A Networked Development and Runtime Environment for StarLogo TNG

by

John Daniel Jackman, III

S.B. C.S., M.I.T., 2006

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

June 13, 2007

Author _____

Department of Electrical Engineering and Computer Science
June 13, 2007

Certified By _____

Eric Klopfer
er Education Program
Thesis Supervisor

Accepted By

mith
Professor of Electrical Eng ering
Chairman, Department Committee on Graduate Theses

A Networked Development and Runtime Environment
for StarLogo TNG
by
John Daniel Jackman, III

Submitted to the
Department of Electrical Engineering and Computer Science

June 13, 2007

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

StarLogo: The Next Generation aims to introduce students to programming. The StarLogo application is frequently used in a classroom setting, but the effectiveness of StarLogo is limited as long as students cannot interact with each other using the application. This thesis describes the design and implementation of a networked development and runtime environment for StarLogo. Hopefully, students will become more interested in programming when they can collaborate to create and run StarLogo programs. In addition, teachers will have more options available when choosing classroom activities using StarLogo.

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

Acknowledgments

This project would not have been possible without the support of the entire StarLogo development team. I would especially like to thank my advisor, Professor Klopfer, for his insight and patience throughout this project.

I also thank my family and Katelyn for their love and support over the years.

Table of Contents

Chapter 1: Introduction **7**

Chapter 2: Related Work **12**

Chapter 3: Design **13**
3.1 Base Layer Design 14
3.1.1 Agent Ownership 14
3.1.2 Agent Transfer 15
3.1.3 Control Signals 16
3.1.4 Canvas Sharing 17
3.1.5 Compiling and Linking 17
3.1.6 Remote SpaceLand Rendering 18
3.2 Middle Layer Design 18
3.2.1 Treatment of Agents with Different Owners 18
3.2.2 Sharing of Code Blocks 19
3.2.3 Conditions for Agent Transfer 20
3.2.4 Conditions for Remote Rendering 20
3.3 Top Layer Design 21
3.3.1 Establishing a Connection 21
3.3.2 Connected Worlds Model 22
3.3.3 Server/Clients Model 23
3.3.4 Additional Models 23
3.4 Fairness Considerations 24

Chapter 4: Implementation **26**
4.1 Existing Implementation 26
4.1.1 Virtual Machine 26
4.1.2 Block Compiler 27
4.2 Implementation Changes 28
4.2.1 Block Compiler 28
4.2.2 Agent Ownership 29
4.2.3 Agent Transfer 30
4.2.4 Control Signals 32
4.2.5 Canvas Sharing 32

4.2.6 Compiling and Linking....33
4.2.7 Treatment of Agents with Different Owners....38
4.2.8 Sharing of Code Blocks....39
4.2.9 Conditions for Agent Transfer....39

Chapter 5: Conclusions....41
5.1 Future Work....41
5.2 Applications....42

List of Figures

Figure 1: SpaceLand, the 3D runtime environment....8
Figure 2: The block programming environment....8
Figure 3: Two breeds: A bear throws paintballs at turtles....9
Figure 4: Connected SpaceLands form a shared runtime environment....10
Figure 5: Agent ownership unchanged....15
Figure 6: Agents being exported....31
Figure 7: Hosts sending compiled bytecodes....36
Figure 8: Server sending linked bytecodes....38

Chapter 1: Introduction

StarLogo: The Next Generation [3] is a modeling and simulation application that is constructed around a block-based programming language. The primary goals of the software system are to lower the barrier to entry for programming, entice more young people into programming, and create compelling 3D worlds for games and simulations. To achieve these goals, StarLogo TNG includes a virtual machine (VM) to execute programs written in the block language, a 3D environment, called SpaceLand (see Figure 1), to render the events that occur in the VM, and a user interface that allows users to construct programs visually by dragging blocks together (see Figure 2). Programs written for StarLogo involve agents, also known as turtles, which are grouped into breeds, and an environment, made up of "patches," in which the agents operate (see Figure 3). All agents of a particular breed execute the same sequence of instructions, but these instructions may have different effects as a result of the individual state of each agent. Agents interact with each other primarily through the use of collision code, which is created by the user and specifies the instructions executed by each agent when two agents collide in SpaceLand. The user has the option of including or not including collision code for every possible combination of breeds of which the colliding agents may be members.

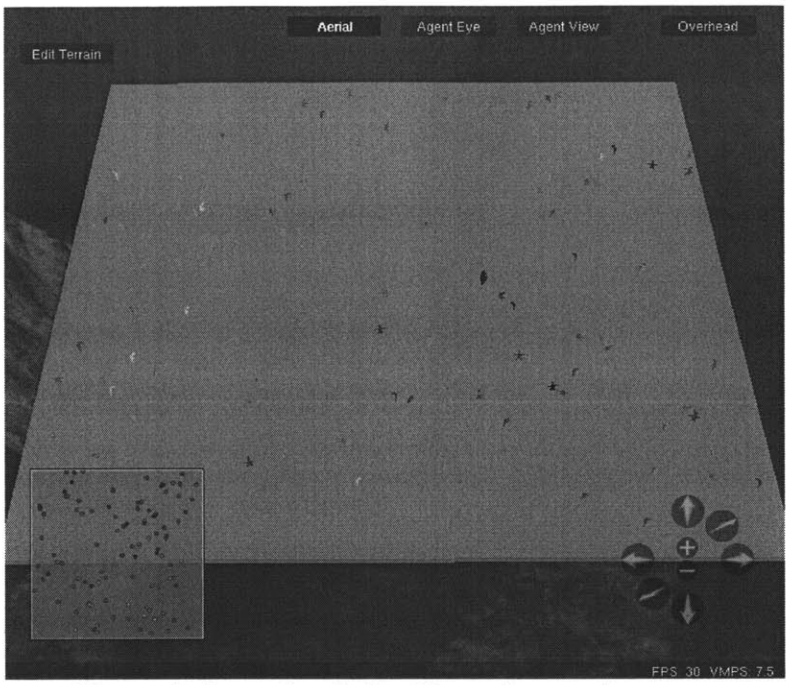

Figure 1: SpaceLand, the 3D runtime environment

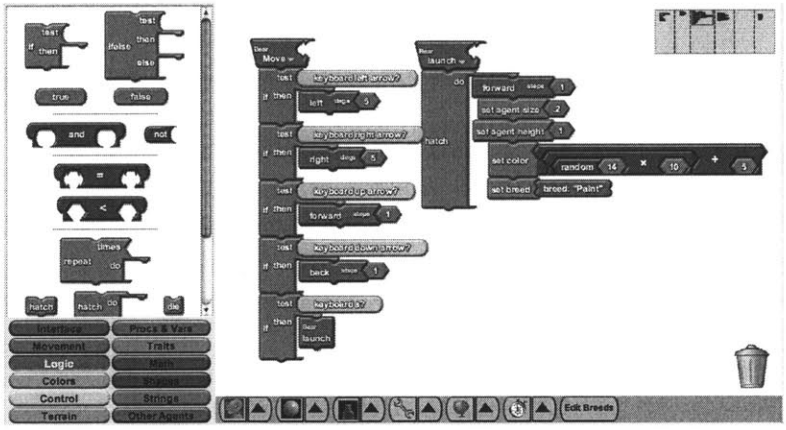

Figure 2: The block programming environment

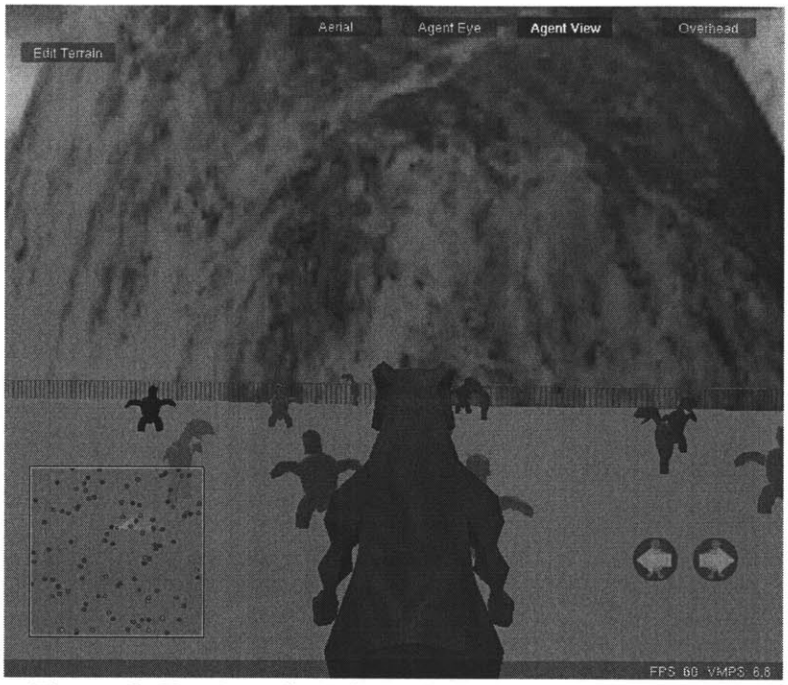

Figure 3: Two breeds: A bear throws paintballs at turtles

There are a variety of ways in which network functionality could be used to enhance the StarLogo application. These uses of networking include a shared space where students can collaborate to create programs, a common runtime environment where multiple students can simultaneously run their own programs that interact with each other (see Figure 4), and classroom exercises in which the teacher provides part of a program and can monitor or assist connected students as they complete and run the program. Given the range of potential applications of a network version of StarLogo, it is important that the design of this project be flexible enough to accommodate many use cases. For this reason, the primary design goals are extensibility, simplicity, and robustness.

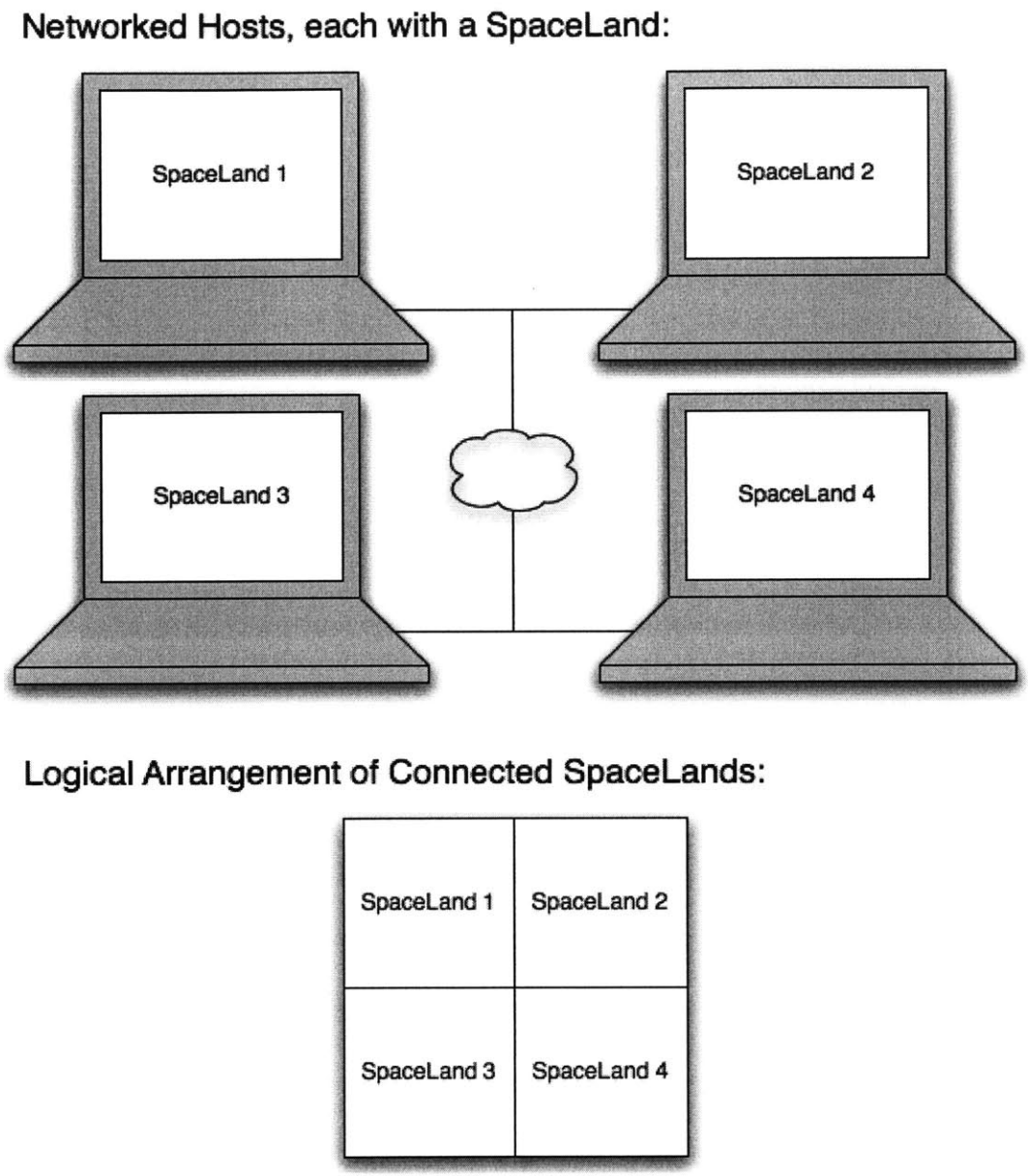

Figure 4: Connected SpaceLands form a shared runtime environment

The design used to add network capability to the StarLogo application consists of a suite of procedures that provide a variety of basic network operations, along with the

developer and user interfaces that are needed to configure the usage of these operations to achieve the desired network behavior. This approach allows developers to provide users with a wide range of applications of the StarLogo network capabilities, while requiring minimal effort from both the developers and the users. The use of abstraction to create the basic network operations is largely responsible for the simplicity of the design. The flexibility provided by the developer and user interfaces makes the design easy to extend without modifying or creating low-level network operations.

Chapter 2: Related Work

As its name suggests, StarLogo: The Next Generation is an offshoot of StarLogo [6], which draws its inspiration from Logo [5]. As an evolving project, StarLogo TNG is advancing in many ways to become more usable and allow more applications of the programming environment. Previous work on the project has mentioned the potential usefulness of collaborative programming for StarLogo [4]. Network functionality is just one of many features that is being actively developed for StarLogo TNG at this time.

There are also other projects with goals similar to those of StarLogo TNG. Kelleher and Pausch provide an excellent survey of programming environments that aim to lower barriers to programming [2]. Alice [1] and Scratch [8] are other prominent projects that feature program creation through a graphical drag-and-drop interface and a visual runtime environment. The Scratch project is also adding online interaction through NetScratch and ScratchR [7].

Chapter 3: Design

The design of the networked system has three layers of abstraction. At the lowest level are the procedures that provide the basic network capabilities. The middle level uses the low-level procedures to provide an interface for developers to implement a number of network operation models. The highest level is part of the user interface and allows users to select a model for network operation and use the application under that model.

Many aspects of the design are easier to present when described in the context of a specific application of a networked StarLogo. The paintball game depicted in Figure 3 serves as an excellent example. The program is part of the existing StarLogo library that can be extended into a multi-player game where each user controls an agent that is free to move across a grid of connected SpaceLands. In the interest of simplicity, the breed names from the figure will be used: the player-controlled agents will belong to a Bear breed, the projectiles will belong to a Paintball breed, and drone agents, which attack players, will be members of a Turtle breed. Each player will have one Bear that he controls using the keyboard and five Turtles that are controlled by an algorithm written by the player. Each player's Bear has a set number of lives when the game begins and loses one life when struck by a paintball. The last player to have his Bear alive wins a match.

3.1 Base Layer Design

The low-level network procedures allow several basic functions, including agent ownership, agent transfer, control signals, block canvas sharing, compiling and linking, and remote SpaceLand rendering.

3.1.1 Agent Ownership

Many applications of a networked StarLogo involve each user creating and controlling agents that behave differently than agents controlled by other users. This could be achieved if each user creates his own uniquely-named breed. However, such an approach would make it difficult to control common characteristics and interactions for all of the similar breeds. Therefore, the design allows each agent to be owned by a user.

The owner of an agent is the user that created the oldest ancestor of the agent. Therefore, when a user executes setup code that creates agents of any breed, all of those agents are owned by that user. Also, if any of those agents later create new agents through hatching, the new agents are also owned by the user, regardless of the SpaceLand in which the agents are hatched.

An agent's owner is largely responsible for which code the agent executes. When a user specifies the forever code for a particular breed, all agents of that breed which are owned by the user will execute the forever code, regardless of the SpaceLand where the agents are currently located, or where the agents may travel in the future (see Figure 5). The agent's owner is also the only user that can send keyboard input and other control

signals to that agent. Again, these signals are seen by the agent regardless of the SpaceLand where the agent currently resides. In the context of the paintball program, each player's Bear and Turtles are owned by that player. Because the ownership will not change as the agents move to other SpaceLands, a player's Bear will always be controlled that player's keyboard and a player's Turtles will always be controlled by that player's algorithm.

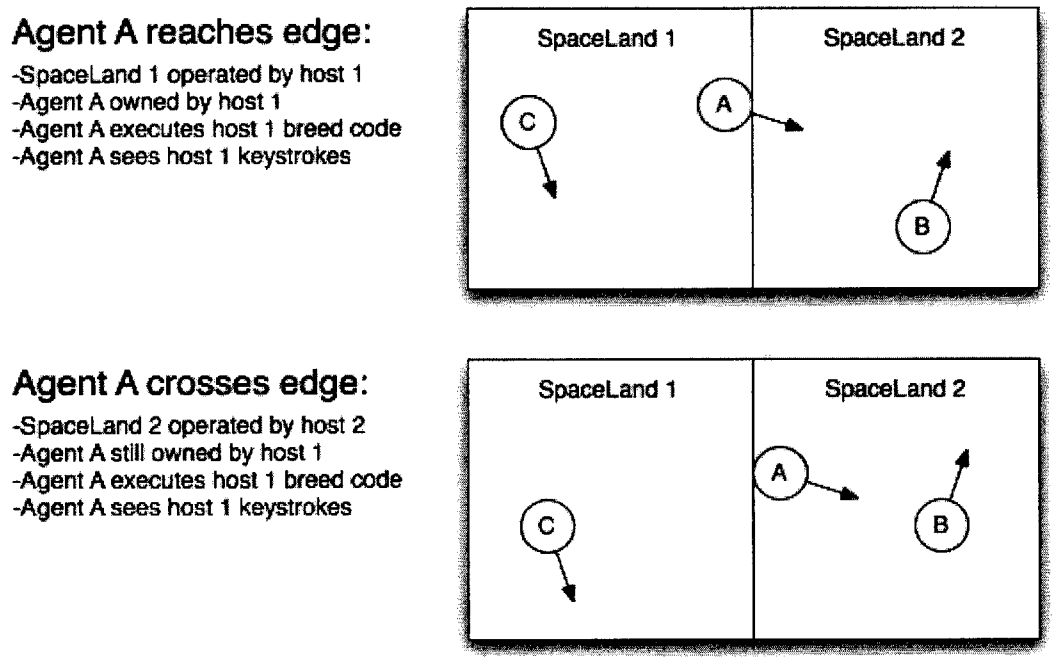

Figure 5: Agent ownership unchanged

3.1.2 Agent Transfer

When it is determined that an agent should move to another SpaceLand, most of the internal state of the agent must be transferred to the new SpaceLand. This situation usually

occurs when the agent reaches a connected boundary of its current SpaceLand. The state to be transferred includes the agent's breed, owner, variables, heading and position on the new SpaceLand, and rendering information. From the paintball example, the agent state includes the number of lives remaining for a Bear, so that variable will not change when a Bear moves to a different SpaceLand. Once the agent's state has been sent, the new SpaceLand creates an agent with the given state, while the previous SpaceLand destroys the agent.

3.1.3 Control Signals

StarLogo contains commands that let agents access user input from the keyboard. This has a number of applications, including user-triggered events, user-controlled agent movement, and user-adjusted parameters during execution. The presence of multiple users in a networked StarLogo necessarily results in multiple keyboards in the system. To address this issue, keyboard events from each host are broadcast to all other hosts during execution. This allows each host to maintain the relevant keyboard state for all other hosts. With this information known on each host, every agent has access to the keyboard state of its owner's keyboard, and uses that state when executing input commands. This mechanism is important to the paintball program because it allows a player to control his Bear with the keyboard at all times.

3.1.4 Canvas Sharing

A number of applications of a networked StarLogo rely on the presence of common rules that all agents of a particular breed must follow. To allow these rules to exist and be enforced, the design allows for a block canvas to be shared by all connected users. When a user makes a significant change to a shared canvas, the new canvas is broadcast to all other users.

3.1.5 Compiling and Linking

When a project is executed, the necessary compiling of code blocks is performed cooperatively by all connected hosts. Each host is responsible for compiling the code segments executed by agents owned by the host, along with any procedures present on the block canvas. However, any procedure calls, variables, or strings referenced by these code segments are not linked by the host during compilation. Instead, one of the hosts acts as a server at compile time by collecting the code segments from all of the hosts. The server then joins all of the code segments and links the procedure calls and variable references to create a single code section to be executed by all hosts. This code section is then distributed to all of the hosts, along with the union of the strings provided by each host. As a final step, each host allocates all of the strings on its own heap, then loads and executes the code section.

3.1.6 Remote SpaceLand Rendering

The design includes a mechanism that allows one host to choose to render the SpaceLand of another host instead of its own SpaceLand. When such a decision is made, the VM of the remote host collects the rendering data for each of its agents and the entire terrain. All of this information is then sent to the local host, which begins rendering the remote SpaceLand instead of its own. Once this initial setup is performed, the remote SpaceLand sends information about any changes to agent rendering data or the terrain as these changes happen, allowing the local host to render the updated SpaceLand.

3.2 Middle Layer Design

The middle layer of the design gives developers a simple interface to specify the four basic rules that govern a network model: treatment of agents from different clients, sharing of code blocks across clients, conditions for transferring agents between SpaceLands, and conditions for rendering a remote SpaceLand.

3.2.1 Treatment of Agents with Different Owners

The VM that executes agent code contains commands that can determine an agent's owner, similar to the commands that are used to determine an agent's breed. These commands can be used by the compiler to restrict execution of certain code areas by agent owner, in a manner similar to the use of breed commands to restrict execution. This design gives the compiler the flexibility to treat agents of the same breed with different owners in

a few ways. The compiler can choose to have all agents of a breed execute shared breed code regardless of owner, or the shared breed code can be used as a default mode that is executed by agents unless they are owned by users that have created their own breed code. If desired, the agents can execute a combination of code segments: each agent first executes the shared breed code, and then the code created by the agent's owner. Another mode of operation can have no shared breed code, where agents execute code only if their owner has created breed code.

In the context of the paintball application, all Bears and Paintballs will execute the same code, and each Turtle will execute the code provided by its owner. Each Bear will do nothing unless a keyboard command tells it to move or throw a paintball. Each Paintball will continue in a straight line until it hits something. A Turtle will perform some combination of moving and throwing paintballs, depending upon the algorithm provided by its owner. Also, when a Paintball collides with a Bear, the Paintball disappears and the Bear loses a life, regardless of which player owns each agent.

3.2.2 Sharing of Code Blocks

The presence of shared code blocks gives the compiler more options for dealing with the aspects of agent ownership. As referenced above, shared code blocks can override any other code that performs a similar function, act as a default mode that is overridden by similar code on an individual host, or be ignored completely. How much of a workspace is shared and the effects of shared blocks can be modified to achieve a desired network

model. In one extreme case, every page in the workspace can be shared, resulting in all users collaborating on a common project that will execute almost like a single user creating and running a project on a larger SpaceLand. At the other extreme, no pages in the workspace could be shared, resulting in a scenario similar to that which would occur if each user created his own project and ran the code on a large SpaceLand without any interaction with the other users' projects. More commonly, there will be a shared collision space that can be used to govern some of the interactions between agents created by different users, and a subset of the breeds will be shared to enforce uniform behavior in some aspects of the networked SpaceLand.

3.2.3 Conditions for Agent Transfer

The design includes an interface for specifying which edges of a given SpaceLand are connected to SpaceLands on other hosts, and which edges should be treated as walls. Given this information, when the agent reaches an edge of a SpaceLand, the VM knows either to transfer an agent to a specified host if the edge is connected or to cause the agent to turn around if the edge is a wall. This interface can be used to give a group of SpaceLands an arbitrarily connected geometry, which may or may not be physically possible.

3.2.4 Conditions for Remote Rendering

A user may choose to render a remote SpaceLand in one of two ways that mirror the

two existing methods for camera placement when rendering a SpaceLand. Currently, the user may choose either to fix the camera at a particular location (which may be changed by the user) or to have the camera follow an agent of the user's choice. Similarly, in the networked design, a user may choose either to select a particular SpaceLand to render locally (and then move the camera to any point in that SpaceLand) or to have the camera follow an agent owned by the user, always rendering the SpaceLand in which that agent is located. In the paintball example, the camera would follow a player's Bear so the player has the proper perspective to move his Bear and aim at nearby targets.

3.3 Top Layer Design

The top layer gives users access to the network functionality of StarLogo. This includes the user interface components that allow a user to establish network connections to other StarLogo users, then select a network model for the creation and execution of StarLogo projects. The users may also adjust some properties of a network model to match the desired behavior for a given project.

3.3.1 Establishing a Connection

There are a number of ways in which a user may find and connect to other StarLogo users. Given an out-of-band method of communication, users may connect directly to each other using a machine's hostname or IP address. Of course, this is not convenient or practical in many situations. For most classroom situations, a service discovery protocol

will be the simplest and most convenient mechanism to find and connect to other users on a local network. Eventually, a centralized StarLogo server could be used to find and connect to previously unknown StarLogo users with similar interests from around the world. Such a server would be most effective when coupled with a online community where users could share and discuss user-created StarLogo projects.

3.3.2 Connected Worlds Model

One of the most useful network models for a classroom setting is the Connected Worlds model. In this model, each user has a SpaceLand that is managed by his instance of the StarLogo application. The SpaceLands of all connected users are then arranged in a certain geometry that connects the SpaceLands to each other. This geometric arrangement could be as simple as a grid of SpaceLands, but it could also contain SpaceLands connected to themselves in a line to form a torus, and it could even connect the SpaceLands in a physically impossible arrangement if desired. The connected users can decide collectively upon an arrangement through a configuration dialog. Once the users have agreed upon an arrangement for their connected worlds, they can decide the behavior of shared code blocks. In some situations, shared code blocks will be used as a default implementation that is used by all agents unless the agent's owner has provided his own implementation to override the default. In other situations, shared code blocks will be executed by all agents, overriding any implementations by individual users. During execution, each user will be able to select which SpaceLand is rendered on their machine. The user can choose either to

view one SpaceLand, which may be his own or another user's, or to follow one of his own agents and always render the SpaceLand where that agent is currently located. The paintball program as described in this paper uses the Connected Worlds model, with each player's camera following his Bear, but could also be run using other models.

3.3.3 Server/Clients Model

Another useful network model for a classroom setting is the Server/Clients model. This model only includes the SpaceLand of one user, usually the teacher, that will act as a server. All other connected users are treated as clients, and these users will create all of their agents in the server's SpaceLand. Also, each user's machine will always render the server's SpaceLand. The server will decide the behavior of shared code blocks. Normally, when using this network model, the shared code blocks will always be executed as a means of enforcing a specific agent behavior for a certain aspect of the project. However, the server can also choose to treat shared code blocks as a default behavior that may be overridden or extended.

3.3.4 Additional Models

The simple nature of the middle layer of design results in a straightforward process that developers can use to create additional network models. All that is required is an additional option in the menu used for model selection, which will call the appropriate procedures in the middle layer to set up the new network model. If desired, configuration

dialogs for the model can be reused from other models or created if no such dialog already exists. This simple process will allow future developers to add or adjust models as needed.

3.4 Fairness Considerations

Much of the existing infrastructure of StarLogo TNG does not lend itself to a truly fair network design. There are commands that allow any agent to retrieve agent variables from any other agent. Also, any agent may kill any other agent. Removing or constraining these commands would break many existing projects created with previous versions of StarLogo. Furthermore, without these commands in their present implementation, new users would find it more difficult to achieve the desired behavior for some of their projects.

In light of the existing unfairness in StarLogo, the network design strives for fairness through transparency. By allowing users to decide collectively how much of a project's codebase is visible to all connected users and how much of the codebase is specified in a shared manner, the users themselves can decide upon and enforce a common set of rules. For example, if there is a breed of projectile agents that should kill any agents with which they collide, this behavior can be enforced by creating the desired collision blocks to govern these projectile collisions. These collision blocks can then be placed in a shared canvas where they will override any other such collision blocks created by individual users.

The effectiveness of transparency can be seen in the paintball program presented in this paper. The code executed by each Bear is present on a shared canvas and overrides

any code written by individual users, so all users playing can be assured that each player has the same capabilities when controlling his Bear. Similarly, the code governing Paintballs and Paintball to Bear collisions is shared and mandatory. Each player writes his own code to control his Turtles, but this code is visible to all other players to guard against algorithms that violate the spirit of the game. As a result, if a player used code that could be considered "cheating," other players would know because they could see the code. Such a provision is important because the existing StarLogo commands would allow a Turtle to find the position of any Bear on the current SpaceLand, jump to that location, and throw a paintball, all in one VM cycle. As mentioned above, removing or restricting these commands could solve this problem, but it would create problems for other existing and future StarLogo programs that require these commands. Therefore, the design gives users a powerful StarLogo application and the responsibility to police themselves instead of a restricted application that exhibits true fairness.

Chapter 4: Implementation

As previously mentioned, StarLogo TNG is an existing project. Implementing the above design requires a number of significant changes to the existing implementation of the StarLogo application.

4.1 Existing Implementation

The existing StarLogo TNG application is written in Java and C. The block programming environment is written in Java and is in the process of a code rewrite that will use Swing to render much of the interface. The block compiler is written in Java and uses JNI to interact with the VM, which is written in C. SpaceLand is written in Java, uses OpenGL to render the 3D world, and uses JNI to receive the latest rendering information from the VM.

4.1.1 Virtual Machine

The Virtual Machine (VM) on which agents execute code is implemented in C. The VM recognizes close to 300 primitives and commands. The basic unit of storage used by the VM is the SLNum (StarLogo Number), which is a 64-bit data word that can be used to store an integer, floating point number, or pointer. One array of SLNums is used to store

the code to be executed by agents, and another SLNum array is used as a heap that stores strings and lists used by the agents. The VM also contains an array of agents, with each agent represented by a data structure that stores the agent's individual state, consisting of breed, position, rendering information, and agent variables. In each VM cycle, the VM iterates through all agents. For each agent, the VM executes the commands in the code array, using the individual state of that agent when required by a command.

4.1.2 Block Compiler

The compiler is responsible for parsing the code blocks that the user has placed on the various block canvases and generating the appropriate bytecode commands for the VM. Nearly all blocks in StarLogo represent either commands or data. The compiler translates the command blocks to their respective bytecode commands. The compiler also adds the necessary stack operations to place the operands of a given StarLogo command onto the stack in the correct order.

The compiler also adds conditional statements to restrict the execution of some code segments. All agents begin execution at the same place of a shared bytecode array, but different behavior is desired for agents of different breeds. To provide this functionality, breed-specific code is wrapped in an "if" statement that checks the executing agent's breed, so the code is only executed by the desired breed. Thus, when the user wishes to compile code blocks for a number of breeds, the code for each breed is preceded by a breed test, and the resulting code segments, one for each breed, are placed sequentially

in the single bytecode array used by the VM.

4.2 Implementation Changes

The addition of network functionality to StarLogo and the concurrent redesign of the block programming interface required a number of changes and additions to the block compiler, the VM, the JNI interfaces, and the application as a whole.

4.2.1 Block Compiler

The block compiler was rewritten from the ground up due to the concurrent redesign of the StarLogo application. The redesign focused on storing more data in the Java objects representing the blocks, and this was achieved in part by defining the block language using an XML document and allowing the storage of arbitrary key-value pairs from the language specification in the block objects. This redesign simplified the process of adding new blocks to the language or modifying existing blocks by adding properties to a block object. A property can be added by using a key-value pair in the language specification, and the application can make use of the property by checking for the key-value pair.

The new compiler makes use of the block properties by including much of the compiler logic in the block objects themselves. The basic loop executed by the compiler starts with a block object. If the block is a command that has arguments, it compiles each block connected as an argument and creates code that will push the value of the argument

onto the stack. Inline procedure arguments, conditionally executed code segments, and direct operands are detected through the use of block properties and handled appropriately. The compiler then adds a bytecode for the command block, with the command being read as a property. For data and variable blocks, the values represented by the blocks must be retrieved and pushed onto the stack, which is done through one of a number of VM commands, depending upon the type of data. The specific commands used for these operations, as well as the presence of any special variables or constants, are stored as block properties.

The resulting logic loop present in the compiler is much simpler than the equivalent code from the previous compiler. This allows for more flexibility when developers decide to add new commands to StarLogo. All that is required is to add a command to the XML document containing the language specification and include the block properties needed by the compiler, then add the command to the VM with an appropriate implementation.

4.2.2 Agent Ownership

The implementation added an additional field to the individual state of each agent, and this field stores a number representing the agent's owner. The owner number can be retrieved using an "owner" command that was added to the VM, similar to the existing "breed" command. Also, an "owner-of" command was added that allows the retrieval of the owner number for an agent other than the agent executing the command. This is analogous to the "breed-of" command in the existing implementation. When an agent is

created directly by a user through the use of the "create-agent" block, the agent is given the number of the host executing the command for its owner field. When an agent is created by a "hatch-agent" command executed by an existing agent, the owner field is copied from the existing agent, which is considered the parent of the newly created agent.

4.2.3 Agent Transfer

Once supplied with the conditions that merit agent transfer, the VM checks agent actions against these conditions. When an agent performs an action that should result in transfer to another SpaceLand, the VM makes note of the agent and the host to which the agent should be sent. A list of these agents and their desired hosts is maintained by the VM over the course of a VM cycle. No other changes are made regarding these agents until the end of the cycle.

If there are agents to transfer at the end of the cycle, the VM notifies the StarLogo application that there are agents to be "exported" to another host. The application can then ask for the necessary data to export these agents, retrieving the information one host at a time. When given a request for the agents to be exported to a specific host, the VM prepares these agents for exportation by storing the agent data in an array of SLNums (see Figure 6). This array contains two parts. The first part contains the relevant parts of each agent's individual state. The second part contains all relevant heap data referenced by the agent state in the first part of the array. The heap data is obtained by scanning all SLNums in the agent state for pointers, and then copying the sections of the heap to which the

pointers refer, recursing in the case of a list that contains pointers. Once the array is complete, the data is given to the application to be sent to the proper host, and the agents that have been exported are destroyed by the VM.

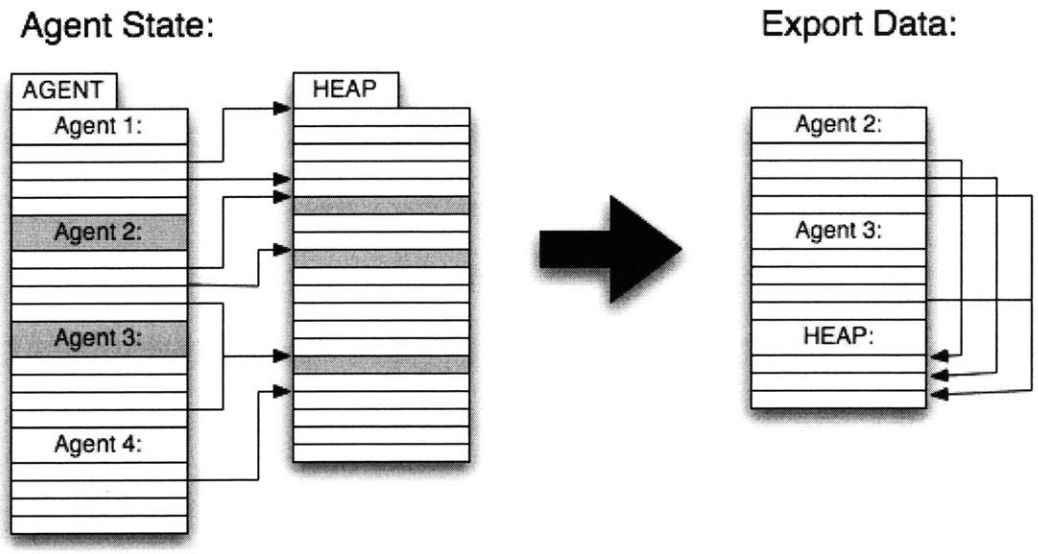

Figure 6: Agents being exported

When a host receives data for a group of agents that should be "imported" from another host, the application waits for the VM to complete its current cycle, then sends the SLNum array to the VM. Upon receiving the array, the VM creates the number of agents contained in the first part of the array, then copies the heap section of the array to the VM's heap. The VM then copies the agent state from the array to the newly created agents, and scans through the agent state and the heap, translating all pointers to point to the correct location on the VM heap. When the next VM cycle begins, the imported hosts are present in both SpaceLand and the VM.

4.2.4 Control Signals

The VM running on each host maintains the keyboard state of all connected hosts to ensure that each agent has access to the keyboard state of the agent's owner. When a network connection is first made, the VM for each host makes note of which keys are pressed on that host's keyboard. The VM then stores this information in an array, and the application retrieves the array and sends it to all other connected hosts. When a VM receives such an array, it stores the keyboard state, maintaining one array of key data for each connected host.

During execution, the VM sends out updates to keyboard state once per VM cycle. At the end of each cycle, the VM looks at the current keyboard state and compares it to the state at the end of the previous cycle. The VM then creates an array with the identifiers of all keys that have changed during the cycle and the current state of each of those keys. The application is notified that a change has occurred, at which point it retrieves the array and sends the information to all other connected hosts. When a VM receives such an array, it modifies the stored keyboard state to reflect the changes for the host that sent the data. Agents are provided with the key data from their owner by the VM. When each agent first begins code execution in a given cycle, the VM changes the pointer used to refer to keyboard state to point to the array of key data from the agent's owner.

4.2.5 Canvas Sharing

The implementation of block canvas sharing is relatively straightforward. There is

an existing interface that is used to listen for changes to the workspace as a whole, and this interface is used with a listener class that looks for changes to any canvases that are marked as a shared canvas. When such a change occurs, the listener serializes the shared canvas that triggered the event and sends the block data for that page to all other connected hosts. The blocks are serialized using the save format of the application, which encodes the blocks in the XML format used to define the block language. When a host receives such an update, it parses the data using the mechanism for loading projects, then updates the appropriate canvas with the new block data. The only modification required is to translate the block identification numbers. These number are required to be unique, so the incoming blocks are assigned new ID numbers that won't conflict with existing blocks in the workspace.

4.2.6 Compiling and Linking

The actions of the block compiler are largely responsible for the behavior of the networked StarLogo application. The responsibility for compiling the necessary blocks is distributed among all connected hosts, with each host primarily responsible for compiling the blocks unique to that host and generating metadata to assist in integrating the compiled code into the larger network project.

When a host compiles a project for networked execution, the compiler delays completion of certain tasks to simplify the final integration and linking of bytecodes. First, a list of all agent, patch, and global variables is created from the variable declaration blocks

in the workspace. Whenever a reference to one of these variables is compiled, a marker is inserted instead of an index number that would normally identify the variable, and a table that maps the markers to the appropriate variable names is created. After handling the variables, the compiler reads all procedure declarations and compiles the procedure code into a bytecode array. The compiler also maintains a table that maps each procedure name to its position in the code array. As is done with variables, when a procedure call is compiled, a marker is inserted instead of the address of the procedure, and a table maps the markers to the procedure names. In similar fashion, any blocks that reference strings are represented by markers, and a table maps these markers to the strings that must be allocated on the VM heap.

After the compiler reads all of the variables, procedures, and strings, it compiles the breed code. The compiler gathers all the blocks that represent breed code and compiles those blocks that have been switched into the "on" position by the user, indicating that the code should be run. For each of these blocks, the compiled code is wrapped in a conditional statement to ensure that only agents of the specified breed will execute the commands. For example, if a group of blocks is located on the workspace page for the "Turtles" breed, the compiler will add VM commands to retrieve the executing agent's breed and compare the returned value to the "Turtles" string. If the agent is in fact a Turtle, it will execute the commands represented by the code blocks. Otherwise, the agent will skip the commands and resume execution after the "if" statement. In this way, all of the breed-specific code is compiled as a single sequence of conditional statements, and when

this code is executed by an individual agent, the agent will only execute the code for its own breed. Additionally, the default mode for network execution is that each host's breed-specific code is only executed by agents owned by the host. To accomplish this, the conditional statement is augmented to check the agent's owner in addition to its breed.

Once the breed code has been compiled, the compiler must handle collisions and send all of the code and metadata to the host acting as a server during the compilation process. To do this, the compiler scans the workspace for all collision blocks, ignoring any blocks that are overridden by shared collision blocks handling the same type of collision. For each collision block, two procedures are created: one for each agent present in the collision. These procedures are added to the procedure table, and the pair of breeds governed by the collision are added to a list of collisions handled by the host. Once this process is complete, the host is ready to send all of its compilation data to the host acting as server. The compiler sends the array of compiled bytecodes, the procedure table, the marker tables for variables, procedure calls, and strings, a list of breeds known to the host, and the collision list to the server (see Figure 7).

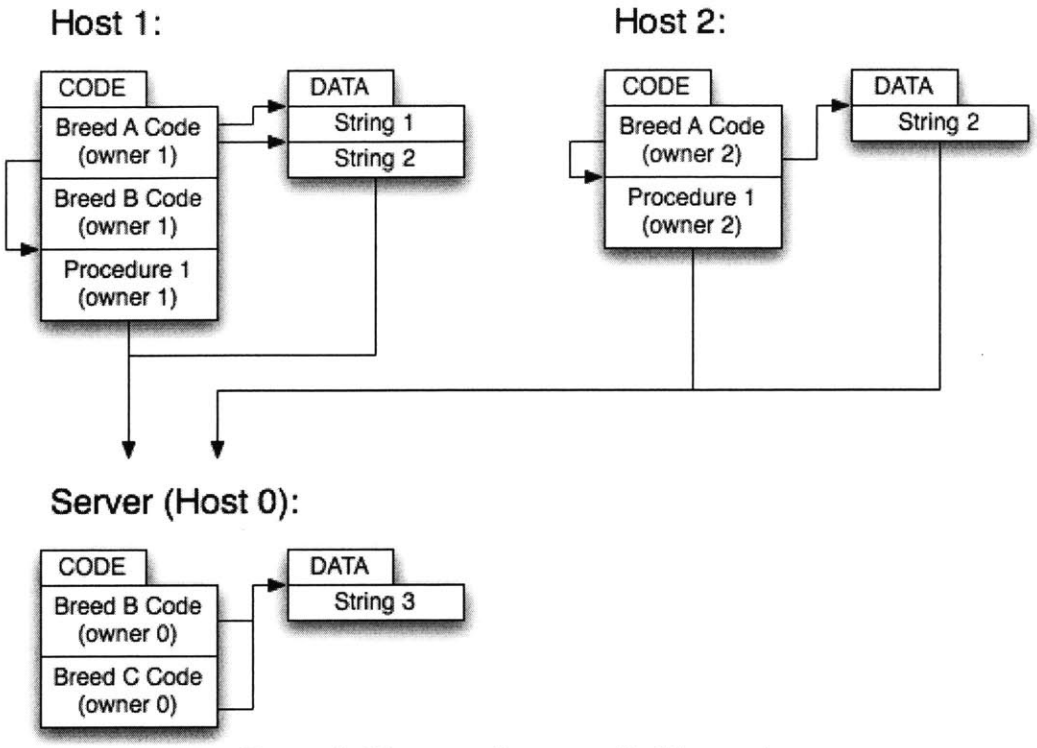

Figure 7: Hosts sending compiled bytecodes

The server collects compilation data from all of the hosts and produces a single code array to be run on all hosts. First, the server creates a code array by concatenating the code arrays from all hosts, including the server. The code array is still separated by code section, however. All procedures form a single section that contains only procedure code, and all breed-specific code is in a single section that only contains breed code. In the process of joining these segments, the addresses of individual bytecodes will change, so all procedure table entries and code markers are translated when the code arrays are joined. The server then creates three variable lists for the three types of variables: agent, patch, and global, by taking the union of the variable lists from all hosts. The index of a variable in

one of these lists will be the index used when a command refers to the variable. At this point, the procedure table and the variable lists contain the final values for procedure addresses and variable indices, respectively, that will be used in VM code.

The server then links the compiled code and sends the resulting code array to all hosts (see Figure 8), which then execute the code. Linking is performed by replacing each procedure marker in the code array with the address of the named procedure, then replacing each variable marker in the code array with the index of the named variable. The server then sends all hosts the single code array, along with the unified variable list, a unified breed list, and a unified string marker table. Upon receipt of this data, each host will first provide the application and the VM with the unified breed list and the unified variable list. Then, the host will allocate each string in the table onto the heap, and replace the markers for that string with the allocated address. The host then sends the finalized code array to the VM for execution.

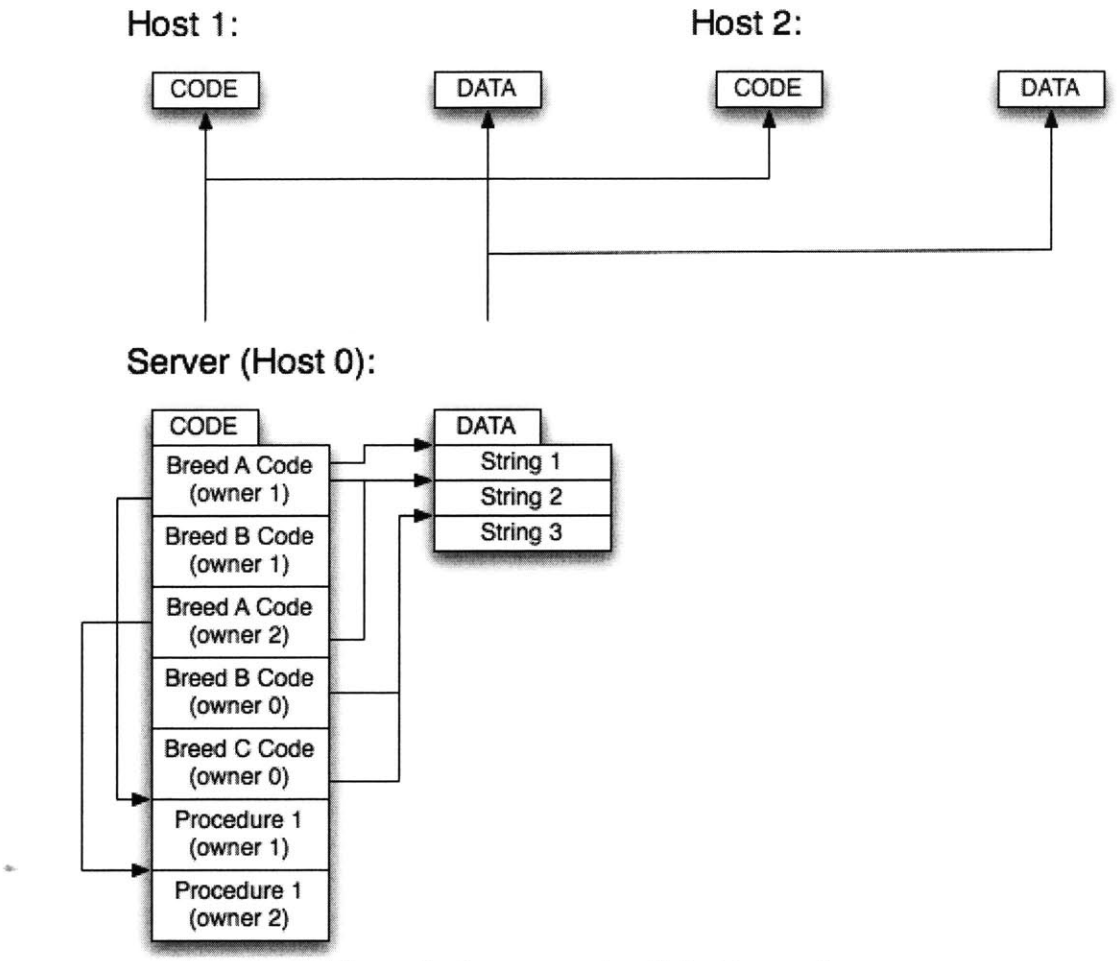

Figure 8: Server sending linked bytecodes

4.2.7 Treatment of Agents with Different Owners

The block compiler contains a procedure for compiling breed-specific code. The procedure has been modified to take three arguments: a breed name, an owner number, and a sequence of code blocks. If no owner is provided, the procedure will compile the given code blocks for any agent whose breed name matches the given breed name. This is done

by wrapping the compiled code in a conditional statement that checks the agent's breed using the "breed" command. If an owner number is provided, the procedure compiles the code blocks only for agents of the given breed that are owned by the specified host, achieving this result by extending the conditional statement to check the agent's owner using the "owner" command.

4.2.8 Sharing of Code Blocks

The interface that controls block canvas sharing is relatively simple. Each canvas is marked with a boolean flag that determines whether or not it should be shared with all connected hosts. The status of the flag is set to a default value for each canvas when the application begins execution, and may be changed by the users if desired. When an event occurs on a canvas, a listener checks to see if the canvas is shared and broadcasts the updated canvas to all connected hosts if necessary.

4.2.9 Conditions for Agent Transfer

The interface used to specify the conditions for agent transfer consists of a procedure that connects one edge of the local SpaceLand to the opposite edge of any remote SpaceLand. The procedure does so by notifying the VM of this connection so that the VM may check for agents that reach the edge and export these agents to the specified host, setting the location information for these agents to the appropriate position along the opposite edge of the remote SpaceLand. The application also includes a mechanism for

calling this procedure remotely. This allows one host to connect one of its edges to a remote SpaceLand, then connect the opposite edge of that remote SpaceLand to the local SpaceLand, which enables agent transfers in both directions.

Chapter 5: Conclusions

StarLogo TNG is a continuing project that is approaching its first beta release. The network design as described in this paper is not yet fully implemented, but the current implementation is still quite usable for many applications. As the implementation is completed in the coming months and extended as needed in the future, the range of applications of a networked StarLogo will only grow.

5.1 Future Work

A few aspects of the network design remain unimplemented. The top layer of the design has not been implemented and integrated into the StarLogo user interface, so users can only run programs in a default Connected Worlds model. Also, the ability to render a SpaceLand remotely has not yet been implemented. These shortcomings are primarily the result of time constraints, but the implementation of the complete network design should proceed within the context of the ongoing redesign of the StarLogo user interface and will take into consideration the user experience in a networked StarLogo, which will develop with further user testing.

5.2 Applications

There is a significant existing library of both educational and recreational StarLogo TNG resources [3]. These resources include structured classroom activities that introduce students to the StarLogo interface and capabilities, tutorials that help users create games, and sample programs that showcase the powerful modeling capabilities of StarLogo. These modeling simulations are particularly useful in conjunction with a science curriculum because StarLogo can model complex ecosystems involving thousands of agents with relatively little effort from the user.

There are a number of potential uses for the network capabilities in the StarLogo application. Many of these usage scenarios are based on a competitive game. The paintball example presented throughout this paper is one such game. This basic "hunting" concept could be extended to involve more strategic decisions that would be needed for a "capture the flag" game where each player must hide and defend his own flag in his SpaceLand, while at the same time attempting to discover and retrieve other players' flags which are hidden in their own SpaceLands. Furthermore, the network capabilities of StarLogo would allow users to develop pure strategy games like checkers and chess, but with more flexible rule sets.

The nature of the StarLogo block programming language will also allow the creation of games and competitive scenarios that involve no user input when run. All of the games listed above could be modified such that each user must write algorithms governing the behavior of his own agents in a variety of situations. These automated

agents could then compete against the automated agents of other users to examine the merits of each algorithm. In the simplest form, a Prisoner's Dilemma contest could be constructed to evaluate different algorithms for the scenario. However, the concept of algorithms that govern automated agents could be extended to the hunting, capture the flag, and more complex strategy games mentioned above. Network capabilities also create the possibility for collaborative simulations using the StarLogo application. Such simulations could be particularly useful for extending the existing ecosystem models into a multi-user invasive species simulation. Each user could create both his own ecosystem on his local SpaceLand, attempting to make it resistant to invasive species, and an invasive species which he could then unleash to see how it disrupts other ecosystems.

References

[1] Alice. 2007. "Alice: Free, Easy, Interactive 3D Graphics for the WWW." http://www.alice.org/.

[2] Kelleher, C. and Pausch, R. 2005. "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers." *ACM Computing Surveys*, 37(2). 83-137.

[3] Klopfer, E. 2007. "StarLogo TNG." http://education.mit.edu/starlogo-tng/.

[4] McCaffrey, C. 2006. "StarLogo TNG: The Convergence of Graphical Programming and Text Processing." MEng Thesis, MIT.

[5] Papert, S. 1980. *Mindstorms: Children, Computers, and Powerful Ideas.* Basic Books, New York.

[6] Resnick, M. 1996. "StarLogo: An environment for decentralized modeling and decentralized thinking." *Human Factors in Computing Systems*, Vancouver. 11-12.

[7] Scratch. 2007. "Lifelong Kindergarten: MIT Media Lab." http://llk.media.mit.edu/projects.php.

[8] Scratch. 2007. "Scratch: imagine, program, share." http://scratch.mit.edu/.