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 Agent 2004, Chicago, October 2004

adapted from:

Abstract

NetLogo [Wilensky, 1999] is a multi-agent programming language and modeling environment for simulating complex phenomena. It is designed for both research and education and is used across a wide range of disciplines and education levels. In this paper, though, we focus on NetLogo as a tool for research and for teaching at the undergraduate level and higher. We outline the principles behind our design and describe recent and planned enhancements.

1   Overview

NetLogo is a multi-agent programming language and modeling environment for simulating complex natural and social phenomena. It is particularly well suited for modeling complex systems evolving over time. Modelers can give instructions to hundreds or thousands of independent "agents" all operating concurrently, in order to explore connections between micro-level behaviors of individuals and macro-level patterns that emerge from their interactions.

NetLogo enables users to open simulations and "play" with them, exploring their behavior under various conditions. NetLogo is also an authoring environment that is simple enough to enable students and researchers to create their own models, even if they are not professional programmers.

We designed NetLogo for both education and research. There has been considerable research on the use of multi-agent modeling in K–12 settings (e.g. [Wilensky, 1995] [Resnick, 1996] [Wilensky & Resnick, 1999] [Ionnidou et al., 2003] [Wilensky, 2003] [Wilensky & Reisman, in press]). In this paper, though, we focus on NetLogo as a powerful research tool and as a tool for learners at the undergraduate level and higher.

Historically, NetLogo is the next generation of the series of multi-agent modeling languages including StarLogo [Resnick & Wilensky, 1993] [Resnick, 1994]. NetLogo is a standalone application

1

written in Java so it can run on all major computing platforms. After five years of development, NetLogo is a mature product that is stable and reliable. It is freeware—anyone can download it for free and build models without restriction. It comes with extensive documentation and tutorials and a large collection of sample models.

As a language, NetLogo is a member of the Lisp family that supports agents and concurrency. Mobile agents called "turtles" move over a grid of "patches," which are also programmable agents. All of the agents can interact with each other and perform multiple tasks concurrently.

NetLogo is being used to build an endless variety of simulations. Members of our user community have turned turtles into molecules, wolves, buyers, sellers, bees, tribespeople, birds, worms, voters, passengers, metals, bacteria, cars, robots, neutrons, magnets, planets, shepherds, lovers, ants, muscles, networkers, and more. Patches have been made into trees, walls, terrain, waterways, housing, plant cells, cancer cells, farmland, sky, desks, fur, sand, you name it. Turtles and patches can be used to visualize and study mathematical abstractions, too, or to make art and play games. Themes addressed include cellular automata, genetic algorithms, positive and negative feedback, evolution and genetic drift, population dynamics, path-finding and optimization, networks, markets, chaos, self-organization, artificial societies and artificial life. The models all share our core themes of complex systems and emergence.

In the following sections, we offer more detail on all of these topics. We begin with a tour of the application, then back up to outline its history. We then give a more detailed account of the language itself. NetLogo has recently become extensible; we explain why and how. A technical discussion of how NetLogo is implemented follows. Finally, we conclude with a summary of work in progress and future plans.

2 Tour

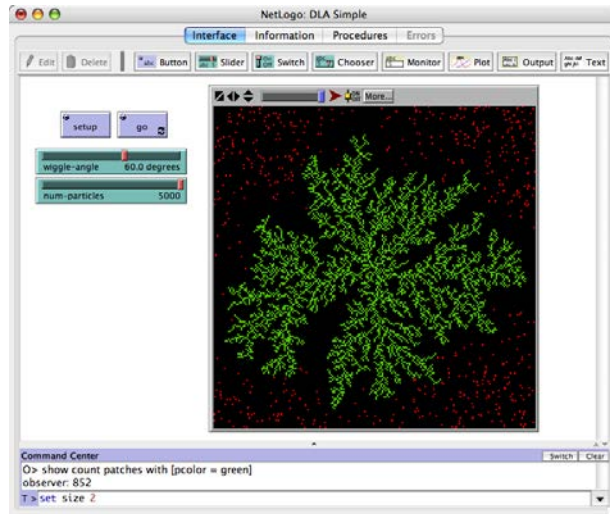In this section we give the reader a brief tour of the NetLogo user interface and Models Library.

Figure 1: NetLogo's user interface, with model Diffusion Limited Aggregation

2.1 User interface

Figure 1 shows NetLogo's user interface after opening and running a model from the Models Library.

On the right is the graphics window, in which the "world" of the model is made visible. In the model shown, the turtles represent diffusing particles. They wander randomly. When the model begins, there is a single green patch in the center. When a particle encounters a green patch, it "sticks" and turns green itself. Over time a beautiful, branching aggregate emerges.

On the left are model controls. In this model, they include:

- Buttons for controlling the model. "Setup" initializes the model and "Go" makes it run.

- Sliders that control model parameters. For example, the "num-particles" slider control the number of particles that build the aggregate.

Note that this is a simple model with only a few controls. For more complicated models, other types of controls are available including switches, choosers, monitors, plots, text boxes, and output areas.

In this screen shot, we see only NetLogo's "Interface" tab. The Interface tab is also an interface builder. No firm distinction is made between using a model and editing it—you can move, modify, or create interface elements at any time. Agents can be inspected and altered and the code for the model can be changed without restarting the simulation. At the bottom of the Interface tab is the "Command Center," in which NetLogo commands can be issued, even while the model is running.

The other tabs are:

- Information, where documentation on the model is found. This typically explains the rules behind the model and suggests experiments for the reader to try.

- Procedures, where the actual code for the model is stored. A well written model includes comments in the code explaining how it works.

- Errors (normally disabled), where any incorrect code can be viewed and fixed.

The order of the tabs is meant to follow a user's typical engagement with a model. Usually people want to dive right in and try out the model first in the Interface tab, then move to the Information tab to more fully understand what they're seeing. Eventually, they can inspect the code in the Procedures tab to understand the underlying rules and make modifications and additions.

Figure 2 shows the Procedures tab containing the complete code for the model. Language elements are automatically color-coded so the code's structure is more clearly visible.

NetLogo can exchange data with other applications. The language includes commands that let you read or write any kind of text file. There are also facilities for exporting and importing data in standard formats. The complete state of the world can be saved and restored in a format that can easily be opened and analyzed with other software. Graphed data can be exported for rendering and analysis with other tools. The contents of the graphics window, or of the model's whole interface, can be saved as an image, or you can record a series of such images as
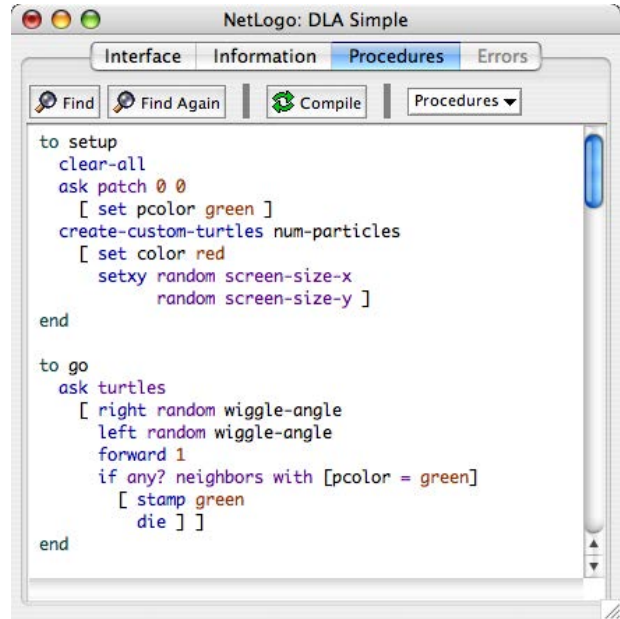

Figure 2: Procedures tab with complete code for the aggregation model.

a QuickTime movie. Finished models can be published on the web or embedded in presentations as Java applets.

NetLogo includes a still evolving tool called BehaviorSpace that allows "parameter sweeping," that is, systematically testing the behavior of a model across a range of parameter settings. Figure 3 shows an example of using BehaviorSpace to study a forest fire model. Based on the experiment setup entered by the user, BehaviorSpace automatically runs the model many times while varying the "density" parameter. The results show the effect of that parameter on the amount of forest burned.

NetLogo supports not only the construction of wholly computer-based simulations, but also 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

Figure 4: Nodes and edges, both represented using turtles in the graphics window.
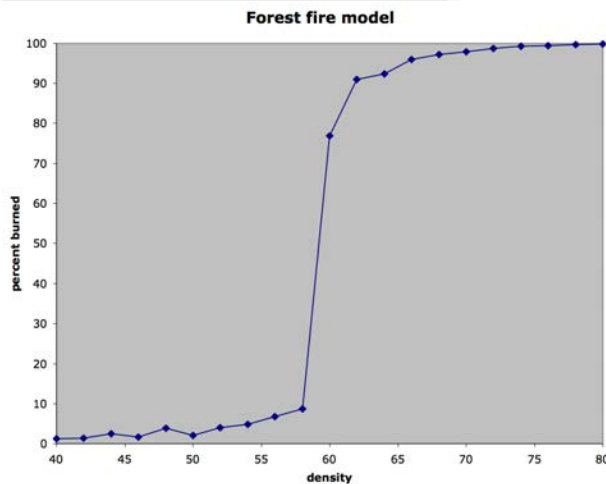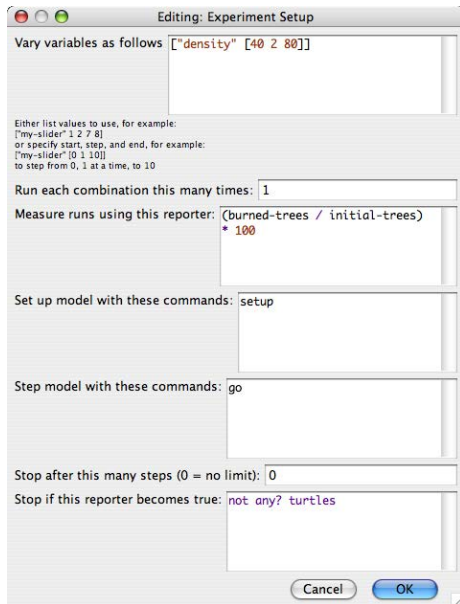

Figure 3: Using BehaviorSpace to study a forest fire model. The density slider is varied from 40 to 80 by steps of 2. We measure the percentage of burned trees at the end of each run. A run ends when no "fire" agents remain. The graph at the bottom (generated from the BehaviorSpace output by means of a graphing packgage) shows the results: an abrupt phase transition at the critical density.
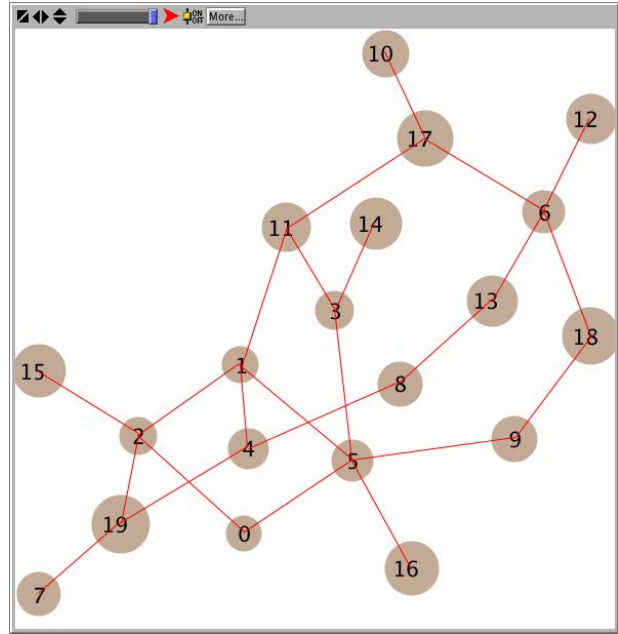
operating as a server and a set of clients, which may be handheld devices or computers running HubNet client software.

The most visible area of change in NetLogo 2.0 was graphics. Now, 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 changes have led to dramatic visual enhancement of models. An example of graphics that weren't possible before is the use of turtles to represent both nodes and edges in a network as in Figure 4.

Significant improvements made for the NetLogo 2.1 release include:

- Improved editor for turtle shapes, to make it easier to customize how a model looks. This is important for data visualization. See Figure 5.

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

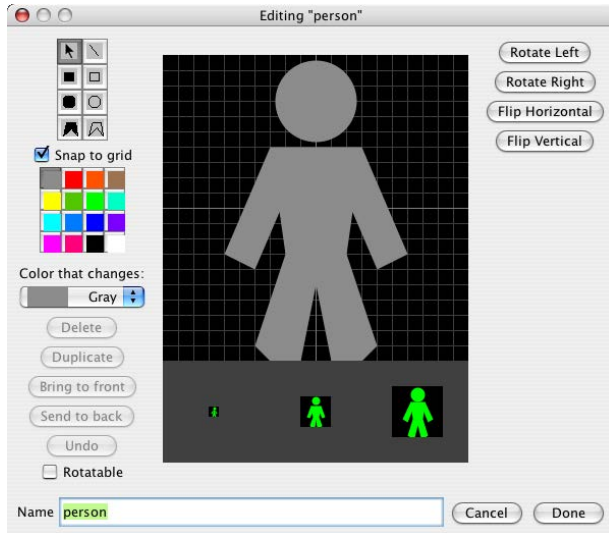- Detecting individual keystrokes from code. This

4

Figure 5: New, improved editor for turtle shapes.

makes highly interactive models (and games) more usable.

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

2.2 Models Library

Just as important as NetLogo itself are 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 150 prebuilt simulations that can be explored and modified. Figure 6 shows the structure of the Models Library. The simulations address many content areas in the natural and social sciences, including biology and medicine, physics and chemistry, mathematics and computer science, and economics and social psychology. 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 clear, elegant, and well commented.
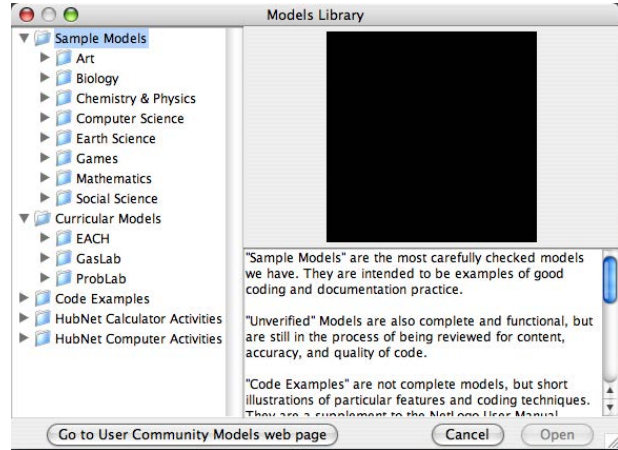

Figure 6: Structure of the Models Library.

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.

5

3   History and audience

In this section we summarize NetLogo's history and how it came to be a tool for both education and research, and we explain the benefits of addressing both audiences.

3.1   Origins

NetLogo originates in a blend of StarLisp [Lasser & Omohundro, 1986] and Logo [Papert, 1980]; Logo is itself a member of the Lisp family. From Logo, it inherits the "turtle." In traditional Logo, the programmer controls a single turtle; a NetLogo model can have thousands of them. NetLogo also follows Logo's philosophy of ease of use, providing a "low threshold" of entry for new users. From StarLisp, a parallel Lisp of the 1980's, NetLogo inherits multiple agents and concurrency.

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. In the transition from StarLogoT to NetLogo, we redesigned both the language and the user interface. NetLogo includes almost all of StarLogoT's features and many new ones. Many of the new features of NetLogo are aimed at research users.

3.2   "Low threshold"

All the multi-agent Logos have adopted design principles from the Logo language. 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 pop-

---

[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.

ular 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 there are opportunities for synergy. Second, when code is easier to write and easier to read, everyone benefits. Models become easier to build; often researchers can build models themselves when otherwise they would have to hire programmers. 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++.

3.3   The integrated approach

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

6

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.

3.4 Development history

NetLogo has been under development since 1999. Since then we've averaged two to three substantial new releases per year. Version 2.0.2 (August 2004) is mature, stable, and reliable. As of October 2004 version 2.1 is available in beta form and we expect a final release soon. Even though our user base has expanded, the rate of incoming bug reports has slowed to a trickle. Models now run much faster than in earlier versions—our users now find it fast enough for most purposes.

3.5 Acceptance

We have much evidence that acceptance of NetLogo in the research and education communities is wide and growing. The software has been downloaded tens of thousands of times. Currently, there are about 50 downloads per day. Our announcements list has over 5,000 members. The NetLogo discussion group (http://groups.yahoo.com/group/netlogo-users/) has over 1,600 members and averages about 100 posts per month. Traffic on the discussion group has increased fivefold since 2002. Several organizations have independently conducted workshops on NetLogo for both researchers and teachers. In the summer of 2004, we held our own first annual workshop at Northwestern. A number of university classes are now taught, in whole or in part, using NetLogo. Some of these classes and workshops have rich collections of associated materials available online. The NetLogo web site has an area where users can upload models to share with the user community. More than 100 models have been uploaded so far.

4 Language

In this section, we describe the NetLogo programming language itself. For further information on the NetLogo language, consult the NetLogo User Manual [Wilensky, 1999], particularly the Programming Guide and Primitives Dictionary sections.

4.1 Language fundamentals

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 Logo isn't limited to graphical applications, it is best known for its "turtle graphics," in which a virtual being or "turtle" moves around the screen drawing figures by leaving a trail behind it. NetLogo generalizes this concept to support hundreds or thousands of turtles all moving around and interacting. The world in which the turtles move is a grid of "patches," which are also programmable. Collectively, the turtles and patches are called "agents". All agents can interact with each other and perform multiple tasks concurrently. NetLogo also includes a third agent type, the "observer". There is only one observer. In most models, the observer gets the ball rolling by issuing instructions to the turtles and patches. Different "breeds" of turtle may be defined, and different variables and behaviors can be associated with each breed.

Some models use the patch world just as a lattice. For example, in a cellular automaton, there are

7

no turtles, only patches. And in some other models, turtles move on the lattice (from patch center to patch center). But the patches are not just lattice sites—they are square sections of a continuous two-dimensional space. Turtle coordinates are floating point values, so a turtle may be positioned anywhere within a patch. For example, in the aggregation model shown above, the aggregate is made up of lattice sites, but particles move freely on the plane.

There are many language elements for talking about space and spatial relations: `towards`, `distance`, `neighbors`, `forward` and `back`, `left` and `right`, `size`, `heading`, `patch-ahead`, `diffuse`, and so on. Some of these come from Logo, while others are new.

An important NetLogo language feature, not found in its predecessors, is "agentsets," or collections of agents. For example, the set of all turtles and the set of all patches are agentsets. You can also make custom agentsets on the fly, for example the set of all red turtles, or a column of patches (the set of patches with a given X coordinate). Agentsets are responsible for much of NetLogo's expressive power.

In addition to special constructs to support multi-agent modeling, NetLogo also includes standard programming constructs such as procedures, loops, conditionals, recursion, strings, lists, and so forth. Both integer math and double-precision IEEE floating point math are supported. The `run` and `runresult` commands can be used to execute code constructed on the fly.

4.2   NetLogo as Logo

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. Still, 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.

4.3   Reproducibility

One of our core design goals for NetLogo is that results be scientifically reproducible, so it is important that models operate deterministically. NetLogo is a "simulated parallel" environment. In true parallel computing, programs must be constructed very carefully to avoid nondeterminism. We think this is too great a burden for novice programmers, so concurrency in NetLogo operates deterministically. That means that if you "seed" the random number

generator the same way, then a NetLogo model always follows the same steps in the same order and produces the exact same results, regardless of what computer you run it on. Java's underlying platform-independent math libraries help assure consistency.

5 Extensibility

In this section, we describe how NetLogo has recently become extensible through the addition of new "extensions" and "controlling" facilities.

Earlier, we described NetLogo as an integrated or "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. They 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.

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).

5.1 Extensions API

NetLogo 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 since NetLogo 2.0.1 we have offered an application programmer's interface (API) for extensions so that users can add new elements to the language by implementing them directly in Java. This lets users add whole new types of capabilities to NetLogo.

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 and music using MIDI

The sound extension is now included with NetLogo. Full Java source code for it, and a number of other sample extensions, are available from our website. Our hope is that extension authors will share their extensions with the wider user community, so that everyone can benefit from their efforts.

5.2 Controlling API

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 is still be useful in situations where BehaviorSpace's present capabilities aren't sufficient.

The API currently requires the full NetLogo user interface to be present, but we are working on removing this limitation so that models can be run "headless" from the command line. (On X11-based systems, it is possible right now to work around this limitation using X11's 'virtual framebuffer" support.)

6 Implementation

In this section, we explain how how we have constructed the NetLogo software. This section is more technical than the others.

6.1 Background: 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 resulted in other limitations as well. Math in StarLogoT 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.

6.2 Starting over

Because of these limitations, we chose to start over and write our new environment, NetLogo, from scratch. We bet that Java would permit us to build a cross-platform application that was reasonably fast. Java doesn't always completely live up to its "write once, run anywhere" promise, but comes close 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. The issue of speed is discussed further below.

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 authoring environment to run as an applet in a web browser, but later we 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, such as agentsets, we were able to reduce barriers to novice entry while also making 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.

6.3 Java

NetLogo is written entirely 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.

- 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 7).

- 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.

6.4   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 agent-based modeling is a field in which users always benefit from more speed.

StarLogoT was written partially 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 model. In general, StarLogoT is still faster for models with very simple code and large numbers of agents.
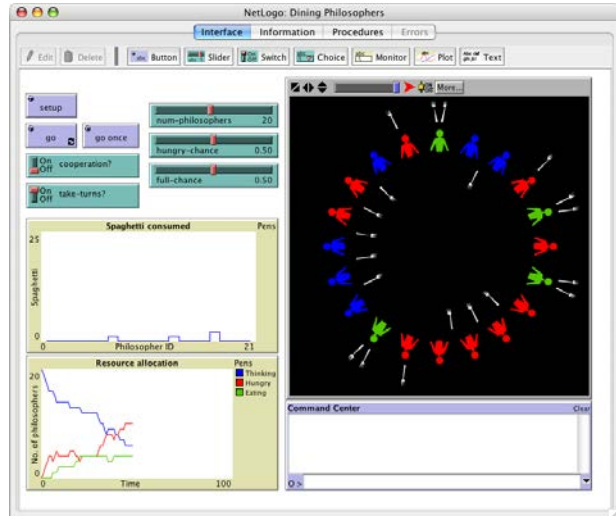

Figure 7: Our new, Swing-based user interface. Also illustrates new graphics features.

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

11

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. 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 performance 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 actually 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 7, Figure 8).

6.5 Concurrency

In many respects the NetLogo 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 "context 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
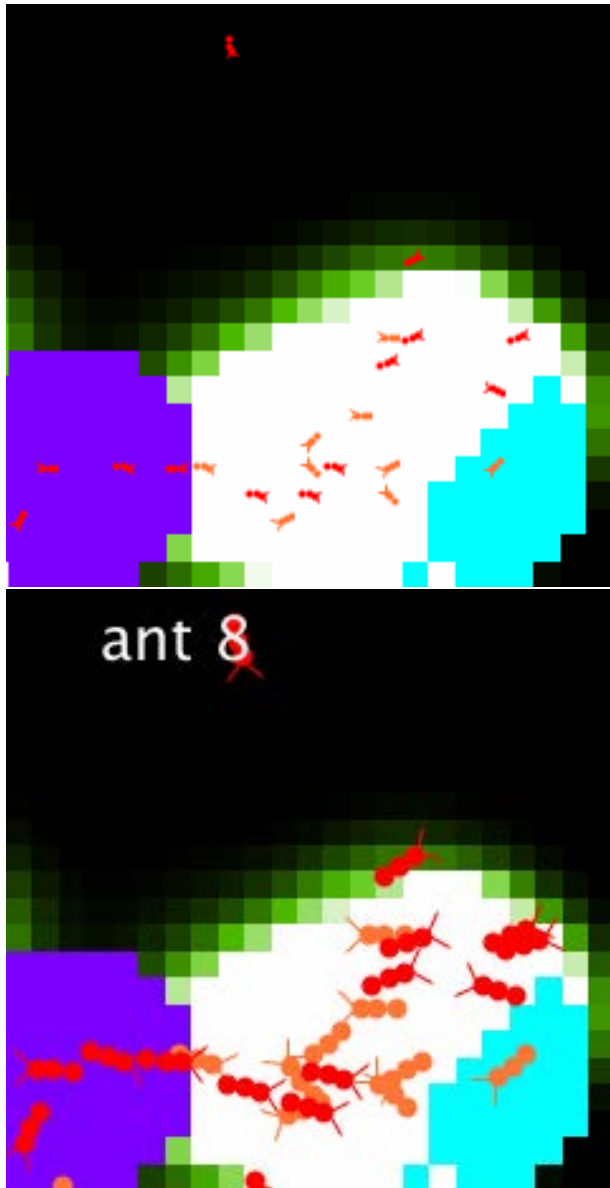
Figure 8: The Ants model, with and without new graphics features.

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.

7   Conclusion

We have already touched upon some goals for future NetLogo versions, such as increased speed and headless operation. Here are some other enhancements for which we already have working prototypes:

- 3-D NetLogo, including language extensions and

13

OpenGL-based 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 9).

- 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 nonwrapping 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 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 can't currently be made as concise as we would like.

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

- 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.

- A profiling tool for identifying speed bottlenecks in model 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 are also adding support to NetLogo for aggregate modeling. Aggregate modeling, also known as systems dynamics modeling, has historically been
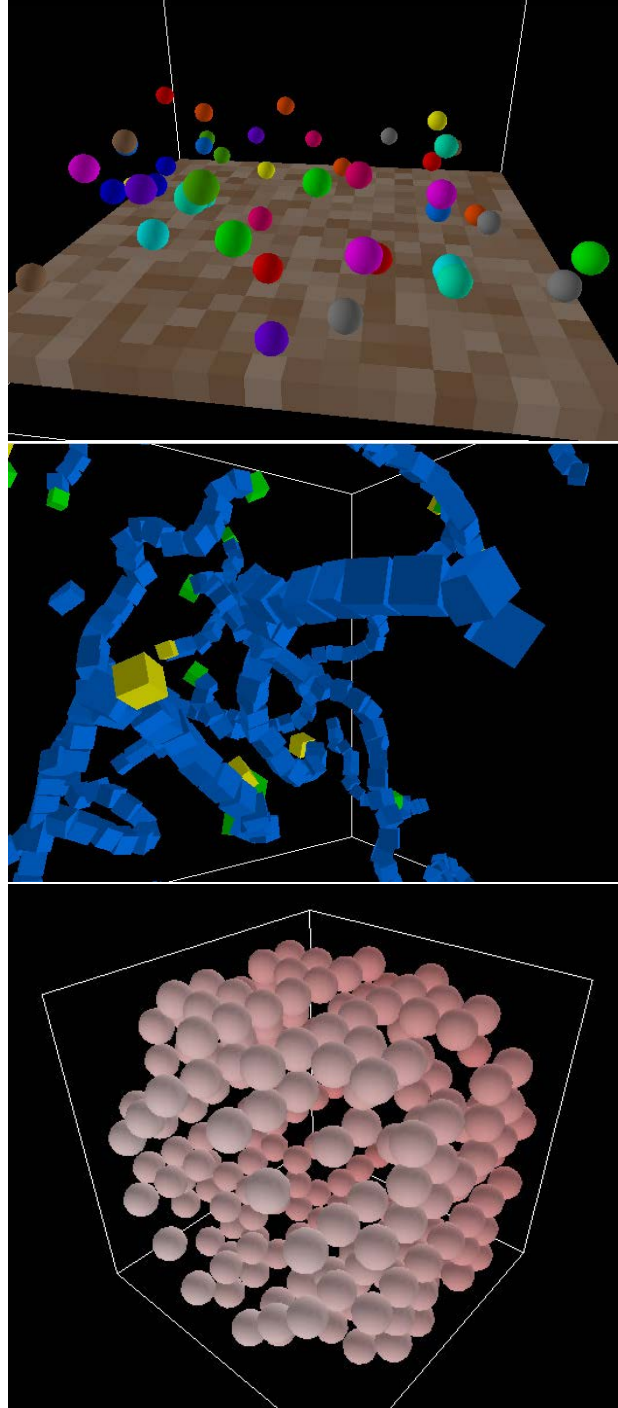

Figure 9: Some screen captures of our prototype 3-D version of NetLogo.

14

supported by separate, non-agent-based modeling tools such as STELLA [Richmond & Peterson, 1990]. We are incorporating similar finite difference engine technology into NetLogo so that researchers and students can investigate systems using agent-based and aggregate techniques in tandem.

There are ongoing efforts within our research group to further explore NetLogo's potential for research and education. Of particular relevance to NetLogo's future as a research tool are these major ongoing long-term projects:

- Integrated Simulation and Modeling Environments (ISME), a project in collaboration with the University of Texas which uses NetLogo to enact "participatory simulations" [Wilensky & Stroup, 1999a] in both classroom and research contexts.

- Procedural Modeling of Cities, a project in which agents "grow" virtual cityscapes for use in architecture, urban planning, training, and entertainment. Preliminary results from the model are shown in Figure 10 [Lechner, Watson, Wilensky & Felsen, 2003].

- Modeling School Reform, a project to build models of the potential effects of educational policy decisions, to assist school leaders and policy makers. This work will include social network modeling and analysis.

These projects will drive substantial expansion of NetLogo's ability to support large, ambitious modeling efforts. We also have a number of other projects, focused on the use of NetLogo in educational contexts.

8   Acknowledgments
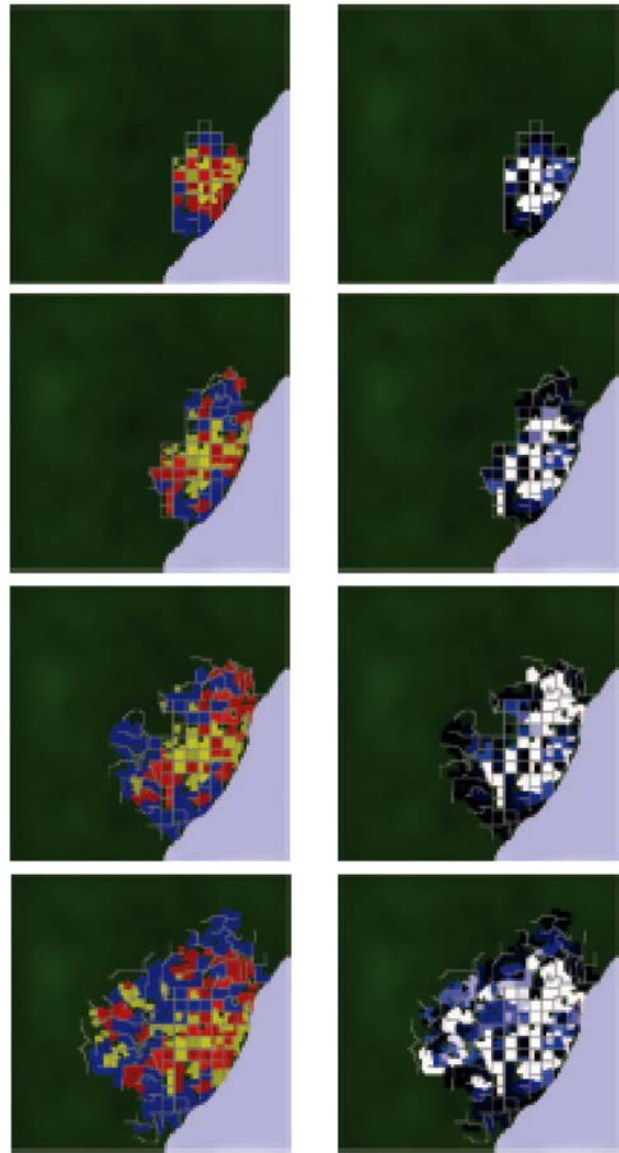
Figure 10: Growth of a simulated city. Left column represents land use, right column represents population density.

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.

[Ionnidou et al., 2003] IOANNIDOU, Andri; REPENNING, Alexander; LEWIS, Clayton; CHERRY, Gina; and RADER, Cyndi; "Making Constructionism Work in the Classroom", International Journal of Computers for Mathematical Learning, Volume 8, Issue 1, pp. 63-108.

[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.

[Lasser & Omohundro, 1986] LASSER, Clifford; OMOHUNDRO, Stephen M.; *The Essential Starlisp Manual*, Thinking Machines Corporation.

[Lechner, Watson, Wilensky & Felsen, 2003] LECHNER, Thomas; WATSON, Ben; WILENSKY, Uri; and FELSEN, Martin. "Procedural Modeling of Land Use in Cities". Proceedings of Midgraph.

[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, 1996] RESNICK, Mitchel, "Beyond the Centralized Mindset", Journal of the Learning Sciences 5(1): 1-22.

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

[Richmond & Peterson, 1990] RICHMOND, Barry; and PETERSON, Steve. STELLA II. High Performance Systems, Inc.

[Wilensky, 1995] WILENSKY, Uri, "Paradox, Programming and Learning Probability", Journal of Mathematical Behavior Vol. 14, No. 2, p 231-280.

[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, 2002] WILENSKY, Uri, "Modeling Nature's Emergent Patterns with Multi-agent Languages", Proceedings of EuroLogo 2002.

[Wilensky, 2003] WILENSKY, Uri, "Statistical Mechanics for Secondary School: The GasLab Multi-Agent Modeling Toolkit", International Journal of Computers for Mathematical Learning, Volume 8, Issue 1, special issue on agent-based modeling.

[Wilensky & Reisman, in press] WILENSKY, Uri; and REISMAN, Kenneth; "Thinking Like a Wolf, a Sheep or a Firefly: Learning Biology through Constructing and Testing Computational Theories", Cognition & Instruction.

[Wilensky & Resnick, 1999] WILENSKY, Uri; and RESNICK, Mitchel; "Thinking in Levels: A Dynamic Systems Approach to Making Sense of the World", Journal of Science Education and Technology, vol. 8, no. 1.

[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.
http://ccl.northwestern.edu/netlogo/hubnet.html