SimLogo: Improving Simulations in StarLogo TNG

by

Nicole A. Bieber

S.B., C.S. M.I.T., 2010
S.B., Math M.I.T., 2010

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science
at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY
February 2011

© Massachusetts Institute of Technology 2011. All rights reserved.
SimLogo: Improving Simulations in StarLogo TNG

by

Nicole A. Bieber

Submitted to the Department of Electrical Engineering and Computer Science on February 1, 2011, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

StarLogo TNG (The Next Generation) is a graphical programming environment originally developed by the MIT Scheller Teacher Education Program as a means of improving computer literacy and programming skills in middle and high school students. The original goal of StarLogo TNG was to make computer programming more accessible to students by designing a programming framework, StarLogoBlocks, which could graphically represent the control flow and data structures of a program, and to introduce a “coolness” factor by permitting students to view their program’s execution in a 3D game-like environment called SpaceLand. However, after the program was released, the team found that researchers and professionals were using StarLogo TNG for more complex agent-based simulations than it had been originally designed to support. This thesis outlines the design and implementation of a new suite of features, code-named SimLogo, which will improve the simulation capabilities of StarLogo TNG through the addition of true internal parallelism to the virtual machine.

Thesis Supervisor: Eric Klopfer
Title: Director, MIT Teacher Education Program
Acknowledgments

This thesis was made possible through the help and support of many people. I would first like to thank my supervisor, Prof. Eric Klopfer, who provided me with the opportunity to be a part of this amazing project, and who has been incredibly supportive throughout my entire M.Eng. process. I learned a lot from him both in lab and in his class on using games in education, and it was truly an honor to work with him.

I would next like to thank Daniel Wendel, without whom this thesis truly could not exist. As the leader of the StarLogo TNG team, Daniel taught me everything I know about StarLogo TNG, and it was through his patient teaching and guidance that nearly every aspect of this project took form. Over the course of the past year, Daniel has proved himself to be both an excellent mentor and friend, and he, along with the rest of the StarLogo TNG team (namely, Wendy Huang, Eli Kosminsky, Hailey Kopp, Chris Cheng, Angel Irizarry, and David Lam) has made the StarLogo office an incredibly fun place to work. I would also like to thank the rest of the members of the Scheller Teacher Education Program; I greatly enjoyed seeing all of the awesome projects they were working on and playing board games with everyone on “Fun Fridays.” Special thanks are also in order for Eli Kosminsky, who worked with me on debugging StarLogo TNG’s 3D Model Importer this summer.

My time at MIT would not have been the same without my friends, who kept me sane (but not too sane) throughout both my M.Eng. and undergraduate years; to the C-Entry and “Summerfun” folks, my roommate Priya, as well as my friends from home, thank you for always being there for me. Special thanks to Raffi Rush, who helped me to brainstorm potential applications of StarLogo TNG to medical education, and to Katie Ray and Ankit Gordhandas for their invaluable help in proofreading.

Last but certainly not least, I would like to thank my family for all of their infinite love and support. Thank you Mom and Dad for always encouraging and helping me to achieve my dreams, to David (the world’s best little brother ever, in my humble and
only slightly biased opinion) for constantly motivating me to be a better programmer, and to Joel, my incredible fiancé who was by my side through every step of my MIT journey, even when he was two hundred miles away.
Contents

1 Introduction ......................................................... 11
   1.1 Structure of StarLogo TNG ................................. 12
      1.1.1 StarLogoBlocks ........................................ 12
      1.1.2 SpaceLand ............................................. 13
      1.1.3 Virtual Machine ....................................... 13
   1.2 Previous Work .............................................. 14
   1.3 What is StarLogo TNG Used For? ......................... 16

2 The Old Virtual Machine .......................................... 19
   2.1 Properties of this Execution Model ....................... 20
      2.1.1 Atomicity ............................................. 20
      2.1.2 Pseudo-Parallelism .................................... 21
   2.2 Use Cases .................................................. 21
      2.2.1 Racing Lions .......................................... 21
      2.2.2 Colliding Spheres ..................................... 22
   2.3 Implementation ............................................ 23

3 The New Virtual Machine .......................................... 27
   3.1 What does it mean for agents to run "at the same time"? 27
   3.2 The Problem With Threads .................................. 28
   3.3 The New Execution Model .................................. 29
   3.4 Implementation ............................................ 29
      3.4.1 Collisions ............................................. 30
List of Figures

1-1 An example of StarLogoBlocks code. The *Walk N Steps* procedure takes a single numerical argument called *N*. On the left, we have the procedure definition, which specifies that when *Walk N Steps* is called, the agent will move forward *N* steps. On the right, *Walk N Steps* is called with a value of 3. .................................................. 13

1-2 Two views of a SpaceLand arrangement containing several Turtle agents. 14

1-3 The original Logo turtle from 1969 was a small robot with a pen that could be programmed to draw interesting patterns. [1] .......................... 15

1-4 A code sample written with LogoBlocks. [2] ................................. 16

2-1 A novice user running this code might be surprised when they do not see an agent moving in a square pattern. ................................................. 20

2-2 A simple program for racing three lion agents (setup code which initializes lion agents not shown). ................................................................. 22

2-3 The code in these StarLogoBlocks does not switch the colors of the agents upon collision as one might expect. ................................. 23

2-4 Data storage in the StarLogo TNG 1.2 virtual machine. All arrows represent pointers. ................................................................. 25

2-5 Data storage in the new (SimLogo / StarLogo TNG 1.5) virtual machine. 26

4-1 The new StarLogoBlocks. .......................... 36

6-1 A StarLogoBlocks function designed to remove an element from a list. 42

6-2 The StarLogo TNG line graph environment. ................................. 43
Chapter 1

Introduction

StarLogo TNG (The Next Generation) is a graphical programming environment for agent-based simulations originally developed by the MIT Scheller Teacher Education Program as a means of improving computer literacy and programming skills in middle and high school students. The original goal of StarLogo TNG was to make computer programming more accessible to students by designing a programming framework, StarLogoBlocks, which could graphically represent the control flow and data structures of a program, and to introduce a “coolness” factor by permitting students to view their program’s execution in a 3D game-like environment called SpaceLand [3,4].

The original mission of StarLogo TNG was successful, and the program was well received by students and teachers in schools around the world. However, an unintended second potential usage of the software soon surfaced. While we intended for teachers and students to perform simple simulations using the software and included functionality to this end, we were surprised when academic researchers and university students from various fields began trying to implement serious scientific simulations in StarLogo TNG and include the results in their publications. In many ways, StarLogo TNG was well suited to this purpose as unlike in other simulation software, the graphical nature of StarLogoBlocks makes coding accessible to researchers whose background may not have included strong programming skills, and the presence of SpaceLand eliminates the need for writing separate code to visualize the data. However, since StarLogo TNG was not designed for this purpose, there are certain features
essential to a good simulation program that are lacking in StarLogo TNG. Namely, StarLogo TNG could stand to benefit greatly from the inclusion of true internal parallelism, that is, an execution model in which all agents in the simulation behave as though they are acting "at the same time".

Although StarLogo TNG approximates this parallelism, it is not powerful enough for some of the more complex simulations made possible with this software, and sometimes leads to unexpected results. In this thesis, I will explore the addition of true internal parallelism to StarLogo TNG through the restructuring of the StarLogo TNG virtual machine.

1.1 Structure of StarLogo TNG

StarLogo TNG is composed of three pieces - StarLogoBlocks, the graphical programming language which allows users to program by connecting graphical puzzle pieces that determine the program flow, Spaceland, the 3D virtual world in which these programs are executed, and the StarLogo TNG Virtual Machine, which runs behind the scenes to control all agent behavior. It is this last piece which will be the focus of this thesis.

1.1.1 StarLogoBlocks

The StarLogoBlocks language is a graphical alternative to text-based programming. Each StarLogoBlock is a graphical object resembling a puzzle piece which represents a single piece of code. However, unlike in traditional programming languages, programmers using StarLogoBlocks do not have to worry about syntax issues - in StarLogo TNG you can never forget a semicolon. Instead, connections between two blocks are constrained by their shapes; many blocks have shaped sockets, in which arguments may be connected only if they are of the proper shape and type. For example, a procedure that takes in a single number argument will have a single V-shaped indentation, indicating that a V-shaped number block is required as a parameter (see Figure 1-1). This puzzle-like structure serves both to enforce correct syntax and pre-
Figure 1-1: An example of StarLogoBlocks code. The Walk \textit{N} Steps procedure takes a single numerical argument called \textit{N}. On the left, we have the procedure definition, which specifies that when \textit{Walk \textit{N} Steps} is called, the agent will move forward \textit{N} steps. On the right, \textit{Walk \textit{N} Steps} is called with a value of 3.

vent errors, making it easy for a novice programmer to identify exactly which pieces of code belong together.

1.1.2 SpaceLand

The coding paradigm in StarLogo TNG revolves around the control of agents which are visible in a 3D world called SpaceLand. Users have a choice of several different types of agents including various shapes, animals, and buildings, and can write programs using StarLogoBlocks to control the behavior of each agent. The code is then executed in SpaceLand so that the users can view the outcome of their programs.

SpaceLand consists of a 101 x 101 2.5D grid, which means that for every \((x,y)\) coordinate ranging from \((-50, -50)\) to \((50, 50)\) there exists exactly one height (these heights can be modified in the SpaceLand editor). However, agents can exist anywhere above or below this grid.

SpaceLand allows users to easily visualize the output of their code and also offers quick real-time feedback, as there is no need to compile and run programs in StarLogo TNG. The 3D nature of SpaceLand also offers a whimsical “game-like” feel which makes programming seem less intimidating for novice users.

1.1.3 Virtual Machine

The StarLogo TNG Virtual Machine is written in C and works behind the scenes to control all agent behavior. On each cycle of the machine, each agent’s code is executed sequentially. However, the externally visible state (that is, the contents of
1.2 Previous Work

The Logo programming language first came into being in 1967 as an educational tool developed by Seymour Papert and Wally Feurzeig. It was a dialect of LISP developed with learning in mind, and after about a decade of incubating at MIT and other research sites, Logo began to make its way into schools, where it was used as a tool to introduce thousands to students to fundamental programming concepts [5].

One of the most unique and interesting features of Logo was the Logo "Turtle." The original Turtle was a small programmable robot that could be controlled by entering commands in the Logo language, but this soon evolved into a graphical on-screen representation. Given the technical limitations of the day, this "Turtle" actually looked much more like a triangle, but the name stuck and turtles would
continue to be coupled with the Logo language as it evolved over the next 30 years [5].

And evolve it did. As various universities and research organizations put their own spin on the Logo language, several variation began to emerge including but not limited to MicroWorlds, NetLogo, UCBLogo, KTurtle, LogoBlocks, and of course, MIT’s own StarLogo.

StarLogo, which was developed by Mitchel Resnick in the late 1980s as part of his masters thesis, was a specialized version of Logo designed for agent-based modeling. That is, instead of controlling a single turtle, users could run simulations involving hundreds of turtles, or agents, which can move around the world and interact with each other. Coupled with a much more sophisticated user interface than the original Logo, this proved to be incredibly powerful, allowing users to model decentralized systems such as bird flocks, traffic jams, and ant colonies [6].

Shortly afterwards, in 1996, Andy Begel, a student in Prof. Resnick’s group at the MIT Media Lab, would develop LogoBlocks - a graphical programming language for controlling the Programmable Brick, a small handheld computer that can be attached to LEGO motors and sensors [7]. In LogoBlocks, color-coded “puzzle pieces” could be interlinked in order to create a program.

StarLogo TNG, literally “the next generation” in the Logo family, made its debut
in 2008 as a fusion of StarLogo and LogoBlocks. As in StarLogo, StarLogo TNG is an agent-based modeling system that can be used to simulate the behavior of up to 4096 agents, but rather than using traditional Logo code, StarLogo TNG agents can only be programmed with LogoBlocks-style blocks. With the addition of a powerful StarLogoBlocks library and a much more sophisticated (and 3D) user interface, StarLogo TNG has stayed true to the original Logo philosophy of “low floor, high ceiling”; with the color-coded StarLogoBlocks making it easier than ever for novice programmers to get started while the powerful agent-based backend allows for very complex simulations.

1.3 What is StarLogo TNG Used For?

The main uses of StarLogo TNG can be divided into three categories:

- Introducing basic programming concepts using StarLogoBlocks. This is often accomplished through the lens of game programming, which makes programming enticing to younger students.

- Teaching STEM concepts to middle and high-school students using pre-written simulations of concepts such as: predator/prey population dynamics, forest fire spread and containment strategies, bird flocking patterns, termite nest building, and chemical reactions [4].
Highly complex simulations run by university students or professional researchers. For example, Banca D’Italia (Bank Italy) used StarLogo TNG to analyze payment systems, and researchers at the University of London have modeled how Dutch Elm Disease spreads on the Isle of Man [8,9].

While simulations in this last category currently make up the minority of programs written in StarLogo TNG, there is great potential here, and we would like to expand StarLogo TNG’s capabilities to encourage more programs from these “power users.”
Chapter 2

The Old Virtual Machine

The StarLogo TNG Virtual Machine works behind the scenes to control all agent behavior. When a user runs their StarLogo TNG code, this code is compiled into a byte array and then executed by the virtual machine, which is implemented in C, in the following manner [10, 11] :

1. Workspace blocks are compiled into a byte array. Any errors are reported to the user.

2. Compiled code is downloaded to the VM.

3. Iterate over all agents and check for collisions. Run collision code.

4. The VM “cycles”:

   (a) The VM iterates over each of the agents and runs their code, one at a time. Agent and patch attributes are updated.

   (b) The VM ticks to notify StarLogo that it has completed a cycle.

5. All agents are rendered in SpaceLand.

6. Steps 4 and 5 repeat until there are no running blocks.
Figure 2-1: A novice user running this code might be surprised when they do not see an agent moving in a square pattern.

2.1 Properties of this Execution Model

2.1.1 Atomicity

This execution model can be considered atomic, or "all-at-once", since it allows programmers to define new atomic actions by combining the commands on several StarLogoBlocks. For example, when a user combines the commands "left 90", "forward 1", and "right 90", they will see an agent take a single step to the side, the result of SpaceLand interpolating between the previous execution cycle and the current one (step 5), rather than rendering the result of every single agent action (see Figure 2-1). This effectively allows users to define new atomic actions by combining arbitrarily complex sequences of commands [12]. For example, a user could program an agent to find the highest adjacent patch by visiting each of the eight surrounding patches, comparing their heights, and then moving to the highest one.

This powerful feature may sometimes be confusing for novice users who expect to see their agent act out the code on each individual block. For instance, a novice user might be surprised that an agent running the code in Figure 2-1 will not move in a square pattern - in fact, it will not appear to move at all, though a square still appears on the terrain. This is because the agent has the same position and heading at the beginning and end of each VM cycle, and intermediate states are not shown in SpaceLand.
2.1.2 Pseudo-Parallelism

Although each agent's code is executed sequentially (in order of agent ID) during each cycle through the VM (step 4), SpaceLand is only updated a single time at the end of each cycle (step 5), creating the illusion that all agents in SpaceLand are in fact running in parallel. Each agent actually runs all of its commands before the VM switches to the next agent, but in most scenarios the end result is the same as if the agents in question had truly been running “at the same time.” In scenarios where the order that agents execute their code does matter, experienced programmers can make use of the fact that the agents always execute their code in order of their ID number.

However, despite this pseudo-parallelism, there are still scenarios which may arise in StarLogo TNG 1.2 where it is apparent that the agents are not actually running at the same time. We will see some examples of these issues in the following use cases.

2.2 Use Cases

2.2.1 Racing Lions

The code in Figure 2-2 illustrates a simple StarLogoBlocks program designed to race three lion agents against each other and have the winner say “I win!”. The lions move forward at a randomized number of steps between four and six at each cycle. Given this program with such a limited range for the lions' speeds, one might expect that two or more of the lions would tie a substantial fraction of the time. Since the code was not written to anticipate ties, the expected behavior might then be that all of the winning lions would say “I win!” and an arbitrary lion would be declared the winner in the status bar.

However, upon running code several times, a user might be surprised to see that this expected behavior does not take place. Instead, the lions never tie! There is always a lion standing at the head of the pack when the simulation halts. Furthermore, the lions do not win with equal probability. Lion 0 wins the most often, followed by
Lion 1, and then Lion 2. This is extremely surprising behavior to users who see that the three lions are executing exactly the same code and thus infer that they should each win the same percentage of the time.

The fallacy here lies in the fact that users assume true internal parallelism - that is, that all of the agents are executing their code at the same exact time. In fact, the virtual machine operates on agents sequentially, so that in this example, the code for Lion 0 is always executed before the code for Lion 1, which in turn is executed before the code for Lion 2 on each cycle. Since the simulation halts once a winner has been declared (once \texttt{winner ID} no longer equals -1), in a simulation where multiple lions would have crossed the finish line on the most recent cycle, the lion with the lowest ID will be declared the winner and the rest of the lions will stop in place and never take that last set of steps. This explains both why lions with lower IDs win more often, as well as why we never see more than one lion cross the finish line.

### 2.2.2 Colliding Spheres

Suppose we want to write a simple program where Sphere agents of different colors travel around SpaceLand until two Sphere agents collide, at which point the colliding agents switch colors. From a user perspective, the code in Figure 2-3 would perform
the expected behavior. However, actually running this code shows that after two Sphere agents collide, they both end up the same color. Once again, this is due to the lack of true internal parallelism in the StarLogo TNG virtual machine. Since the behavior of the agents is executed in series, when the second agent’s color is set to match the color of the first agent, the first agent has already been changed to match. For example, if a black and white sphere collide, the black sphere will be set to white, but then the original white sphere will be set to match the other sphere (which is now also white) so the two spheres will remain white.

2.3 Implementation

The bulk of the StarLogo TNG virtual machine is contained within a file called vm.c. vm.c contains a large switch statement with cases for each of the over 300 possible commands that can be compiled from StarLogoBlocks, as well as a function oncthrough, which controls everything that takes place on a single VM cycle.

Other important files include turtle.c, which contains several important functions for manipulating agent data, and turtle.h, where the data structures that store all agent information are defined. It is important to note that all agents, regardless of breed, are referred to as “turtles” throughout the codebase - a naming convention that harkens back to one of StarLogo TNG’s predecessors, the original Logo programming language.
The following elements form the basis of the data structures which store the state of the StarLogo TNG world:

- **turtles** - a pointer to a byte array where all agent data is stored, with the exception of agent variables which are stored in a separate byte array called *turtle_heap*. Each agent is represented by a Turtle struct, which contains information such as agent location and ID, as well as pointers *prev* and *next*, which refer to the next living turtle with a lower and higher ID, respectively. The Turtle struct also contains pointers into byte arrays *VisibleTurtles* and *ShownTurtles*, which store information about the visual state of the turtle, and a pointer towards that agent's allotted portion of *turtle_heap*.

- **turtle_heap** - A pointer to a byte array where agent variables are stored. Agent variables are the variables which users may create for their agents within StarLogo TNG. For example, a user might decide that each agent should keep track of the number of steps it has taken and create a variable called "steps". This would then increment *num_turtles_own*, which keeps track of the number of agent variables, and the amount of memory allotted to *turtle_heap* would be adjusted accordingly.

- **num_turtles_own** - The number of existing agent variables.

- **max_turtles** - The maximum number of agents allowed. This number is fixed at 4096.

- **num_turtles** - The number of agents currently alive.

For a further illustration of how these pieces fit together, see Figure 2-4.
Figure 2-4: Data storage in the StarLogo TNG 1.2 virtual machine. All arrows represent pointers.
Figure 2-5: Data storage in the new (SimLogo / StarLogo TNG 1.5) virtual machine.
Chapter 3

The New Virtual Machine

In SimLogo, we seek to avoid the problems described in the use cases above by furthering the illusion of true internal parallelism. In the old execution model, each agent’s behavior on each VM cycle was dependent on the behavior of agents with lower IDs on that cycle. This execution model sometimes led to unexpected behaviors, such as in the racing lions case, and did not seem to be in the true spirit of parallelism. In SimLogo, we implemented a new execution model based on the principle that agent behavior should not be dependent on ID number. In other words, all agents should behave as if they are the first to run on a VM cycle. This will create the illusion that all agents are executing their code at the same time.

3.1 What does it mean for agents to run “at the same time”?

When users have an expectation of parallelism, what does this actually mean? For agents that are scattered throughout SpaceLand in such a way that they will not interact with each other, making these agents appear to move “in parallel” is relatively straightforward. However, what happens when the agents do need to interact? In most cases, this too is not incredibly complicated: when two agents collide, the reaction of one agent to the collision should not be dependent on the other’s. In fact,
any change that an agent makes to the world on a given VM cycle should not affect the behavior of any other agent on that cycle. If it did, one could say that the agent whose behavior affected the other had acted “before” the other agent, breaking the illusion of parallelism.

Unfortunately, there are still some cases where it is difficult or even impossible to determine what the correct agent behavior should be. Suppose two agents running terrain-stamping code, one in red and one in blue, step onto the same patch. Which color should the patch be after the agents leave? Red? Blue? Purple? StarLogo TNG doesn’t use additive color mixing, so the latter is not a possibility under any execution model.

The answer to this question may be somewhat surprising: the correct color of the patch in question is actually undefined! Since the backend still actually operates by running agent code in sequence, the patch will be painted the color of the agent whose code ran last. However, what differentiates this from the previous model is that as long as the agents are still running their code for that cycle, both of them will perceive the patch color to be the color that they painted it.

### 3.2 The Problem With Threads

A common question I faced when discussing this project with those not familiar with the specific implementation details of StarLogo TNG was, “why are you trying so hard to imitate parallelism? Why don’t you just use threads?” The answer to this question lies in the fact that threads are actually no more parallel than this representation. Unless you are running a multi-processor machine, a single computer can only execute one instruction at a time; a multi-threaded application must still switch between instructions running on those threads.

It may seem confusing that we refer to “true internal parallelism” and the “illusion of parallelism” to mean the same thing, but in fact, all “truly” parallel applications (with the exception of those running on multiple processors) are simply creating a very convincing illusion.
3.3 The New Execution Model

In order to make each agent behave as if it is the first agent to run on a cycle, it is important that each agent have access to the state of the world as it was at the beginning of the cycle, before any other agents made changes to this state. In order to accomplish this, we must store two copies of the global state in this model, the current state which is updated by each agent as it executes its code, and a past state which remains fixed until the end of a cycle, at which point the current state is copied into the past state. This lends itself to the following execution model:

1. Workspace blocks are compiled into a byte array. Any errors are reported to the user.
2. Compiled code is downloaded to the VM.
3. The VM “cycles”:
   (a) Copy current state into past state.
   (b) The VM iterates over each of the agents:
      i. Check for collisions with all agents with a higher ID. Run collision code.
      ii. Run agent code, referring to past state for any queries about the state of the world and to current state for any queries about this agent’s own attributes. Update agent and patch attributes in current state.
   (c) The VM ticks to notify StarLogo that it has completed a cycle.
4. All agents are rendered in SpaceLand.
5. Steps 3 and 4 repeat until there are no running blocks.

3.4 Implementation

The following steps were performed in order to implement this new execution model:
1. Move collision handling so that collisions for each agent are handled immediately before that agent executes its code, rather than handling all collisions up front at the beginning of each VM cycle.

2. Change all heap and turtle pointers to be offsets to prepare for state duplication.

3. Create a second copy of the state. Copy the current state to the past state at the beginning of each VM cycle.

4. Go through every StarLogo TNG command and ensure that each command is using the correct copy of the state.

3.4.1 Collisions

In the old virtual machine found in StarLogo TNG 1.2, keeping track of collisions between agents is a highly memory-intensive operation. At the beginning of each VM cycle, a function update_collisions is called which iterates through every pair of agents (this takes $O(n^2)$ time) to determine whether they collide. Each agent has an array called collisions, which update_collisions fills with the IDs of all other agents it has collided with.

StarLogo TNG allows for a maximum of 4096 agents, so it is necessary for each agent’s collisions array to have the capacity to store up to 4096 IDs. Since these IDs are stored as integers, on a 32-bit machine, this would require a total of $32 \frac{\text{bits}}{\text{int}} \times 4096 \frac{\text{ints}}{\text{agent}} \times 4096 \text{ agents} = 64 \text{ MB}$. On a 64-bit machine, we would require twice as much memory, as integer storage requires 64-bits instead of 32 [13].

By moving the collision computations into the VM cycle as described in the SimLogo execution model above, it is no longer necessary to maintain a separate collisions array for each agent with the capacity to store up to 4096 IDs. Instead, a single collisions array with the capacity to store 4096 IDs can be used and overwritten after each agent’s collisions are computed. This requires a total of $32 \frac{\text{bits}}{\text{int}} \times 4096 \text{ ints} = 16 \text{ KB}$, creating nearly a 100% memory savings for collision storage and roughly a 25% memory savings in StarLogo TNG overall [13].
This memory-saving change was made possible by the changes to the rest of the execution model which involve duplicating the internal state of the world. Had collisions been restructured in this way under the old execution model, some agents would be forced to run their collision code while viewing a world state that was dependent on the actions of previous agents. It is only possible to interleave collision code with agent code in this manner in the new virtual machine because the state duplication ensures that every agent runs its collision code while viewing the world state as it was at the beginning of the VM cycle, a fact that was ensured in the old virtual machine by running collision code for all agents at the beginning of each cycle.

3.4.2 Change Heap and Turtle Pointers to Offsets

In preparation for duplicating the internal state of the world found in the byte arrays pointed to by turtles and turtle.heap, it was first necessary to convert all pointers in these arrays into numerical offsets that could be used to index into not only these arrays but their newly created duplicates as well. These pointers include those which are shown in Figure 2-4 as well as every single reference to the internal state of the world, most of which can be found in the giant switch statement located in vm.c.

The new structure for indexing into the turtles array works as follows: instead of each TurtleState struct containing pointers prev and next, each TurtleState struct now contains two ints prev.offset and next.offset. These ints contain the location of the agent in question with respect to turtles, making turtles + current.turtle->next.offset a valid way to refer to the next living turtle in turtles following current.turtle. The prev.offset and next.offset located in the first and last TurtleStates in the array, respectively, are both set to -1.

TurtleState's heap pointer was also replaced with an offset called heap.offset, which operates similarly to prev.offset and next.offset except with respect to turtle.heap rather than turtles.

It was also necessary to restructure TurtleState by removing the pointers vt and st which referred to instances of the VisibleTurtles and ShownTurtles structs, respectively, and instead incorporating these structs as fields in TurtleState so that
this data would ultimately be copied along with the rest of turtles when we duplicated the state. In order to accomplish this, it was necessary to adjust the amount of memory allocated for these structs by editing the file TurtleManager.java. Due to the integrated nature of the C and Java code in StarLogo TNG, all memory for the virtual machine is allocated in TurtleManager.java and then passed to the C-implemented virtual machine through the Java Native Interface (JNI).

For an illustration of storage in the new virtual machine, see Figure 2-5.

### 3.4.3 Create a Second Internal State

The next step toward implementing the execution model described above was to create a second copy of StarLogo TNG’s internal state. Now that we have moved the data contained by VisibleTurtles and ShownTurtles into TurtleState, this state is contained entirely in the byte arrays pointed to by turtles, turtle_heap, and another pointer called patches, which refers to the state of the terrain.

We created two new byte arrays, pointed to by past_turtles and past_turtle_heap to represent the copies of turtles and turtle_heap, respectively. At the beginning of each VM cycle (in the oncthrough function in vm.c), the contents of turtles are copied in to past_turtles and turtle_heap are copied in to past_turtle_heap, so that at any point during the cycle, the state at the beginning of the cycle can still be accessed in the “past” arrays.

We opted not to duplicate patches, the array which contains the terrain state, as there are very few scenarios in which an agent’s behavior is dependent on terrain that changes from one cycle to the next, and it was decided that forgoing these changes in order to avoid allocating additional memory for a past_patches array and to eliminate the need for refactoring all VM code that references patches was a reasonable tradeoff.
3.4.4 Modify Commands to Use New State

The final step toward true internal parallelism was to actually make use of the newly created “past state” storage by systematically modifying every reference to StarLogo TNG’s internal state in order to ensure that agents read and write to the correct state copy. Most of these references were found either in the large switch statement in vm.c or in TurtleManager.java, which is responsible for allocating all of the memory used by the virtual machine.

As a general rule, any command which involved agents reading data about the state of the world read from the “past state,” while agents changing state information write to the “current state.” However, agents looking at their own internal variables (agent variables) always read and write to the “current” copy of the heap.

3.4.5 Dead Agents

An agent can be “killed,” that is, removed from SpaceLand, through use of either the clear or die blocks. This is handled by the die(Turtle *t) function in turtle.c. When an agent is killed, it is killed in the current copy of SpaceLand’s internal state; its prev_offset and next_offset and those of it’s adjacent agents are modified so that it is no longer connected to the linked list of live turtles. However, all other agents operating on this cycle will still operate as though this agent is alive, since any references they make to this agent will be to the previous copy of the state found in past_turtles. Thus, all other agents will not register the death until the next VM cycle.

3.5 Disadvantages of this Execution Model

3.5.1 Patches Not Duplicated

As discussed in Section 3.4.3, we opted not to duplicate the patches array which contains the state of the terrain. While scenarios in which this decision comes into play are very rare, it is important be aware that this implementation may lead to
unexpected results in simulations where agent behavior is dependent on the state of terrain (ie. color and patch height) that may change mid-simulation. For instance, suppose we have a simulation in which agent behavior is dependent on the color of the patch they stand on. In this model, agents with lower IDs have the ability to change the terrain color during a cycle before agents running later in the cycle have a chance to sample it - leading them to behave slightly differently than they would if they were truly following the assumption that they were the first agent to run on that cycle.

However, it is important to note that while this is unfortunate in that it is generally best to design an execution model with as few exceptions to the rules as possible, this behavior is no worse than that found in the old execution model of StarLogo TNG 1.2, as patches has been left untouched. In addition, while this behavior may technically be breaking the “first agent to run on a cycle” rule, it may actually be closer to what users expect of their agents’ behavior:

For example, suppose we have a simulation that requires agents to say “I’m on a red patch” while standing on a red patch. Now, suppose that the first agent to run on a cycle colors a given patch red, and then the next agent moves onto that patch. The second agent will say “I’m on a red patch,” which, while inconsistent with the rules defined by the execution model, is consistent with what users will see in SpaceLand at the end of the cycle.
Chapter 4

Other Modifications

In addition to modifying the StarLogo TNG virtual machine, I made the following additions to the software. Most of these additions were user requested.

4.1 New StarLogo Blocks

4.1.1 Random Color

The random color block exists in the Colors drawer and can be used to create a random color. It works by reporting a random number between 1-140, which refers to a hue in StarLogo TNG’s color table.

4.1.2 Random Shape

The random shape block exists in the Shape drawer and can be used to change an agent’s shape to a random shape.

4.1.3 Set Clock

The set clock block exists in the Setup and Run drawer and can be used to set the clock time which appears in the status bar.
4.1.4 Clear Breed Name

A clear block for each breed was added to that breed’s drawer in the My Blocks panel. This block clears all agents of a specific breed. For example, a Clear Turtles block may now be found in the Turtles drawer and may be used to remove all Turtle agents from SpaceLand while leaving other agents untouched.

4.1.5 Inc Score

The inc score block increments the “score” variable shown in the status bar.

4.1.6 Dec Score

The dec score block decrements the “score” variable shown in the status bar.

4.2 Collision Blocks Now Found in Collisions Drawer

In StarLogo TNG 1.2, Collision Blocks for each breed were located in that breed’s drawer on the My Blocks page. However, we had observed that many users found this confusing and initially checked the Collisions drawer on the My Blocks page for these blocks. To avoid this confusion, in StarLogo TNG 1.5, Collision blocks are now located in the Collisions drawer.
Chapter 5

Evaluation

Throughout my work on StarLogo TNG, I have had several opportunities to observe the software in action, both before and after my additions were incorporated into the release. Of course, given that the majority of my thesis work was focused on the back-end of StarLogo TNG, very few users actually noticed any changes in this regard - but this may be a case where a lack of feedback may actually be positive feedback; the fact that we were able to run the new version of StarLogo TNG in so many contexts without most users noticing a change is a good indication that the new virtual machine is stable.

5.1 Imagination Toolbox January Refresher Workshop

The Imagination Toolbox January Refresher Workshop was a one-day workshop where local middle and high school teachers, most of whom had already been introduced to StarLogo TNG at one of our summer workshops, could come practice their StarLogo TNG skills, discuss issues they had run into using StarLogo TNG in the classroom, and take a tour of the new features included with StarLogo TNG 1.5.

One of the important issues that arose during the discussion was the quality of computing resources in many of the schools where StarLogo TNG is used. Many
school computer labs contain machines which are a few years old and low on memory and/or processor speed. Due to its graphical nature and the large number of computations that need to be run in order to handle interactions between up to 4096 agents, StarLogo TNG is a very memory-intensive application and the 25% memory savings afforded by the changes to the collisions implementation will undoubtedly be welcomed by these teachers.

Another common complaint voiced by several teachers was the bureaucratic hassle many of them faced when attempting to install educational software such as StarLogo TNG. Many of these teachers did not possess administrator privileges on their school computers and found that administrators were reluctant to install new software. While my thesis work does not directly address this problem, the StarLogo team hopes to address this in the future with a web-based version of StarLogo TNG that will not require installation at all. My work with the StarLogo TNG virtual machine will hopefully inform the design of the new WebLogo back-end.

5.2 IAP Game Programming Competition

Over MIT’s January Independent Activities Period, I organized a week-long competition for MIT students to design games in StarLogo TNG using StarLogo TNG 1.5. The purpose of this competition was threefold: to introduce MIT students to StarLogo TNG and raise awareness about the existence of this tool on campus, to acquire new games programmed in StarLogo TNG 1.5 that would make useful demonstrations of the software, particularly the new features, and of course, as a means of user testing this brand new release.

We provided the contest entrants with the following 100-point scoring rubric. The “Programmatically Interesting” category was specifically worded to encourage users to play with the new features incorporated in to StarLogo TNG 1.5:

- **Aesthetics (15)** – Game is aesthetically pleasing.
- **Coherence (10)** – Game makes sense, has a coherent theme/overall design.
• **Coding Style (5)** – Code is well labeled and organized, making it easy to read and understand.

• **Completeness (10)** – Game feels polished, complete – no obvious missing pieces or holes in gameplay.

• **Creativity/Uniqueness (15)** – Game idea is novel and exciting – stands out.

• **Difficulty Balance (10)** – Game strikes a balance between being challenging enough to make users want to keep playing, but not so challenging as to be discouraging.

• **Fun Factor (20)** – Game is fun and engaging – makes players happy.

• **Programmatically Interesting (5)** – Either showcases one of the following new features: Importing models from Google SketchUp or the new terrain editing panel, or does something else that’s really cool.

• **User Friendliness (10)** – Game is easy to play and understand.

The contest kickoff event had a good turnout, with several MIT students gathering during the software overview to ask technical questions about the limits of the system that the typical StarLogo TNG user probably never considers. However, there was an unfortunately low yield rate in actual contest submissions - a fact that may be attributable in part due to the timing of the competition, as IAP is generally a very busy time at MIT and our event was forced to compete for student attentions with several other events, but also due to the fact that despite the complexity of the simulations it allows, on the surface StarLogo TNG still looks like a child’s toy. A more professional-looking theme that plays down the brightly-colored graphical elements that have helped to make the software so attractive to a younger audience might help in this regard.
Chapter 6

Future Work

The new execution model will be incredibly beneficial to all users of StarLogo TNG, from grade school users to professional researchers. However, there are still many areas of improvement which need to be addressed in order for StarLogo TNG to truly be considered a competitive research tool. The following areas have been identified as those which would be most useful for implementing complex agent-based simulations.

6.1 Lists and Data Storage

The current implementation of StarLogo TNG is severely lacking in functionality with respect to lists, data storage, and graphing. The code in Figure 6-1 illustrates the necessary steps in order to remove an item from a list - a relatively common operation in simulations that was overlooked when writing StarLogo TNG. As illustrated in the diagram, the workaround for this is rather complicated and probably elusive to most novice programmers, but also time and memory intensive, as it requires making a complete copy of the list every time the user wishes to remove a single element.

As evidenced by this example, StarLogo TNG is missing some very basic functionality with regards to lists and data storage. The ability to remove items from a list easily is just one such example; right now, StarLogo TNG also lacks the ability to easily concatenate lists, sort lists, or create any other simple data structures besides lists. A full redesign of how lists are handled in StarLogo TNG may be needed so
Figure 6-1: A StarLogoBlocks function designed to remove an element from a list.
that complicated procedures such as the one shown in Figure 6-1 are not necessary to perform basic data storage functions.

6.2 Graphing and Data Visualization

StarLogo TNG currently contains very basic graphing functionality. There are StarLogoBlocks to create line graphs, bar graphs, and tables (see Figure 6-2). However, there are many potential additions that could improve the graphing and data visualization capabilities of StarLogo TNG. For example:

- The addition of other types of graphs (ie. histograms, scatterplots, etc.)

- The inclusion of a “clear graph” StarLogoBlock

- The ability to customize graph look and feel (ie. change titles, pen/bar colors, axis labels, etc.)

- Modifying how the graph scale auto-updates to improve efficiency or allow programmatic control
- Ability to write graphs programmatically with user-defined axes

- Ability to customize table look and feel, rearrange rows and columns, add new data from the table interface (rather than from StarLogoBlocks)

The addition of several or all of these features would be welcomed by teachers and students who wish to use StarLogo TNG for in-class science simulations as well as the small but growing number of “power users” that are starting to use StarLogo TNG simulations for professional and/or research activities.
Chapter 7

Conclusion

Among the many agent-based modeling systems available today, StarLogo TNG is unique in the way its graphical nature affords usability among even the most novice programmers, while the back-end still allows for highly complex simulations. Since its introduction in 2008, StarLogo TNG has not only allowed thousands of students and teachers to explore STEM concepts ranging from gravity to population ecology, but has also enabled more professional researchers to tackle complex concepts ranging from economics to infectious diseases.

The work in this thesis was designed with these more complex simulations in mind, in order to more realistically simulate the effects of several agents interacting in parallel. In order to accomplish this, I restructured the StarLogo TNG virtual machine to use a new execution model that relies on a duplicate copy of StarLogo TNG’s internal state. Each agent in the simulation now behaves as if it is the first to run on a VM cycle, creating the illusion that all agents are running in parallel. In addition to creating true internal parallelism, this new execution model was also responsible for a 25% overall memory savings in StarLogo TNG.

The Logo language has come a long way over the past four decades, and even today it continues to evolve. As we look toward the future, we hope to expand StarLogo TNG’s reach in two directions: a stronger, more powerful desktop-based version that builds upon the work I have begun to allow StarLogo TNG to handle more complex simulations, and a lightweight web-based version that will make graphical
programming more accessible than ever before. It is my hope that my work with the StarLogo TNG virtual machine will inform both of these developments as a small piece in the time-honored Logo legacy.
Appendix A

Code

A.1 turtle.h

Selections from turtle.h, StarLogo TNG 1.5

```c
typedef struct VisibleTurtle {
    sinum xcor, ycor, height_above_terrain;
    sinum heading;
    sinum color, shape;
    sinum xscale;
    sinum yscale;
    sinum zscale;
    sinum saytext;
    sinum bounding_radius, height;
    sinum bounding_height_high;
    sinum bounding_height_low;
    sinum owner;
    sinum transparency;
} VisibleTurtle;

typedef struct ShownTurtle {
    char shownp;
    char pendownp;
    char alivep;
    char rotatablep;
    char monitorp;
    char showskinp;
    char extra_space[2]; // this space is necessary to match up with memory allocated in 
                        // TurtleManager.java
} ShownTurtle;

/* Turtle heap is an array of simums, allocated by the compiler
   when it informs StarLogo of how many turtles-own variables
   there are. */
extern int num_turtles_owners;

typedef struct turtleState {
    VisibleTurtle vt;
    ShownTurtle st;
};
```
typedef struct VisibleTurtle {
    slnum xcor, ycor, height_above_terrain;
    slnum heading;
    slnum color, shape;
    slnum xscale;
    slnum yscale;
    slnum zscale;
    slnum saytext;
    slnum bounding_radius, height;
    slnum bounding_height_high;
    slnum bounding_height_low;
    slnum owner;
} VisibleTurtle;

typedef struct ShownTurtle {
    char shownp;
    char pendownp;
    char alivep;
    char rotatablep;
    char monitorp;
} ShownTurtle;
A.2  turtle.c

Selections from turtle.c, StarLogo TNG 1.5
int i; // i like to use this for looping
num_collisions = 0; // clears collisions
der = past_turtles + dec->next_offset;
vtde = &(dec->vt);
who = dec->who;
deexcor = SLNUMToLocalFloat(vtde->xcor);
decykor = SLNUMToLocalFloat(vtde->ycor);
deeheight = SLNUMToLocalFloat(vtde->height_above_terrain) + SLNUMToLocalFloat(get_point_height(deexcor, decykor));
der = SLNUMToLocalFloat(vtde->bounding_radius);
derhigh = SLNUMToLocalFloat(vtde->bounding_height.high);
derlow = SLNUMToLocalFloat(vtde->bounding_height.low);
breedIndex = getBreedIndex(dec->breed);
while (der >= past_turtles) { // check all breeds that would have collision points
  if (der->level_num != dec->level_num) { // skip others on different levels
    der = past_turtles + der->next_offset;
    continue;
  }
  vtder = &(der->vt);
  checkCollisionFlag = FALSE; // this one is not worth checking by default
  for (i = 0; i < numBreedCollisionInterests[breedIndex]; i++) { // loop through all breeds that would have collision points
    if (breedCollisionInterests[breedIndex][i].longnum == (dec->breed.longnum)) { // if this one is interesting
      checkCollisionFlag = TRUE; // flag it to be checked
      break; // and break to check
    }
  }
  if (checkCollisionFlag) { // if it was flagged to be checked then it is checked
    dx = deexcor - SLNUMToLocalFloat(vtde->xcor);
dy = decykor - SLNUMToLocalFloat(vtde->ycor);
dh = (deehight - SLNUMToLocalFloat(vtde->height_above_terrain) + SLNUMToLocalFloat(get_point_height((int)SLNUMToLocalFloat(vtde->xcor), (int)SLNUMToLocalFloat(vtde->ycor)))) * .5;
r = der + SLNUMToLocalFloat(vtde->bounding_radius);
f ((dx*dx + dy*dy) <= (r*r) &&
      ((dh < (SLNUMToLocalFloat(vtde->bounding_height.high) - derlow)) &&
       (dh > (SLNUMToLocalFloat(vtde->bounding_height.low) - derhigh)))
     add_collision_faster(dee, dec->who);
  }
  der = past_turtles + der->next_offset;
}
Turtle *FindAnyWho(int who) {
  // Return null if the who number is out of bounds.
  if (who < 0 || who >= max_turtles)
    return NULL;
  return past_turtles + who; // looks at old state
}
// This used to be *findAnyWho(int who), but *findAnyWho(int who) was changed to look at past_turtles
Turtle *findAnyWhoCurrent(int who)
{
    // Return null if the who number is out of bounds.
    if (who < 0 || who >= max_turtles)
        return NULL;
    return turtles + who;
}

A.3 vm.c

Selections from vm.c, StarLogo TNG 1.5

```c
void oncethrough(int frames_per_run)
{
    Turtle *t;
    Turtle *next;
    KeyboardData *this_cycle;
    int i, first_cycle;

    // copy turtle state before oncethrough.
    // (djwendel - need to copy ALL state (max_turtles), because live
    // turtles may be spread throughout the array, not packed into the front)
    memmove(past.turtle.heap, turtle.heap, max.turtles * num.turtles.own * sizeof(slnum));
    memmove(past.turtles, turtles, max.turtles * sizeof(Turtle));

    past.num.turtles = num.turtles;
    past.first.turtle = first.turtle;
    past.last.turtle = last.turtle;

    s1.time = s1.time + 1;

    changedPatchHeight = FALSE, changedPatchColor = FALSE,
    changedPatchTexture = FALSE;
    first.export.turtle = NULL;
    last.export.turtle = NULL;
}

// let the observer turtle run any setup code
int = turtles + past.first.turtle;
start(&observer);
dispatch(&observer, &t);

if (stopnow) {
    printf("Stopping now in oncethrough in setup\n");
    fflush(stdout);
    stopnow = FALSE;
    return;
}

// Set the erase flag to ALLOW, indicating that no one has executed any code.
// As agents run run-once code they will set it to false, indicating that it is not
// yes time to erase run-once code from the instruction list
eraseCond = TRUE;
//SDLDEBUG = TRUE;
```
// loop through and run code for each turtle in the list
while (t >= turtles)
{
    if (stopnow) {
        printf("Stopping now in oncethrough\n");
        fflush(stdout);
        stopnow = FALSE;
    return;
}

    // djwendel - ask the PAST turtle who should go next, but run that next turtle
    // in its future context
    next = turtles + (past.turtles[t->who].next_offset);
    update_collisions_for_one_turtle(t);
    start(t);
    while (!dispatch(t, &next));
    t = next;
}

flush_lines(index.in.line.buffer);
index.in.line.buffer = 0;

//SLDEBUG = FALSE;
// maybe_realloc_collision_array(): // free up collision array space if lots of agents have died
// Run the string/array heap garbage collector, if necessary
run_heap.gc();
// Now we're ready to inform starlogo that we ticked
vm.ticked();

if (first.export.turtle != NULL) {
    turtles.to.export();
}

if (my.keyboard.data != NULL) {
    this_cycle = (last_cycle.keyboard.data == &keyboard.data.cycle1) ? &keyboard.data.cycle2 : &keyboard.data.cycle1;
    memcpy(this_cycle->keys, my.keyboard.data->keys, NUMKEYS * sizeof(char));
    keyboard.delta.size = 0;
    first_cycle = SLNETWORKSTATUSCHANGED;
    SLNETWORKSTATUSCHANGED = FALSE;
    for (i = 0; i < NUMKEYS; ++i)
        if ((first_cycle && (this_cycle->keys[i] != last_cycle.keyboard.data->keys[i]))) {
            keyboard.delta[keyboard.delta.size++] = (this_cycle->keys[i] == 0) ? KEYBOARD_DELTA_DOWN(i) : KEYBOARD_DELTA_UP(i);
        }
    last_cycle.keyboard.data = this_cycle;
}

if (SLNETWORKED && keyboard.delta.size > 0) {
    keys.to.export();
}

Selections from vm.c, StarLogo TNG 1.2

void oncethrough(int frames_per_run)
{
    Turtle *t = first.turtle;
Turtle *next;
KeyboardData *this_cycle;
int i, first_cycle;
sl.time = sl.time + 1;
changedPatchHeight = FALSE, changedPatchColor = FALSE,
changedPatchTexture = FALSE;
first_export_turtle = NULL;
last_export_turtle = NULL;
}
next = t;
start(&observer);
dispatch(&observer, &t);
if (stopnow) {
    printf("Stopping now in once-through in setup\n");
    fflush(stdout);
    stopnow = FALSE;
    return;
}
eraseCond = TRUE; // Set the erase flag to ALLOW
while (t != NULL)
{
    if (stopnow) {
        printf("Stopping now in once-through\n");
        fflush(stdout);
        stopnow = FALSE;
        return;
    }
    next = t->next;
    start(t);
//printf("Turtle #\%d starting with once-through ip: \%d\n", t->who, SLNUMTOINT(IP));
    while(!dispatch(t, &next));
//printf("Turtle #\%d done with once-through ip: \%d\n", t->who, SLNUMTOINT(IP));
    t = next;
}
//SLDEBUG = FALSE:
//maybe_realloc_collision_array(); // free up collision array space if lots of agents have
died
//Run the string/array heap garbage collector, if necessary
run_heap_gc();

// Now we're ready to inform starlogo that we ticked
vm.ticked();

if (first_export_turtle != NULL) {
    turtles_to_export();
}

if (my_keyboard_data != NULL) {
    this_cycle = (last_cycle_keyboard_data == &keyboard_data_cycle1) ? &keyboard_data_cycle1 : &keyboard_data_cycle2;
    memcpy(this_cycle->keys, my_keyboard_data->keys, NUMKEYS * sizeof(char));
    keyboard_delta_size = 0;
    first_cycle = SLNETWORKSTATUSCHANGED;
    SLNETWORKSTATUSCHANGED = FALSE;
}
for (i = 0; i < NUM.KEYS; i++) {
    if (first_cycle || (this_cycle->keys[i] != last_cycle.keyboard.data->keys[i])) {
        keyboard.delta[keyboard.delta.size++] = (this_cycle->keys[i] == 0) ? KEYBOARD.DELTA.UP(i) : KEYBOARD.DELTA.DOWN(i);
    }
    last_cycle.keyboard.data = this_cycle;
}

if (SLNETWORKED && keyboard.delta.size > 0) {
    keys_to_xport();
}

A.4 TurtleManager.java

Selections from TurtleManager.java, StarLogo TNG 1.5

// Cached multiplications
private int turtleStateOffset;
private int turtleBouncesOffset;
private final ByteBuffer turtles;
private final ByteBuffer pastTurtles;

// Indices into turtles buffer

// Visible Turtle
private static final int XCOR = 0; // snum
private static final int YCOR = 8; // snum
private static final int HEIGHXTERRAIN = 16; // snum
private static final int HEADING = 24; // snum
private static final int COLOR = 32; // snum
private static final int SHAPE = 40; // snum string heap pointer
private static final int XSCALE = 48; // snum
private static final int YSCALE = 56; // snum
private static final int ZSCALE = 64; // snum
private static final int SAYTEXT = 72; // snum string heap pointer
private static final int BOUNDINGCYLINDER.RADIUS = 80; // snum
private static final int BOUNDINGCYLINDER.BOT = 96; // snum (distance from origin)
private static final int BOUNDINGCYLINDER.TOP = 104; // snum (distance from origin)
private static final int OWNER = 112; // snum
private static final int TRANSPARENCY = 120;

private static final int SHOWNP = 128; // char
private static final int PENDOWNP = 129; // char
private static final int ALIVEP = 130; // char
private static final int ROTATABLEP = 131; // char
private static final int MONITORP = 132; // char
private static final int SHOWSKINP = 133; // char
private static final int BREED = 136; // snum
private static final int XDELTA = 144; // double
private static final int YDELTA = 152; // double
private static final int NAME = 160; // double (temporary)
private static final int WHO = 168; // int
private static final int Slot = 172; // int
private static final int INCARNATION_COUNT = 180; // int
private static final int NUM_BOUNCES = 184; // int
private static final int NUM_COLLISIONS = 188; // short
private static final int PREV_OFFSET = 192; // pointer
private static final int NEXT_OFFSET = 196; // int
private static final int HEAP_OFFSET = 200; // int
private static final int BOUNCES = 208; // pointer
private static final int LEVEL = 212; // int
private static final int REF_COUNT = 216; // int // don't use this! use the incRefCount and decRefCount functions
private final ByteBuffer exportTurtles;
private int numTurtlesOwn;
private LongBuffer turtleHeap;
private LongBuffer pastTurtleHeap;
private LongBuffer turtleBounces;
// Indices into turtle bounces buffer
private static final int BOUNCE_XCOR = 0; // short
private static final int BOUNCE_YCOR = 1; // short
private static final int BOUNCE_HEIGHT = 2; // short
private static final int BOUNCE_HEADING = 3; // short
private LongBuffer turtleSounds;
private static final int SOUND_WHO = 0; // short
private static final int SOUND_NAME = 1; // short
private static final int SOUND_XCOR = 2; // short
private static final int SOUND_YCOR = 3; // short
private static final int SOUND_HEIGHT_ABove_TERRAIN = 4; // short
private ByteBuffer turtleCollisions;
private List<Variable> varList = new ArrayList<Variable>();
private static final int sizeOfTurtles = Cached multiplications
private int turtleStateOffset;
private int visibleTurtleOffset;
private int turtleBouncesOffset;
private int shownTurtleOffset;
private int turtleCollisionsOffset;
private final ByteBuffer turtles;
// Indices into turtles buffer
private static final int VT = 0; // pointer
private static final int ST = 4; // pointer
private static final int BREED = 8; // short
private static final int XDELTA = 16; // double
private static final int YDELTA = 24; // double
private static final int NAME = 32; // double (temporary)
private static final int WHO = 40; // int
private static final int HATCHED_WHO = 44; // int
private static final int SLOT = 48; // int
private static final int INCARNATION_COUNT = 52; // int
private static final int NUM_BOUNCES = 56; // int
private static final int NUM_COLLISIONS = 60; // int
private static final int COLLISIONS = 64; // pointer
private final LongBuffer visibleTurtles = 0; // pointers
private static final int XCOR = 0; // X coordinate
private static final int YCOR = -1; // Y coordinate
private static final int HEIGHT.ABOVE.TERRAIN = 2; // height above terrain
private static final int HEADING = 3; // heading
private static final int COLOR = 4; // color
private static final int SHAPE = 5; // shape
private static final int XSCALE = 6; // X scale
private static final int YSCALE = 7; // Y scale
private static final int ZSCALE = 8; // Z scale
private static final int SAYTEXT = 9; // say text
private static final int BOUNDING.CYLINDERLRADIUS = 10; // bounding cylinder LR radius
private static final int BOUNDING.CYLINDERBOTTON = 11; // bounding cylinder bottom
private static final int BOUNDING.CYLINDER.TOP = 12; // bounding cylinder top
private static final int BOUNDING.CYLINDER.TOP = 13; // bounding cylinder top
private final ByteBuffer exportTurtles;
private int numTurtlesOwn;
private LongBuffer turtleBounces = 0; // indices into turtle bounces buffer
private static final int BOUNCE.XCOR = 0; // bounce X coordinate
private static final int BOUNCE.YCOR = 1; // bounce Y coordinate
private static final int BOUNCE.HEIGHT = 2; // bounce height
private static final int BOUNCE.HEIGHT.ABOVE.TERRAIN = 3; // bounce height above terrain
private LongBuffer turtleSounds;
private static final int SOUNDWHO = 0; // sound who
private static final int SOUNDNAME = 1; // sound name
private static final int SOUNDXCOR = 2; // sound X coordinate
private static final int SOUNDYCOR = 3; // sound Y coordinate
private static final int SOUND.HEIGHT.ABOVE.TERRAIN = 4; // sound height above terrain
private ByteBuffer turtleCollisions;
private List<Variable> varList = new ArrayList<Variable>();
private final int sizeOfTurtle; sizeOfVisibleTurtle, sizeOfShownTurtle, sizeOfExportTurtle, sizeOfBounce, sizeOfSound;
private final int adjustedSizeOfVisibleTurtle;
private final int sizeOfCollision; sizeOfExportTurtle = StarLogo.sizeOfExportTurtle();
private static final int sizeOfBounce = StarLogo.sizeOfBounce();
private static final int sizeOfSound = StarLogo.sizeOfSound();

//private int sizeOfCollision;

Selections from TurtleManager.java, StarLogo TNG 1.2

private final ByteBuffer turtles; // Indices into turtles buffer
//private static final int VT = 0; // pointer
//private static final int ST = 4; // pointer
private static final int BREED = 8; // int
private static final int XDELTA = 16; // double
private static final int YDELTA = 24; // double
//private static final int NAME = 32; // double (temporary)
private static final int WHO = 40; // int
private static final int HATCHEDWHO = 44; // int
//private static final int SLOT = 48; // int
private static final int INCARNATION-COUNT = 52; // int
private static final int NUMBOUNCES = 56; // int
//private static final int NUMCOLLISIONS = 60; // int
//private static final int COLLISIONS = 64; // pointer
//private static final int PREV = 68; // pointer
//private static final int NEXT = 72 // pointer
private static final int HEAP = 76; // pointer
private static final int BOUNCES = 80; // pointer
private static final int LEVEL = 84; // int
//private static final int REFCOUNT = 88 // int // don't use this! use the incRefCount
// and decRefCount functions

private final ByteBuffer shownTurtles; // Indices into shown turtles buffer
//private static final int SHOWNP = 0; // char
private static final int PENDOWNP = 1; // char
private static final int ALIVEP = 2; // char
private static final int ROTATABLEP = 3; // char
private static final int MONITORP = 4; // char
private static final int SHOWSKINP = 5; // char
private static final int XCOR = 0; // double
private static final int YCOR = 1; // double
private static final int HEIGHTABOVE-TERRAIN = 2; // double
private static final int HEADING = 3; // double
private static final int COLOR = 4; // double
private static final int SHAPE = 5; // double
private static final int XSCALE = 6; // double
private static final int YSCALE = 7; // double
private static final int ZSCALE = 8; // double
private static final int SAYTEXT = 9; // double (temporary)
private static final int BOUNDINGCYLINDER-8ADIUS = 10; // double (roughly Mobile's Y coordinate)
private static final int HEIGHT = 11; // double (distance from origin)
private static final int BOUNDINGCYLINDERBOTTOM = 12; // double (distance from origin)
private static final int BOUNDINGCYLINDER-TOP = 13; // double (distance from origin)
private final ByteBuffer exportTurtles;

private int numTurtlesOwn;
private LongBuffer turtleHeap;
private LongBuffer turtleBounces;
// Indices into turtle bounces buffer
private static final int BOUNCE_XCOR = 0; // slnum
private static final int BOUNCE_YCOR = 1; // slnum
private static final int BOUNCE_HEIGHT = 2; // slnum
private static final int BOUNCE_HEADING = 3; // slnum

private LongBuffer turtleSounds;
private static final int SOUND_VHO = 0; // slnum
private static final int SOUND_NAME = 1; // slnum
private static final int SOUND_XCOR = 2; // slnum
private static final int SOUND_YCOR = 3; // slnum
private static final int SOUND_HEIGHT_ABOVE_TERRAIN = 4; // slnum

private ByteBuffer turtleCollisions;
private List<Variable> varList = new ArrayList<Variable>();
private final int sizeOfTurtle, sizeOfVisibleTurtle, sizeOfShownTurtle, sizeOfExportTurtle,
sizeOfBounce, sizeOfSound;
private final int adjustedSizeOfVisibleTurtle;
//private int sizeOfCollision;
Bibliography


