Chapter 6
StarLogo TNG: Making Agent-Based Modeling Accessible and Appealing to Novices

Eric Klopfer, Hal Scheintaub, Wendy Huang, and Daniel Wendel

Computational approaches to science are radically altering the nature of scientific investigatiogn. Yet these computer programs and simulations are sparsely used in science education, and when they are used, they are typically "canned" simulations which are black boxes to students. StarLogo The Next Generation (TNG) was developed to make programming of simulations more accessible for students and teachers. StarLogo TNG builds on the StarLogo tradition of agent-based modeling for students and teachers, with the added features of a graphical programming environment and a three-dimensional (3D) world. The graphical programming environment reduces the learning curve of programming, especially syntax. The 3D graphics make for a more immersive and engaging experience for students, including making it easy to design and program their own video games. Another change to StarLogo TNG is a fundamental restructuring of the virtual machine to make it more transparent. As a result of these changes, classroom use of TNG is expanding to new areas. This chapter is concluded with a description of field tests conducted in middle and high school science classes.

6.1 Computational Modeling

New skills and knowledge will be required to successfully navigate the complex, interconnected, and rapidly changing world in which we live and work. Older paradigms alone are not sufficient to handle these challenges. The field of complexity science is providing the insights scientists and engineers need to push our thinking in new directions [16, 28], providing the tools to move beyond the reductionist point of view and look at whole systems [31, 4, 10]. This approach has been driven by advances in computer technology that have facilitated the development of advanced multi-agent simulations. This powerful new modeling approach has been called the third way of doing science,

with traditional experimentation and observation/description being the other two [16]. The increasing importance of computational modeling has changed the knowledge base required to practice science. Scientists in all fields now rely on a variety of computer skills, from programming computer simulations, to analysis of large-scale datasets, to participation in the globally networked scientific communities.

While this explosion in technology and the increased importance of modern paradigms has radically altered the landscape of science and engineering practice, these changes have not yet impacted the way high school students learn and perform science. We believe that computational science and simulation can and should modernize secondary school science courses by making these tools, practices, and habits of mind an integral part of the classroom experience. Although the mere use of computer simulations may aid in understanding both the scientific concepts and the underlying systems principles, a deeper understanding is likely to be fostered through the act of creating models. These principles center on the idea that we must create a culture in which students can navigate novel problem spaces and collaboratively gather and apply data to solutions. Resnick [20] suggested that an equally important goal is for educational programs to foster creativity and imagination. In other words, students should be acquiring habits of mind that will not only enable them to address today's problems and solutions but will also allow them to venture into previously unimagined territories. To achieve this, they should have a facility with tools to create manifestations of their ideas so that they can test them and convey their ideas to others.

6.1.1 Simulations and Modeling in Schools

Science has long relied on the use of scientific models based on ordinary differential equations (ODEs) that can be solved (in simple cases) without computers. These models describe how aggregate quantities change in a system, where a variable in the model might be the size of a population or the proportion of individuals infected by a disease. The mathematics required for these models is advanced, but several commonly used modeling programs like Model-It 1.1 [29], Stella [24], and MatLab [15] have graphical interfaces that make them easy to construct. These programs have become very popular in the classroom as well as the laboratory. The user places a block on the screen for each quantity and draws arrows between the blocks to represent changes in those quantities. While this interface does not remove the need for the user to learn math, it lowers the barrier for entry. However, the abstraction required to model these systems at an aggregate level is a difficult process for many people, which often limits the utility of this modeling approach [33].

With the increase of accessibility to the Internet in schools, teachers are turning to the World Wide Web for visual and interactive tools in the form

of Java applets or Flash applications to help students better understand important scientific processes. However, most of these on-line applications are simulation models whose rules and assumptions are opaque to students.

Additionally, many of the systems that are studied in the classroom are more amenable to simulation using agent-based modeling. Rather than tracking aggregate properties like population size, agent-based models track individual organisms, each of which can have its own traits. For instance, to simulate how birds flock, one might make a number of birds and have each bird modify its flight behavior based on its position relative to the other birds. A simple rule for the individual's behavior (stay a certain distance away from the nearest neighbor) might lead to a complex aggregate behavior (flocking) without the aggregate behavior being explicitly specified anywhere in the model. These emergent phenomena, where complex macro-behaviors arise from the interactions of simple micro-behaviors, are prevalent in many systems and are often difficult to understand without special tools.

6.1.2 Computer Programming

Computer programming is an ideal medium for students to express creativity, develop new ideas, learn how to communicate ideas, and collaborate with others. The merits of learning through computer programming have been discussed over the years [8, 11, 12, 18]. As a design activity, computer programming comes along with the many assets associated with "constructionist" [9] learning. Constructionism extends constructivist theories, stating that learning is at its best when people are building artifacts. These benefits of learning through building include [23]:

1. Engaging students as active participants in the activity
2. Encouraging creative problem-solving
3. Making personal connections to the materials
4. Creating interdisciplinary links
5. Promoting a sense of audience
6. Providing a context for reflection and discussion

Other learning outcomes from programming are more broadly applicable. Typical programming activities center on the notion of problem-solving, which requires students to develop a suite of skills that may be transferable to other tasks. As students learn to program algorithms, they must acquire systematic reasoning skills. Modeling systems through programs teaches students how to break down systems into their components and understand them. Many problems require students to acquire and apply mathematical skills and analysis to develop their programs and interpret the output. Together these skills comprise a suite of desirable learning outcomes for students, many of which are difficult to engender through typical curriculum.

They can also form the cornerstones of true inquiry-based science [2], providing students with a set of skills that allows them to form and test hypotheses about systems in which they are personally interested.

6.1.3 Programming in Schools

Computer programming was widely introduced to schools in the early 1980s, largely fueled by inexpensive personal computers and inspired by Papert's manifesto, Mindstorms [17], which advocated that programming in Logo was a cross-cutting tool to help students learn any subject. Logo was popularly used to enable students to explore differential geometry, art, and natural languages. Microworlds Logo [13] extended this use into middle school, where students continued to learn about mathematics, science, reading, and storytelling. KidSim/Cocoa/Stagecast Creator [5] brought modeling and simulation to children through a simple graphical programming interface, where they could learn thinking skills, creativity, and computer literacy. Subsequent work with Stagecast Creator has shown some success in building students' understand of simple mathematical concepts through building games [14]. Boxer [7] is used to help students in modeling physics and mathematical reasoning while building understanding of functions and variables. Kafai [12] introduced novel paradigms for helping students learn about mathematics and social and relational skills through programming of games. Through this paradigm, older elementary school students use Logo to develop mathematical games for younger elementary students. It is of note that most of these efforts have been targeted at elementary-school-aged children. Few efforts have targeted the middle school and high school audience.

Despite many small-scale research efforts to explore programming in schools, Papert's larger vision of pervasive programming in school curricula encountered some serious problems. Programming did not fit into traditional school structures and required teachers to give up their roles as domain experts to become learners on par with the children they were teaching [1]. Computers were used to teach typing skills, and Papert's vision of programming as a vehicle for learning seemed to be dead in the water.

With the growing popularity of the Internet, many schools have revisited the idea of computers in the classroom and rediscovered their utility in the educational process. Schools now use computers to target job-oriented skills, such as word processing and spreadsheets. Some schools encourage children to create mass media, such as computer-based music, animations, or movies. For the last several years, the state of Maine has provided laptops to all of its 7th and 8th grade students (and teachers) to make computers a ubiquitous tool in every subject in school [30].

Now that computers in school are enjoying a resurgence in popularity, it is time to reconsider programming as a vehicle toward fluency in information

technology and advancing science education through building simulation and complex systems models.

6.2 Original Design Criteria

In designing an appropriate modeling tool for use in the K–12 classroom we needed to consider several design criteria. One criterion is the choice of agent-based as opposed to aggregate-based modeling. This approach is not only more amenable to the kinds of models that we would like to study in the classroom, but it is also readily adopted by novice modelers.

The next criterion is to create a modeling environment that is a "transparent box." Many of the simulations that have been used in classrooms to date are selected for the purpose of exploring a specific topic such as Mendelian genetics or ideal gases. Modeling software has been shown to be particularly successful in supporting learning around sophisticated concepts often thought to be too difficult for students to grasp [26, 32]. Such software allows students to explore systems, but they are "black box" models that do not allow the students to see the underlying models. The process of creating models – as opposed to simply using models built by someone else – not only fosters model-building skills but also helps develop a greater understanding of the concepts embedded in the model [21, 25, 32]. When learners build their own models, they can decide what topic they want to study and how they want to study it. As learners' investigations proceed, they can determine the aspects of the system on which they want to focus and refine their models as their understanding of the system grows. Perhaps most importantly, building models helps learners develop a sound understanding of both how a system works and why it works that way. For example, to simulate a cart rolling down an inclined plane in the popular Interactive Physics program, a student could drag a cart and a board onto the screen and indicate the forces that act about each object. In doing so, the student assumes the existence of an unseen model that incorporates mass, friction, gravity, and so forth, calculates the acceleration of the cart, and animates the resulting motion. It would be a much different experience to allow the student to construct the underlying model herself and have that act on the objects that she created.

We also considered the level of detail that we felt would be appropriate in student-built models. All too often, students want to create extremely intricate models that exhaustively describe systems. However, it is difficult to learn from these "systems models" [27]. It is more valuable for students to design and create more generalized "idea models" that abstract away as much about a system as possible and boil it down to the most salient element. The ability to make these abstractions and generalize scientific principles is central to the idea of modeling. For example, a group of students might want to create a model of a stream behind their school, showing each species of insect, fish,

and plant in the stream. Building such a model is not only an extremely large and intricate task, but the resulting model would be extremely sensitive to the vast number of parameters. Instead, the students should be encouraged to build a model of a more generalized system that includes perhaps one animal and one plant species. The right software should support the building of such "idea models."

Based on these criteria, we originally created a computer modeling environment named StarLogo, which we describe in the next section.

6.3 StarLogo

StarLogo is a programmable modeling environment that embraces the constructionist paradigm of learning by building, and it has existed in numerous forms for many years. StarLogo was designed to enable people to build models of complex, dynamic systems. In StarLogo [19], one programs simple rules for individual behaviors of agents that "live" and move in a two-dimensional (2D) environment. For instance, a student might create rules for a bird, which describe how fast it should fly and when it should fly toward another bird. Because StarLogo makes use of graphical output, when the student watches many birds simultaneously following those rules, she can observe how patterns in the system, like flocking, arise out of the individual behaviors. Building models from the individual, or "bird" level, enables people to develop a good understanding of the system, or "flock"-level behaviors (Fig. 6.1 shows the graphical interface for the growth of an epidemic, another phenomenon that can be studied through simulation).

6.3.1 StarLogo Limitations

For years we worked with students and teachers in the Adventures in Modeling (AIM) program, a program which helps secondary school students and teachers develop an understanding of science and complex systems through simulation activities using our StarLogo programming environment. AIM offered computer modeling to science and math teachers as part of their professional development. AIM-trained teachers developed simulations that were integrated into their standard classroom practices. However, activities in which students program their own simulations have been limited to a small number of classrooms. Our research shows that there are several barriers to students developing their own models in science classes (instead of isolated computer classes). These barriers include the time it takes to teach programming basics and teacher and student comfort with the syntax of programming languages.

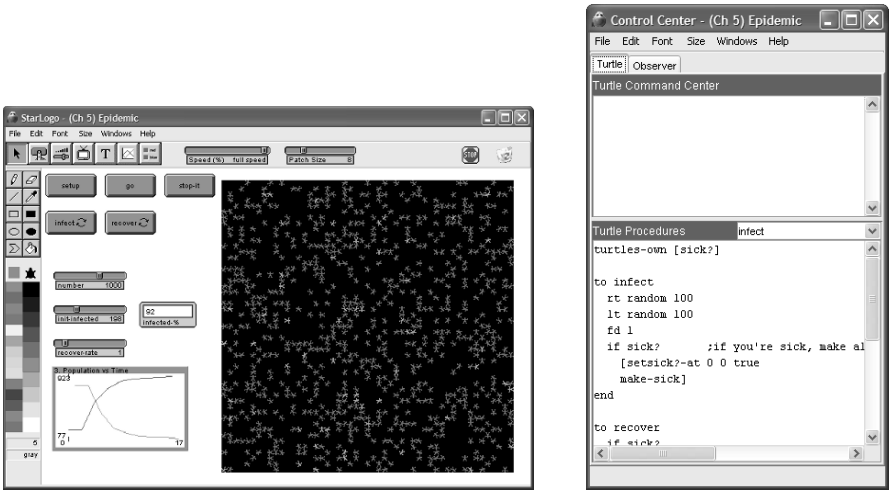

Fig. 6.1 The left-hand panel shows a StarLogo model of an epidemic in action. One can see the interface area, in which users can start and stop the model and adjust parameters, the running model where the different colors code for different states, and the graphs at the bottom tracking model data. The right-hand panel shows the code that was written to generate this model.

These barriers stand in the way of the deeper learning we have observed when students are given the opportunity to develop their own models.

6.4 StarLogo: The Next Generation

We designed StarLogo: The Next Generation (TNG) to address these problems and to promote greater engagement of students in programming. StarLogo TNG provides two significant advances over the previous version. First, the programming is now done with graphical programming blocks instead of text-based commands. This innovation adds a powerful visual component to the otherwise abstract process of programming. The programming blocks are arranged by color based on their function, which enables students to associate semantically similar programming blocks with each other. Since the blocks are puzzle-piece shaped, blocks can fit together only in syntactically sensible ways. This eliminates a significant source of program bugs that students encounter. Blocks snap together to form stacks, which visually help students organize their mental models of the program. StarLogo TNG's interface also encourages students to spatially organize these block stacks into "breed drawers," which affords an object-oriented style of programming. StarLogo TNG's second significant advance is a 3D representation of the agent world. This provides the ability to model new kinds of physical phenomena,

as shown in Section 6.7, and allows students to take the perspective of an individual agent in the environment, which helps them bridge the description of individual behaviors with system-level outcomes.

StarLogo TNG retains the basic language and processes of StarLogo. Many of the favorite StarLogo complex systems simulations have been reprogrammed and included in the TNG download. Now users can see "termites" build 3D mounds in the termite simulation. In the predator-prey simulation called "Rabbits," users can choose from a variety of agent-shapes to play the role of predator and prey; and novice students can build their own African Plains predator–prey simulation in just two class periods.

6.4.1 Spaceland

As people today are inundated with high-quality 3D graphics on even the simplest of gaming platforms, students have come to expect such fidelity from their computational experiences. While the 2D "top-down" view of StarLogo is very informative for understanding the dynamics and spatial patterns of systems, it is not particularly enticing for students. Additionally, students often have difficultly making the leap between their own first-person experiences in the kinesthetic participatory simulation activities and the systems-level view of StarLogo.

Enter "Spaceland," a 3D OpenGL-based view of the world. This world can be navigated via keyboard and mouse controls and an on-screen navigator. In the aerial 3D view (Fig. 6.2), the user can zoom in on any portion of the world for a closer look and view the 3D landscape as they fly by. First-person views include the "Agent Eye" and "Over Shoulder" view (Fig. 6.2). In these views, the user sees out of the "eyes" or over the shoulder of one of the agents in the system. As the agent moves around, the user sees what the agent sees. This works particularly well in seeing how agents bump off of objects or interact with other agents. Shapes can be imported so that the agents in the system can be turtles or spheres or firefighters (Fig. 6.2). The ability to customize the characters provides a sense of personalization, concrete representation, and also of scale. Lending further to the customization capabilities are the abilities to change the "Skybox" background (the bitmap image shown in the distance) as well as the texture, color, and topography of the landscape. Since the top-down view is also still quite useful, that 2D view is seen in an inlaid "mini-map" in the lower-left corner of the screen. The 2D and 3D views can be swapped by toggling the Overhead button.

In runtime mode, users will see not just the 3D representation of the world but also the tools to edit the landscape and the runtime interface (buttons and sliders) with which a user can interact to start and stop the model or change parameters on the fly (Fig. 6.3).

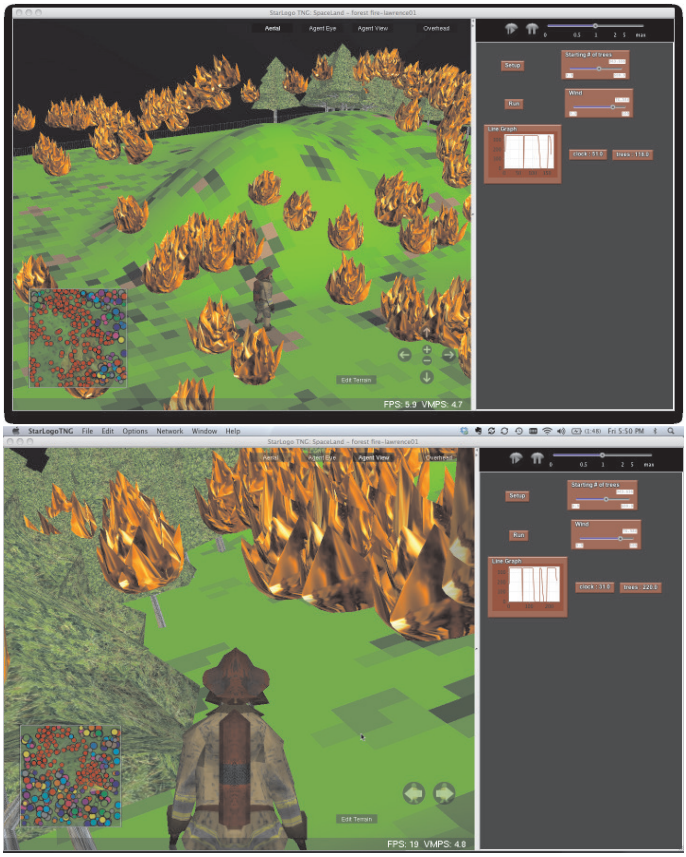

Fig. 6.2 Top-down and turtle's-eye views in TNG.

Additional 3D Benefits

Making it easier to initially engage students through a more up-to-date look, a 3D representation has additional advantages. A 3D view can make the world feel more realistic. This can actually be a limitation in modeling and simulation, in that students feel that more realistic-looking models actually represent the real world, but in the proper context, a more realistic-looking world can be used to convey useful representations of scale and context - for example, conveying the sense of a landscape or the changing of seasons. Moreover, the 3D view of the world is what people are most familiar with from their everyday experiences. Making the leap into programming already comes with a fairly high cognitive load. Providing people with as familiar an environment as possible can lighten that load and make that leap easier. Similarly, a navigable 3D landscape can provide a real sense of scale to a model that otherwise appears arbitrary and abstract to a novice modeler.

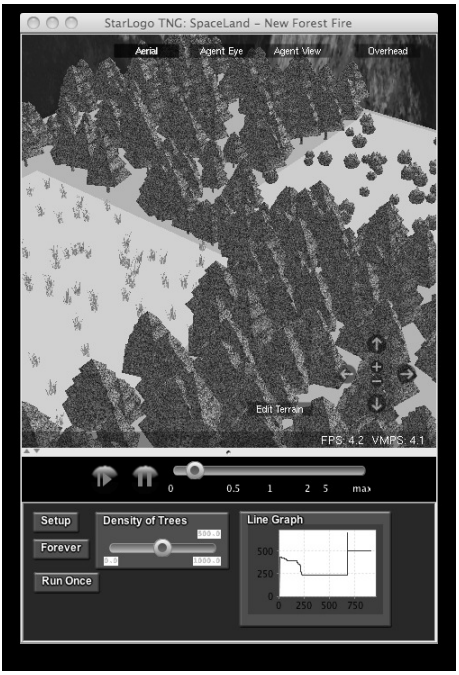

Fig. 6.3 Runtime view.

Finally, with the proliferation of 3D tools (particularly for "modding" games) we can tap into the growing library of ready-made 3D shapes and textures to customize the world.

6.4.2 StarLogoBlocks

StarLogoBlocks is a visual programming language in which pieces of code are represented by puzzle pieces on the screen. In the same way that puzzle pieces fit together to form a picture, StarLogo blocks fit together to form a program. StarLogoBlocks is inspired by LogoBlocks [3] (Fig. 6.4), a Logo-based graphical language intended to enable younger programmers to program the Programmable Brick [22] (a predecessor to the Lego Mindstorms product).

StarLogoBlocks is an instruction-flow language, where each step in the control flow of the program is represented by a block. Blocks are introduced into the programming workspace by dragging a block from a categorized palette of blocks on the left side of the interface. Users build a program by stacking a sequence of blocks from top to bottom. A stack may be topped by a procedure declaration block, thereby giving the stack a name, as well as enabling recursion. Both built-in and user-defined functions can take ar-

gument blocks, which are plugged in on the right. Data types are indicated via the block's plug and socket shapes.

The workspace is a large horizontally oriented space divided into sections: one for setup code, one for global operations, one for each breed of turtles, and one for collisions code between breeds. A philosophy we espouse in the programming environment is that a block's location in the workspace should help a user organize their program. While separating code by section appears to the user as a form of object-oriented programming, in reality the system does not enforce this. All code may be executed by any breed of turtle. Fish can swim and birds can fly, but that certainly should not prevent the user from successfully asking a fish to fly.

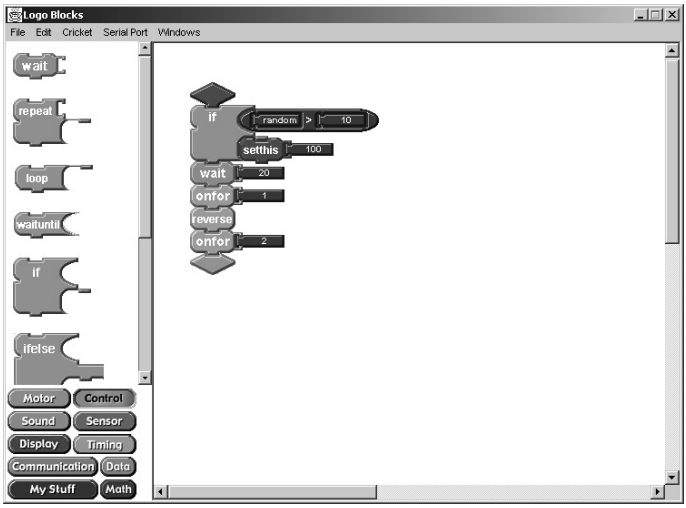

Fig. 6.4 The LogoBlocks interface. A simple program is shown that includes logical statements, random behaviors, and movement. The blocks are assembled into a complete procedure, drawing upon the palette of blocks at the left.

Graphical Affordances

The "blocks" metaphor provides several affordances to the novice programmer. In the simplest sense, it provides quick and easy access to the entire vocabulary, through the graphical categorization of commands. As programmers build models they can simply select the commands that they want from the palette and drag them out. Perhaps more importantly, the shape of the blocks is designed so that only commands that make syntactic sense will actually fit together. That means that the entire syntax is visually apparent to the programmer. Finally, it also provides a graphical overview of the con-

trol flow of the program, making apparent what happens when. For example, Fig. 6.5 shows text-based StarLogo code on the left and the equivalent graphical StarLogoBlocks code on the right for a procedure which has turtles turn around on red patches, decrease their energy, eat, and then die with a 10% chance or continue moving otherwise.

```
to run
 if pc = red
   [ rt 180 ]
 setenergy energy -- 1
 eat
 ifelse (random 100) < 10
   [ die ]
   [ move ]
end
```

Fig. 6.5 StarLogo and StarLogoBlocks representations of the same code.

While the code in StarLogoBlocks takes up more space, it can be seen that the syntax is entirely visible to the user. In contrast, the StarLogo code has mixtures of brackets [ ], parentheses ( ), and spaces that are often confusing to users. The if-else conditional fully specifies what goes in which socket. The if predicate socket has a rounded edge, which can only fit Booleans, such as the condition shown. Similarly, functions which take integers have triangular sockets. The two other two sockets (for "then" and "else") are clearly labeled so that a programmer (or reader of a program) can clearly see what they are doing.

Blocks create more obvious indicators of the arguments taken by procedures, as compared to text-based programming. This enables us, as the software developers, to provide new kinds of control flow and Graphical User Interfaces (GUIs) for the blocks without confusing the programmer. Adding a fifth parameter to a text-based procedure results in children (and teachers) more often consulting the manual, but introducing this fifth parameter as a labeled socket makes the new facility more apparent and easier to use. Graphical programming affords changes in these dimensions because a change in the graphics can be more self-explanatory than in a text-based language.

While StarLogoBlocks and LogoBlocks share the "blocks" metaphor, StarLogoBlocks is presented with much greater challenges due to the relative complexity of the StarLogo environment. LogoBlocks programs draw from a language of dozens of commands, are typically only 10–20 lines long, have a maximum of 2 variables, have no procedure arguments or return values, and

have no breeds. StarLogo programs draw from a language with hundreds of commands and can often be a hundred lines or more long. Screen real estate can become a real limit on program size. Driven by this challenge, we created a richer blocks environment with new features specifically designed to manage the complexity and size of StarLogo code.

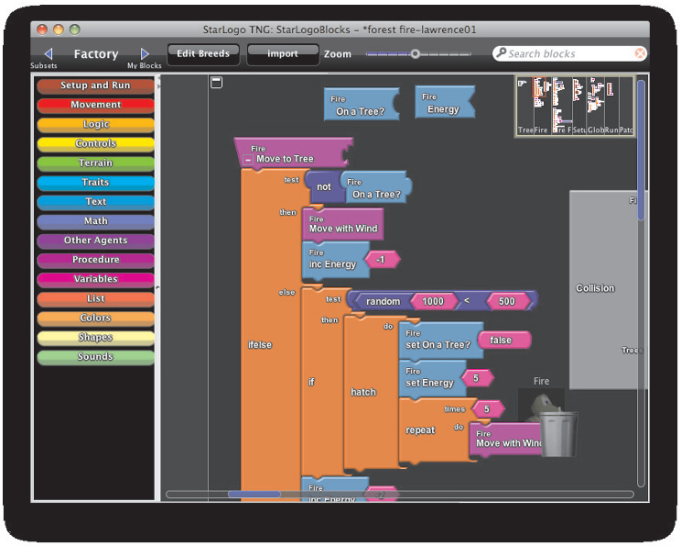

Fig. 6.6 The StarLogoBlocks interface.

One of the most important innovations is to incorporate dynamic animated responses to user actions. We use this animation to indicate what kinds of user gestures are proper and improper while the user is performing them. For example, when a user drags a large stack of blocks into an "if" statement block, the "if" block will stretch to accommodate the stack. See the "Move to Tree" procedure in Fig. 6.6. The first part of the if-else command has two commands to run (Move With Wind and inc Energy), while the else has many (starting with a nested if command inside of it). The blocks expand to fit as many commands as necessary so that the else clause could in theory have dozens of commands.

If a user tries to use a procedure parameter in a different procedure from which the parameter was declared, all of the block sockets in the incorrect stacks of blocks close up. When a user picks up a number block to insert into a list of values (which in Logo may contain values of any type), all block sockets in the list will morph from an amorphous "polymorphic" shape into the triangular shape of a number, to indicate that a number block may be placed in that socket. We plan to continue adding new kinds of animations to help prevent users from making programming errors in the system.

These animations are implemented using vector-based drawing. Vector drawing enables a second important feature: the zoomable interface. Using a zoom slider, the user can zoom the entire interface easily (see the zoom slider in the upper right of Fig. 6.6) to look closely at a procedure they are writing or to expand their view to see an overall picture of their project.

Another innovation has been "collapsible" procedures. Individual procedures can be collapsed and expanded to see or hide their contents. This allows the programmer to build many small procedures and then "roll up" the procedures that they are not using and put them on the side. Fig. 6.6 shows the "Move To Tree" procedures in its unrolled state. The individual commands are visible, but they can be hidden by clicking on the plus sign on the block. Additional advances provide for the easy creation of new procedures, including procedures with an arbitrary number of parameters and variables. Also of note is the ability to enter blocks simply by typing, a method we call "typeblocking." Typeblocking allows for quick entry of blocks, making the construction of sophisticated projects quite rapid for experts.

Together, these innovations lower the barrier of entry for programming, thus facilitating model construction/deconstruction in the context of science, math, or social science classes.

6.5 The StarLogo TNG Virtual Machine

The StarLogo TNG virtual machine (VM) is responsible for executing the commands that make up a StarLogo TNG program. It is at the core of StarLogo TNG, giving virtual life to the agents living in Spaceland and running their StarLogoBlocks programs. The VM has several advancements aimed at making StarLogo TNG programs faster, more flexible, and easier to understand than their StarLogo 2 counterparts.

6.5.1 The Starting Point: StarLogo 2

The StarLogo TNG virtual machine is based on the design of the StarLogo 2 virtual machine, which allows thousands of agents to execute programs in pseudo-parallel, meaning that execution, although serial in actuality, is switched so quickly between agents that it appears to be almost parallel. In StarLogo 2, each agent executes a handful of commands and then the VM switches to the next agent, allowing it to execute a handful of commands, and so on. Each time an agent moves, changes color, or changes any of its state, its visible representation on the screen is updated immediately. Additionally, movement commands such as forward (fd) and back (bk) are broken up into single-step pieces, effectively converting fd 5 into five copies of fd 1. This

causes the execution to appear almost as if it is parallel, as each agent only takes a small step or does a small amount of execution at a time.

6.5.1.1 Synchronization

The design of StarLogo 2's VM is effective for many types of programs, but it also leads to several conceptual difficulties stemming from the problem of synchronization. In any pseudo-parallel system where two agents interact, synchronization takes place when the agents momentarily enter the same frame of time. For example, if two agents meet and each decides to take the color of the other, one of them will in actuality go first, leading to unexpected behavior. In the colors example, if a red and blue agent meet, and the red one goes first, it will take the color of the other, thereby becoming blue itself. The blue agent, then, because it executes second, will take the color of its now blue partner, producing a net result different from what is expected at first glance. Another synchronization-related difficulty is getting agents to move at the same pace. In StarLogo 2's VM, an agent with fewer commands in its program will execute the entirety of its program more quickly than one with more commands, causing that agent to appear to move more quickly even if both agents' only movement command in each iteration of the program is fd 1.

6.5.1.2 Speed Control

Another difficulty stemming from the StarLogo 2 VM is that agents cannot predictably be made to move faster or slower. As described above, adding commands to an agent's program can slow it down, but if the programmer tries to make the agent go, for example, fd 2 to make up for lost time, the command is broken down and run as two separate steps of fd 1, making fd 1 the absolute fastest speed possible. Additionally, because the extra commands slow the overall movement rate, there is no guarantee that an agent executing fd .5 as a part of its program will be moving half as fast as an agent executing fd 1, unless both agents also have an identical number of other commands. This makes any type of model where velocity needs to be explicitly controllable difficult to program in StarLogo 2.

6.5.1.3 Atomicity

A third, and perhaps the most important, drawback to the StarLogo 2 VM is that it does not allow agents to execute sequences of commands atomically, or as one command without interruption. By enforcing its pseudo-parallel, instantly visually updated execution model, the VM causes all agent com-

mands to be reflected on the screen. For example, the command sequence left 90, forward 1, right 90 (lt 90, fd 1, rt 90 in StarLogo 2) causes the agent to turn twice, even though the final heading is the same as the initial heading. If a programmer wanted to program a sidestep, in which the agent simply slid to the left while still facing forward, he or she would be out of luck. In fact, StarLogo 2's VM requires a new built-in primitive command for every action that needs to be animated as one fluid action rather than steps.

6.5.2 StarLogo TNG: The Next Step Forward

StarLogo TNG's VM was designed with the pros and cons of StarLogo 2 in mind, and it specifically addresses the issues of synchronization, speed control, and atomicity mentioned above. The StarLogo TNG VM is similar to the StarLogo 2 VM in that it allows agents to execute commands, but its execution model is substantially different. Rather than letting each agent execute one or a handful of commands before switching to the next agent, the StarLogo TNG VM actually lets each agent run all of its commands before switching. Additionally, visible updates to the agents and the world are only displayed at the end of the execution cycle after all of the agents have finished. In this way, although agents run fully serially, the visible output is indistinguishable from that of truly parallel execution, as every agent is updated graphically at the same time.

6.5.2.1 Synchronization

Synchronization issues, although still existent in StarLogo TNG, are now explicit and more easily understandable. Whereas in StarLogo 2 one could never be sure which agent would execute a command first, in StarLogo TNG the agent with the lower ID number is guaranteed to go first, leading to behavior that can be predicted and therefore explicitly programmed. In the example with the red and blue agents, if the programmer knew the red agent would always execute first, he or she could tell the red agent to save its old color to variable before setting its new color. Then when the blue agent executes, it can look at the value in the variable rather than at the other (previously red) agent's current color. Note that such a solution only works because of the explicit nature of synchronization in StarLogo TNG. If the programmer were unsure of which agent was going to execute first, he or she would not know which agent should look at the actual color and which should look at the color stored in the variable.

6.5.2.2 Speed Control

StarLogo TNG's VM also addresses the issue of controlling movement speed. Rather than breaking up movement commands into smaller steps (which would be futile since all commands are executed before switching anyway), it executes the commands fully and relies on Spaceland to draw the changes properly. Spaceland takes the difference in location and heading between consecutive execution cycles of each agent and draws several intermediate frames of animation connecting the two, making for smoothly animated movement at the user-programmed speed rather than jerky jumps as agents change locations instantly. To the users, an agent moving forward .5 simply appears to be moving half as fast as an agent moving forward 1.

6.5.2.3 Atomicity

Probably the most powerful benefit of StarLogo TNG's revised execution model is that it allows programmers to define new atomic actions. Sidestepping, for example, is the automatic result of turning left 90, moving forward 1, and turning right 90. Because Spaceland only interpolated between the previous execution cycle and current one and because the heading ended up where it started, no heading change is animated, making for a smooth sidestep. This same pattern holds for arbitrarily complex sequences of commands, allowing the agents to perform complex measurements such as walking to every neighboring patch to see which is higher, while only appearing to move the distance between the prior execution cycle and the current one.

In fact, although extremely powerful, this atomicity can be difficult to grasp at first because it causes some commands to be executed invisibly. StarLogo TNG addresses this problem with the introduction of the yield command, which causes the agent executing it to save its current state and end its turn in the execution cycle, thereby allowing Spaceland to draw its state at that intermediate moment in its execution. Then, in the next execution cycle, the agent restores its state and continues from where it left off rather than starting from the beginning of its program again. By making the execution model more explicit, StarLogo TNG avoids confusion while giving more power and flexibility to the programmer.

6.5.3 An Addition in StarLogo TNG: Collisions

StarLogo TNG's VM also implements a new form of interaction that is easy for programmers to understand and use: collisions. The collisions paradigm allows for event-driven programming, in that users can specify commands for each type of agent to run in the event of a collision, and the VM constantly

checks for collisions and tells the agents to execute the specified commands as appropriate. For example, a common collision interaction would be for one agent, the "food," to die, while the other, the "consumer," increments its energy. This pattern is also particularly useful in games where a character collects items for points or additional capabilities. An additional benefit of introducing collisions as an event-driven paradigm is that it allows the VM to calculate collisions once, for all agents, rather than having agents ask for surrounding agents, as they do in StarLogo 2.

All of the features in StarLogo TNG, including the virtual machine, were made with the goals of providing the user with more power and flexibility while simultaneously lowering the barrier to entry for novices. By starting with a proven capable VM from StarLogo 2 and improving it with the additions and adjustments described above, all based on feedback from users and careful thought, StarLogo TNG moves closer to those goals. By doing so, it opens up new possibilities and new domains for StarLogo TNG, including games and physics simulations, as discussed in Section 6.7.

6.6 StarLogo TNG Example Model

To illustrate the features of StarLogo TNG, this section describes how to build a simple model of an epidemic – a model in which red agents represent sick individuals and green agents represents healthy individuals. When a red individual touches a green one, the green one will get sick. Then we add a recovery variable that can be adjusted by the user and graph the changing number of sick and healthy agents.

6.6.1 Setup

First, we will create some agents and put them on the screen. In the Setup section of the canvas, we put together the following blocks:

The Setup block runs once to set the initial conditions of the simulation. In this case, we want to delete all agents from the previous run, create 500 turtle agents and set their color to green (to represent 500 healthy individuals), and scatter them randomly in Spaceland. Note that turtles are the default agent, but both the name and shape can be changed in the Breed Editor:

To test the Setup block, look at the Runtime space connected to Spaceland. You should see a button named Setup. Click it and you will see the agents created and dispersed as seen in Fig. 6.9.

Fig. 6.7 Setup block for epidemic model.

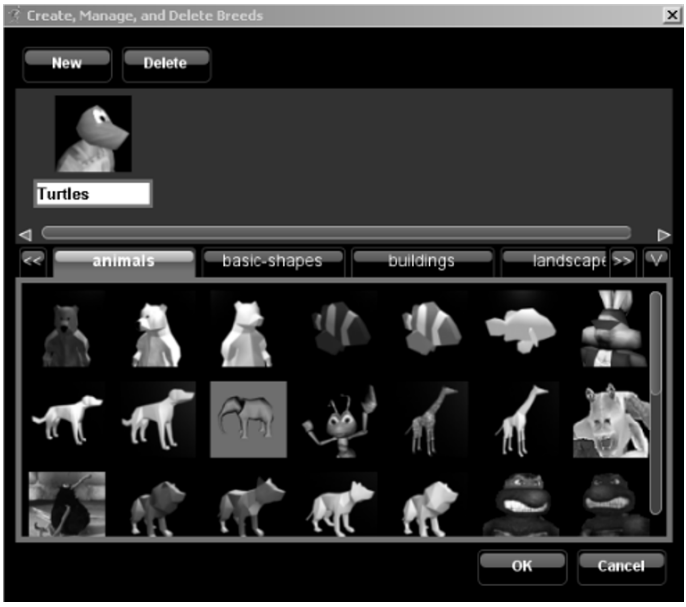

Fig. 6.8 Breed Editor.

6.6.2 Seeding the Infection

To infect some of the turtles at the start, we put together some blocks in a Run Once block and put them in the Setup area. We edit the Run Once name to say Infect to identify the purpose of this Run block (Fig. 6.10).

Notice that the Infect code is connected to a hook that is labeled "Turtles." This means that every turtle agent runs this If block once. The If block test condition randomly selects a number between 1 and 100 and compares it to 10. If the randomly selected number is less than 10, the turtle agent executes the then section of the block, which tells it to set its color to red. Since random 100 means that every number between 1 and 100 is equally likely to

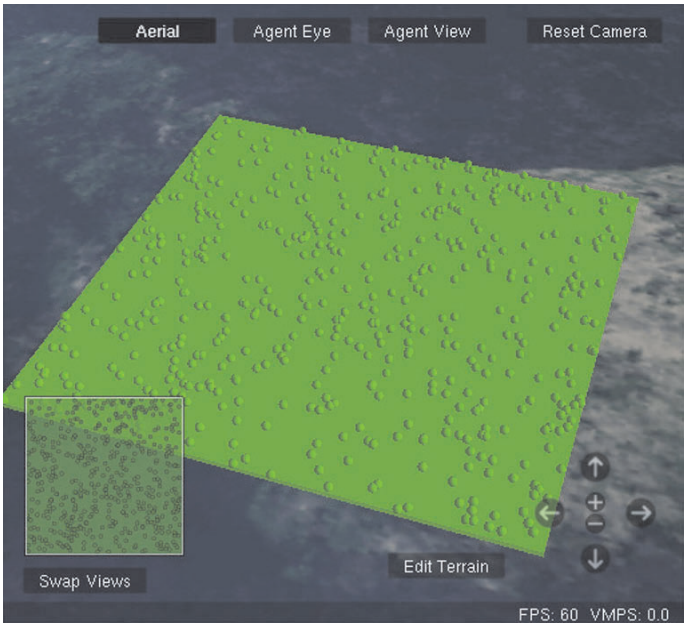

Fig. 6.9 Five hundred healthy agents created and scattered in Spaceland.

Fig. 6.10 Infect Run Once code to seed the infection at the start of the simulation.

be selected, there's a 10% chance that the number selected will be less than 10 and thus, approximately 10% of the 500 turtles will be infected.

We can test the Infect code by clicking on it in the Runtime window. The results are seen in Fig. 6.11.

6.6.3 Motion

To get the agents moving, we use a Forever block, which works as a loop. As seen in Fig. 6.12, we put the Forever block in the Runtime section of the programming canvas and attach some movement blocks to it to make the turtle agents move forward, with some random left and right "wiggle."

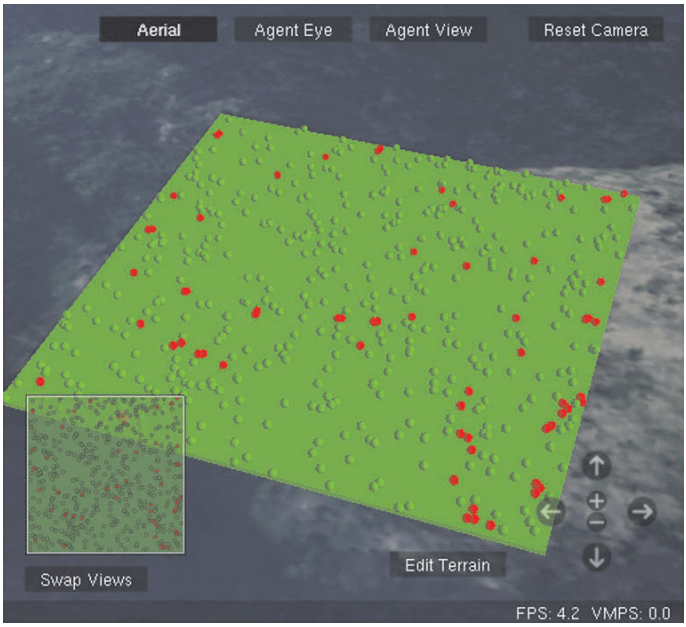

Fig. 6.11 Result of the Infect code as shown in Spaceland.

Fig. 6.12 Forever (loop) code to make the agents move.

The forever block also appears in the Runtime window and clicking on it will produce a green highlight to show that the forever block's code is running continuously in a loop until it is clicked again to stop (Fig. 6.13).

Fig. 6.13 Forever block turned on in Runtime window.

Fig. 6.14 Collision code between two turtle agents to spread the infection.

6.6.4 Infection

We program a Collision block to get the agents to infect each other when they "touch" when moving around in Spaceland. The Collisions blocks are found in the My Blocks palette, where breed-specific blocks are located. We drag the Collisions block between Turtles and Turtles to the Collisions section of the canvas (Fig. 6.14) and put in some blocks that tests if the color of the turtle being collided with is red. If it is, then the turtle agent turns its own color to red, showing that it has been infected by the other red (and sick) turtle. We use the same test for both turtles in the collision to make sure that we cover both cases where either of the turtles in the collision can be red.

Now when we run the model, we can see the infection spreading as more and more agents turn red from contact with other red agents (as seen in Fig. 6.15).

6.6.5 Recovery

To make our model more complex, we can add a recovery element, which is represented by a percent change of recovery (change color to green). Thus, during each iteration of the Forever loop, all turtle agents will have some chance of recovering, according to a percent chance variable that the user can set with a slider. A high number means a higher chance of recovering and a low number means a lower chance of recovering.

First, we create a global variable called Recovery by dragging out a "shared number" block from the Variables drawer to the Everyone page of the canvas and renaming it Recovery as seen in Fig. 6.16. Because we want the user to set the variable, we also drag out a slider block and snap it in front of the shared number block.

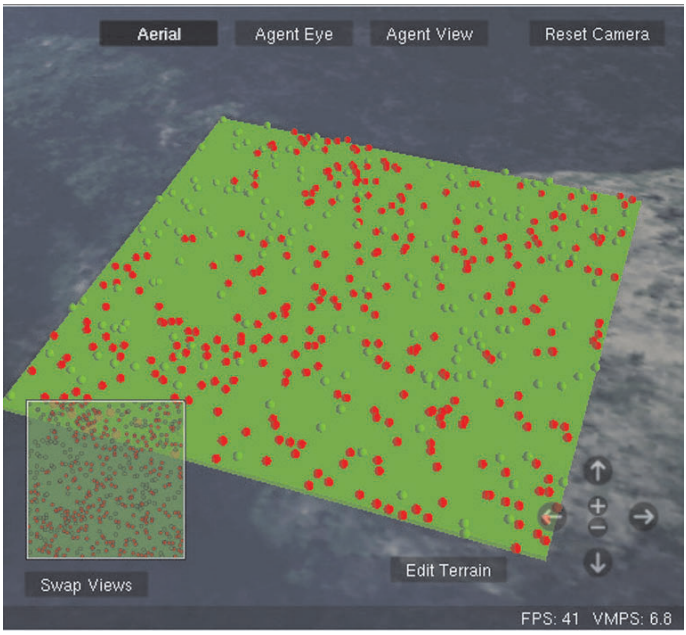

Fig. 6.15 Most of the agents infected after some time as a result of the Collision code.

Fig. 6.16 Slider block attached to variable declaration.

In the Runtime window, the slider appears as in Fig. 6.17.

Fig. 6.17 The slider appears in the Runtime window so that the user can adjust the value of the Recovery variable.

The minimum and maximum values of the slider can be adjusted by clicking on the numbers below the slider and typing in new values.

To give the turtles a chance to recover, we write a procedure called Recover. To define a procedure, we grab a Procedure block and put it in the Turtles

section of the canvas. Rename the procedure Recover and put in the code as shown in Fig. 6.18, noting that it uses the recovery variable.

Fig. 6.18 Recover procedure for epidemic model.

Now we have the turtle agents call the Recover procedure in the Forever block so that the turtle agents run the Recover code after they move during each iteration (Fig. 6.19).

Fig. 6.19 Turtle agents call the Recover procedure as part of Forever code.

6.6.6 Graphs

It is useful to track and compare the numbers of healthy and sick turtles over time so the code in Fig. 6.20 shows how to set up a double-line graph.

Fig. 6.20 Code to set up a line braph that tracks the number of green and red agents.

The line graph appears in the Runtime window and can be double-clicked to enlarge in a separate window with other options (Fig. 6.21).

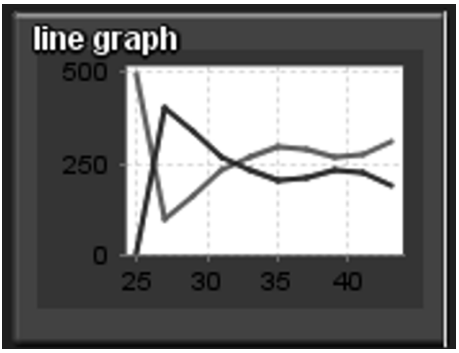

Fig. 6.21 How the line graph appears in the Runtime window.

These are just some of the basic features of programming a simulation model in StarLogo TNG: Setup, Forever, Breed Editor, Movement, Collisions, Slider, Variables, Procedures, and Line Graph.

6.7 Field Testing

StarLogo TNG has been downloaded by more than 10,000 users to date and it is being used in a variety of schools, clubs, and learning environments. We have used StarLogo TNG for introducing programming to students, teaching complex systems, and providing in-depth insights into a variety of scientific concepts. One of the most recent forays that StarLogo TNG has enabled is its use in physics classes. The following subsection is a description of this use.

6.7.1 Learning Physics Through Programming

diSessa [6] showed that fundamental physics concepts could be made accessible to students as early as the sixth grade by using simple programming activities. We believed that at the high school level, programming could help students build a deep understanding of traditionally difficult physics concepts, so we introduced StarLogo TNG programming basics through a series of physics-based activities to three high school physics classes at a private school in the Boston metropolitan area. A video-game-building activity helped establish a relationship between students and TNG that enabled them to use TNG effectively as a learning tool. Students programmed realistic motion in their games, including velocity, acceleration, friction, gravity, and other aspects of Newtonian physics. The algorithmic thinking of the 2D motion activities provided a new and useful entry point for learning physics

concepts that along with labs and problem-solving formed a stable, three-legged platform for learning that aided classroom dynamics and appealed to a diversity of learning styles. While students in this study were not randomly assigned, three other comparable physics classes at the same school were used for comparison. Data were collected in the form of written assessments, lab reports, surveys, and interviews.

We found that TNG programming and simulations were useful in the physics classroom, because TNG added a new dimension to the ways students think and learn physics. For example, a physics teacher usually defines velocity as displacement/time, $v = d/t$. Then she describes the motion of an object by stating that it has a velocity of, say, 1 meter per second. With TNG, students program motion by building the code (Fig. 6.22): set x [x +1]. The 1 is called stepsize – the distance an agent moves in each time-step. Stepsize is equivalent to the concept of velocity. The code is abstract, but less so than the equation. When executing the code, students see the agent stepping 1 unit each time-step. TNG has graphing capabilities, so they can see how their agent, executing the code they built, generates a linear position versus time graph.

TNG proved to be more than a programming environment; it was a tool with which one could think and ask questions. If some students in the class are setting x [x+1] while others are setting y [y+1], there is a good chance someone in the class will ask, "What if I set x [x+1] and set y [y +1] together?" If the physics teacher creates an environment where some students are programming motion in the x direction and others in the y direction, there is a very good chance that someone in the room will "discover" that if x and y direction codes are executed simultaneously, the agent will move diagonally. TNG made the discovery of the vector addition of motions more likely.

The concept of simultaneous but independent change that is central to understanding 2D motion is difficult and frustrating to teach and learn. We hypothesized that by adding the programming of 2D motion in a virtual world, in addition to the usual mathematical analysis of that motion in the physical world, we could enhance the understanding of this difficult concept. Below are some programming activities that both introduced and supported the new concepts students would need to develop a deep understanding of 2D motion.

Students began the unit by building separate simple programming procedures that changed agent attributes like color, size, or location. Keys could be pressed individually or simultaneously, producing many humorous combinations of change.

Students recognized that the value "1" represented the distance covered in a given iteration. This number, designated "stepsize," was linked to the concept of velocity – the distance covered in a given second. With stepsize as a variable (Fig. 6.23), students could change "velocity" by using a slider that sets that variable and moves their agent in any desired direction.

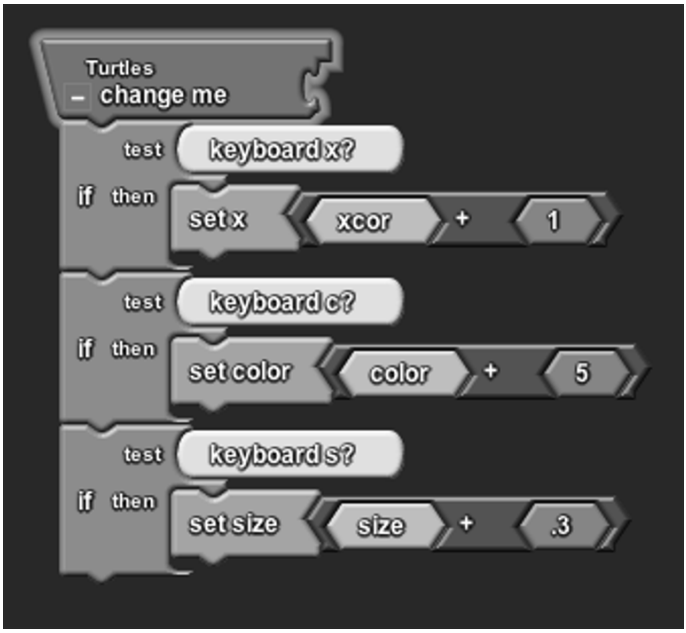

Fig. 6.22 Simultaneous and Independent change – code of keyboard controls that change agent attributes.

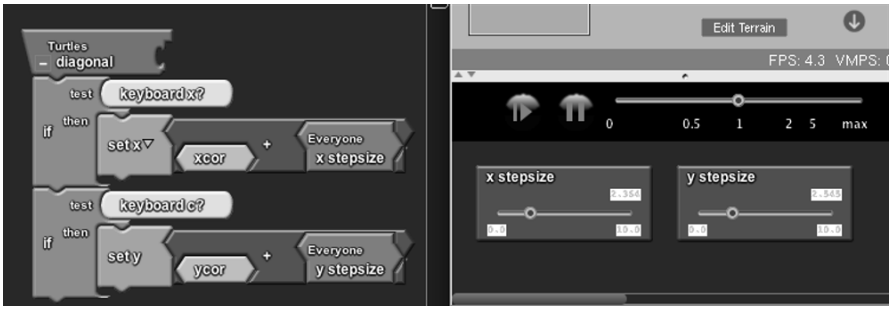

Fig. 6.23 Stepsize and Variables – code of keyboard controls for movement where the steps taken can be adjusted as slider variables.

The teaching of the advanced programming concept of variables and the addition of vertical and horizontal motions occurred naturally, linked by the students' desire to create a variety of motions for their agents. Because students were comfortable with TNG as a learning tool, the new ideas were introduced and the learning occurred in an environment where students had competence, comfort, and confidence.

The culmination of the kinematics section was a unit on projectile motion. To get realistic vertical motion, students added a procedure for the negatively directed acceleration of freefall in the z direction to independent motions on

the ground. To give students experience with this hard-to-picture situation, a game was used. Students built a jumping game from instructions and the help of two students in the class who had used jumping in their video game. To get an agent to jump over an obstacle, students had to separately build yet simultaneously execute a constant-velocity, move-forward procedure and a vertical-jump procedure that employed acceleration. The goal of the game was to get a raccoon to jump over a wall and hit a target (Fig. 6.24).

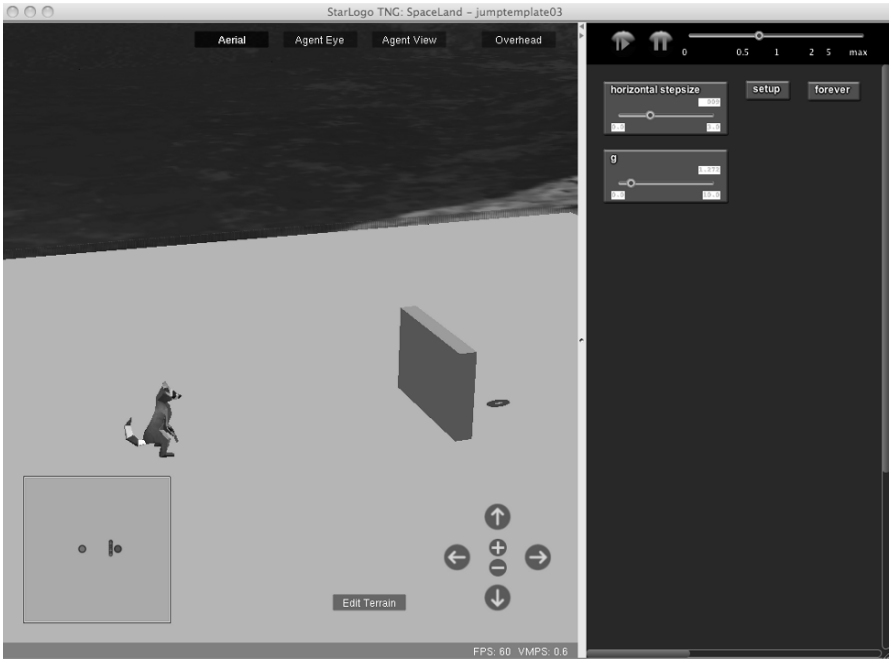

Fig. 6.24 Jumping game screenshot – students adjust variables to get the racoon to jump over the wall and land on the target on the other side.

Here, solid, experiential learning is achieved with the game-like simulation. "To get the raccoon to hit the donut, I had to…I changed the horizontal stepsize smaller, and increased the raccoon's jump start speed (*initial vertical velocity*). I did this because a large horizontal stepsize made the agent exceed past the donut."

The acquisition of new skills and confidence through programming is demonstrated in this anecdote. In a junior physics class, students built swimmer-in-river simulations as part of a unit on vectors. The model has a swimmer with velocity, s, swimming at a given heading across a river of given width with current velocity, r. After building and playing with the simulation, students were assigned 2D motion problems for homework. During a class discussion about the answer to a contested problem, one student went,

unprompted, to the computer and opened his swimmer model. He plugged the variables of the problem into the code of the model and ran the model. As the swimmer reached the opposite shore, he exclaimed, "I told you I was right!" When asked in an interview why he chose to use the computer rather than mathematical analysis to prove his point, he said. "This way you could ***see*** I was right." StarLogo simulations were a comforting, concrete, and useful refuge for some students in the sometimes troubling abstract world of physics equations. One student commented in an interview after the projectile motion test, "When I do TNG everything makes sense and I can do it. Once ... the questions are all on paper, it really messes me up and I have problems doing it." In one of the physics class that used StarLogo TNG, while 75% of the surveyed students agreed with the statement that the StarLogo unit was more difficult than other units, 100% of the students felt the unit was more rewarding.

6.7.2 From Models to Games

We developed and piloted StarLogo TNG curricula at two middle schools in a low-income town in Massachusetts. At one school, students learned programming by creating a first-person perspective treasure hunt game, in which the player controlled an agent using the keyboard to "collect" treasure and avoid hazards. Students had little to none prior programming experience. What is most striking from this field test is students' motivation to learn advanced programming concepts because they want to add a "cool" feature in their games, like shooting a projectile or creating a pattern in Spaceland's terrain. Also, when one student discovers how to use a particular block, such as "say" or "play sound," the knowledge quickly spread throughout the class and are incorporated into the students' own games. These two elements – a "need-to-know" desire to learn and informal peer learning – are features of classrooms where students are excited and motivated to persevere through challenging activities.

At another middle school pilot site, students made observations of and conducted experiments by manipulating variables of simulation models of forest fires and virus epidemics. The interesting twist is that they then turned these simulations into games by adding breeds and keyboard controls. The first activity took about 1 hour and the second activity took between 1 and 2 hours. So within 3 hours, students were able to both learn to use a simulation and program a basic game, with the guidance of a teacher and supporting worksheets.

When interviewed about their experiences, students shared the following:

- "You changed my mind. I thought that I was never going to be a programmer. I thought that the computer was just for work, and to look [up] stuff, and research on the Internet... it's more than just work and stuff, you can

do whatever you want on the computer. It was a process 'cause to tell you the truth. I thought it was going to be boring. I usually play video games, not make them. But it was fun."

- "It's like science, but better. It's not all textbook. You get to do what you're learning. We were doing things that we're going to remember because with Starlogo, I'm going to keep on keep on using it, using it, using it and I'm going to keep on getting better... I'm not going to forget it, like if I was reading a book or something."
- "It makes me think about how to control things. I have to think. I have to use calculations."

6.8 Conclusion

The testing and focus groups that we have used so far have suggested that the blocks design will promote the use of StarLogo as a programming tool in the classroom. Teachers can feel more secure using a tool that removes anxiety about getting the syntax right and makes learning programming easier. While it is still early to tell, it seems that it may also provide a more female-friendly programming environment than other tools.

The 3D world is "definitely cool," in the words of many of the students. While teachers have been more lukewarm in their reception of this aspect, they recognized that it will be more attractive to students. It is clear that both 2D and 3D representations will have a place in the learning environment, and the ability to customize the world and make it look like something recognizable (although not realistic – which is important in conveying that these are simply models) is a big plus for students as they approach modeling for the first time. It also allows students to invest themselves personally in their products, which is empowering and motivating.

StarLogo TNG will be in development for some time. Some of the next steps involve adding features like network connectivity and joystick control, in order to add to the attraction of those wishing to build games. However, we also will be building libraries of blocks for specific academic domains (e.g., ecology, mechanics, etc.) that will allow students to quickly get started building science games using some higher-level commands, which can, in turn, be "opened up" and modified as their skills progress.

Acknowledgements The research in this chapter was supported in part by a National Science Foundations ITEST Grant (Award 0322573).

References

1. Agalianos, A., Noss, R., Whitty, G.: Logo in mainstream schools: the struggle over the soul of an educational innovation. British Journal of Sociology of Education 22(4), 479–500 (2001)
2. American Association for the Advancement of Science: Project 2061: Benchmarks for Science Literacy. Oxford University Press, Oxford (1993)
3. Begel, A.: LogoBlocks: A graphical programming language for interacting with the world. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA (1996)
4. Casti, J.L.: Complexification: Explaining a paradoxical world through the science of surprise. Abacus, London (1994)
5. Cypher, A., Smith, D.C.: Kidsim: End user programming of simulations. In: I.R. Katz, R.L. Mack, L. Marks, M.B. Rosson, J. Nielsen (eds.) Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems', Vol. 1 of Papers: Programming by Example, pp. 27–34, ACM, Denver, CO (1994)
6. diSessa, A.: Changing Minds: Computers, Learning, and Literacy. MIT Press, Cambridge, MA (2000)
7. diSessa, A., Abelson, H.: Boxer: A reconstructible computational medium. Communications of the ACM 29(9), 859–868 (1986)
8. Harel, I.: Children as software designers: A constructionist approach for learning mathematics. The Journal of Mathematical Behavior 9(1), 3–93 (1990)
9. Harel, I., Papert, S. (eds): Constructionism, Ablex Publishing, Norwood, NJ (1991)
10. Holland, J.H.: Emergence: From chaos to order. Helix, Reading, MA (1998)
11. Kafai, Y.B.: Minds in Play: Computer Game Design as a Context for Children's Learning. Lawrence Erlbaum Associates, Hillsdale, NJ (1995)
12. Kafai, Y.B.: Software by kids for kids. Communications of the ACM 39(4), 38–39 (1996)
13. Logo Computer Systems Inc. Microworlds Logo, Highgate Springs, VT (2004)
14. Lucena, A.T. (n.d.): Children's understanding of place value: The design and analysis of a computer game.
15. Mathworks: Matlab, Natick, MA (1994)
16. Nature: 2020 Vision. Nature 440, 398–419, New York, NY (2006)
17. Papert, S.: Mindstorms: Children, Computers, and Powerful Ideas. Perseus Books, Cambridge, MA (1980)
18. Papert, S.: The Children's Machine: Rethinking School in the Age of the Computer. Basic Books, New York, NY (1993).
19. Resnick, M.: Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds (Complex Adaptive Systems). MIT Press, Cambridge, MA (1994)
20. Resnick, M.: Rethinking learning in the digital age. In: G. Kirkman (ed) The Global Information Technology Report: Readiness for the Networked World. Oxford University Press, Oxford (2002)
21. Resnick, M., Bruckman, A., Martin. D.: Pianos not stereos: Creating computational construction kits. Interactions 3(5), 40–50, New York, NY (1996)
22. Resnick, M., Martin, F., Sargent, R., Silverman, B.: Programmable bricks: Toys to think with. IBM Systems Journal 35(3–4), 443–452, Yorktown Heights, NY (1996)
23. Resnick, M., Rusk, N., Cooke, S.: The Computer Clubhouse. In: D. Schon, B. Sanyal, W. Mitchell (eds): High Technology and Low-Income Communities. pp. 266–286. MIT Press, Cambridge, MA (1998)
24. Roberts, N., Anderson, D., Deal, E., Garet, M., Shaffer, W.: Introduction to Computer Simulation: A System Dynamics Modeling Approach. Addison-Wesley, Reading, MA (1983)
25. Roschelle, J.: CSCL: Theory and Practice of an Emerging Paradigm. Lawrence Erlbaum Associates, Hillsdale, NJ (1996)

26. Roschelle, J., Kaput, J.: Educational software architecture and systemic impact: The promise of component software. Journal of Education Computing Research 14(3), 217–228, Baywood, NY (1996)
27. Roughgarden, J., Bergman, A., Shafir, S., Taylor, C.: Adaptive computation in ecology and evolution: A guide for future research. In: R. K. Belew and M. Mitchell (eds.). Adaptive Individuals in Evolving Populations: Models and Algorithms, Santa Fe Institute Studies in the Science of Complexity Vol. 16, pp. 25–30. Addison-Wesley, Reading, MA (1996)
28. Sander, T.I., McCabe, J.A.: The Use of Complexity Science. A Report to the U.S. Department of Education. USGPO, Washington, DC (2003)
29. Soloway, E., Pryor, A., Krajik, J., Jackson, S., Stratford, S.J., Wisnudel, M., Klein, J.T.: Scienceware model-it: Technology to support authentic science inquiry. T.H.E. Journal, 25(3), 54–56, Irvine, CA (1997)
30. State of Maine, D.o.E.: Maine Learning Technology Initiative, Augsuta, ME (2004)
31. Waldrop, M.M.: Complexity: The Emerging Science at the Edge of Order and Chaos. Simon and Schuster, New York, NY (1992)
32. White, B.: Thinkertools: Causal models, conceptual change, and science education. Cognition and Instruction 10, 1100, Philadpelphia, PA (1993)
33. Wilensky, U.: Statistical mechanics for secondary school: The GasLab modeling toolkit. International Journal of Computers for Mathematical Learning 8(1), 1–41 (special issue on agent-based modeling), New York, NY (2003)