule followed is to switch from job to job once every agent in the first job has had a turn. Here, the NetLogo engine is taking on a task more typically associated in computer scientists' minds with the process scheduler in a cooperatively multi-tasked operating system rather than with a language interpreter.

Concurrency is still an active area of concern for us. We're not sure we've arrived at final decisions on how best to support it. We're presently revisiting and rethinking our current design choices with an eye towards both helping newcomers avoid mistakes and increasing the power available to advanced users.

9 Extensibility

At one time, NetLogo was a closed platform. Users couldn't alter or extend it, or control it from external code. This is now changing—NetLogo is becoming extensible. It has always been a full-fledged programming language, so users may write procedures in NetLogo and then use them just like built-in commands. But now in NetLogo 2.0.1, we have an application programmer's interface (API) for extensions so that users can add new elements to the language by implementing them directly in Java. For example, you might let agents make sounds and music using Java's MIDI capabilities, or communicate with remote computers, and many other things. We have been using this new API internally for a while now, and have written extensions that let NetLogo:

- Talk to other NetLogos running on different computers, peer-to-peer
- Pull down data from a web server
- Make sounds using MIDI

Now that the API is in the hands of actual users, we hope that feedback from them will help guide further development.

We also offer a "controlling" API which allows external code to operate the NetLogo application by remote control, so to speak. This API includes calls for opening a model and running any NetLogo commands. This permits users willing to do a little light Java programming to automate large numbers of model runs from the command line. This is useful both on a single machine and when distributing runs across a cluster. (We already provide an automated parameter-sweeping tool called BehaviorSpace, but the API will still be useful in situations where BehaviorSpace's present capabilities aren't sufficient.)

In making NetLogo extensible, we are bridging the gap between integrated modeling environments (easy to use, but potentially restricting) and modeling toolkits (more flexible, but much harder to use). Extensions lift the "ceiling" on NetLogo's usefulness and range of applications. The integrated NetLogo environment provides core functionality; our APIs

will allow advanced users to move outside that core. Extension authors can share their extensions with the user community, so that everyone can benefit from their efforts.

Earlier, we described NetLogo as an "all in one" environment. The full NetLogo environment bundles together many components: a programming language, a compiler, an interpreter, a syntax highlighting editor, an interface builder, a graphics engine, BehaviorSpace, and so on. The downside of the all-in-one approach is that "all in one" can turn into "all or nothing." We run the risk that if one component doesn't suit a user's needs, then that user won't be able to use any of the components, because they're all tied together.

We want to avoid this all-or-nothing trap by letting users extend or replace parts of NetLogo that don't suit their purposes. That way even users who have unique needs, or just needs we didn't think of or haven't gotten around to addressing yet, can build what they need themselves in Java, and they will still get the benefit of the rest of our work. These new APIs are steps towards that goal.

10 Models Library

Just as important as NetLogo itself is the materials it comes with. We've devoted almost as much development effort to our Models Library as to the NetLogo application.

The Models Library contains more than 140 prebuilt simulations that can be explored and modified. All of the models include an explanation of the subject matter and the rules of the simulation and suggestions for activities, experiments, and possible extensions. To aid learning and encourage good programming practice, the code for the simulations is well commented and as elegantly written as we can make it.

Our goal for the library is to include as many as possible of the standard, well-known "chestnuts" of complex systems science. This serves several purposes:

- Researchers, already knowing the ideas behind the models, can easily learn the language by studying them.
- Modelers can usually find something in the library to base a new model on, rather than starting from scratch.
- These well-known examples are introduced to a new generation of students of complex systems science.

The Models Library also includes a "curricular models" section. It contains groups of models that are intended to be used together in an educational setting as part of a curricular unit. Most of them include extra associated curricular materials (above and beyond that which we provide with all of our models).

In addition to the 140 simulations, the library also includes several dozen "code examples." These are not full simulations, but brief demonstrations of NetLogo features or coding techniques.

11 Conclusion

We have already touched upon some goals for future NetLogo versions, such as increased speed and greater extensibility.

Here are some other enhancements for which we already have working prototypes:

- 3-D NetLogo, including language extensions and 3-D graphics. Some 3-D models are already possible, but language support will make them easier to build and OpenGL will enable much higher quality 3-D visualization. This is a very big job, but we have a working prototype already (see Figure 3).
- Support for different lattices and world topologies. with no extra code required. Currently, the NetLogo patch world "wraps" in the X and Y directions, forming a torus. Some language elements are available in both wrapping and non-wrapping versions. Typically, models that don't want wrapping use the outer layer of patches as a barrier. In a future version, we plan to make wrapping a global option which can be

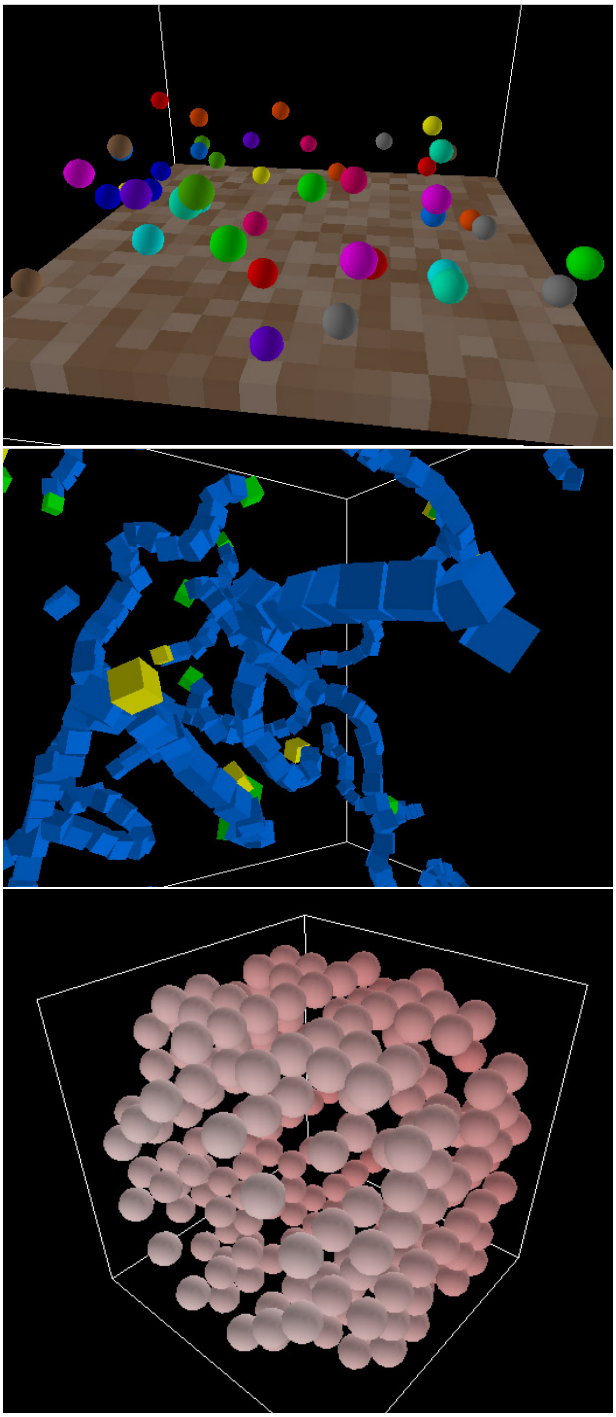

Figure 3: Some screen captures of our prototype 3-D version of NetLogo.

turned off. This is an example of an alternate world topology. Soon, we will also support even-numbered grid sizes and arbitrary placement of the origin of the coordinate plane. In the longer term, we would like to support unbounded plane models. We already have some models that operate on a hexagonal lattice, but their code is not as concise as we would like.

- Improved editor for turtle shapes, to make it easier to customize the look of models. This is important for data visualization. The existing editor is serviceable, but limited.

- Parenthesis and bracket matching in the code editor, to make editing complex code easier.

- Easier, more flexible randomized agent scheduling. (Random scheduling is already possible by adding extra code, but will be built in.)

We have also begun work on the following:

- Detecting individual keystrokes from code. This will make highly interactive models more usable—games, too.

- Improved plotting requiring less additional code in the procedures tab. Separating code for agent behaviors from code for data generation and visualization code will improve clarity and conciseness of models.

- Adding `let` to the language, so new local variables can be introduced anywhere. This will help modelers write clearer, more concise code.

Networks are currently a very active area of research in the agent-based modeling community. Network models are already possible in NetLogo, but we want to make them easier to build, including making it easier to leverage the capabilities of existing network analysis and visualization tools.

We also want to integrate NetLogo with aggregate modeling engines, so that researchers and students can investigate systems using agent-based and aggregate techniques in tandem.

12 Acknowledgments

Thanks to the members of our user community and to all past and present members of the Center for Connected Learning and Computer-Based Modeling, for all of their contributions to NetLogo.

We gratefully acknowledge the support of the National Science Foundation.

References

[Begel, 1999] Begel, Andrew, "StarLogo: Building a Modeling Construction Kit for Kids", Proceedings of the Workshop on Agent Simulation: Applications, Models, and Tools.

[Collier & Sallach, 2001] Collier, N. & Sallach, D.; Repast. University of Chicago. http://repast.sourceforge.net/

[Harvey, 1997] Harvey, Brian, *Computer Science Logo Style*, 2nd ed., vols. 1–3, MIT Press.

[Klein, 2002] Klein, Jon, "Breve: a 3D simulation environment for the simulation of decentralized systems and artificial life", Proceedings of Artificial Life VIII, the 8th International Conference on the Simulation and Synthesis of Living Systems, MIT Press.

[Minar, Burkhart, Langton & Askenazi, 1996] Minar, Nelson; Burkhart, Roger; Langton, Chris; and Askenazi, Manor; "The Swarm Simulation System: A Toolkit for Building Multi-agent Simulations." Santa Fe Institute working paper 96-06-042.

[Papert, 1980] Papert, Seymour, *Mindstorms: Children, Computers, and Powerful Ideas*, Basic Books.

[Repenning, Ioannidou & Zola, 2000] Repenning, Alexander; Ioannidou, Andri; and Zola, John; "AgentSheets: End-User Programmable Simulations", Journal of Artificial Societies and Social Simulation vol. 3, no. 3.

[Resnick, 1994] Resnick, Mitchel, *Turtles, Termites and Traffic Jams: Explorations in Massively Parallel Microworlds*, MIT Press.

[Resnick & Wilensky, 1993] Resnick, Mitchel; and Uri Wilensky, "Beyond the Deterministic, Centralized Mindsets: New Thinking for New Sciences", American Educational Research Association.

[Tisue & Wilensky, 2004] Tisue, Seth; and Uri Wilensky, "NetLogo: A Simple Environment for Modeling Complexity", International Conference on Complex Systems.

[Wilensky, 1997] Wilensky, Uri, StarLogoT, Center for Connected Learning and Computer-Based Modeling, Northwestern University. http://ccl.northwestern.edu/cm/starlogot/

[Wilensky, 1999] Wilensky, Uri, NetLogo (and NetLogo User Manual), Center for Connected Learning and Computer-Based Modeling, Northwestern University. http://ccl.northwestern.edu/netlogo/

[Wilensky, 2001] Wilensky, Uri, "Modeling Nature's Emergent Patterns with Multi-agent Languages", EuroLogo.

[Wilensky & Stroup, 1999a] Wilensky, Uri; and Stroup, Walter; "Learning through Participatory Simulations: Network-based Design for Systems Learning in Classrooms", Proceedings of the Computer Supported Collaborative Learning Conference, Stanford University.

[Wilensky & Stroup, 1999b] Wilensky, Uri; and Stroup, Walter; HubNet. Center for Connected Learning and Computer-Based Modeling, Northwestern University. ccl.northwestern.edu/netlogo/hubnet.html