Three Dimensional Terrain Editing in StarLogo TNG

Michael D’Ambrosio
S.B. CS, M.I.T., 2007

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, 2009

©2009 Massachusetts Institute of Technology
All rights reserved.
ABSTRACT

StarLogo TNG is a block based programming language designed to create models and simulations that are used to help middle and high school students understand mathematical and scientific concepts. However, there is a desire to use the program to help facilitate game creation but the tools do not exist. This thesis evaluates and discusses the design of some of the tools necessary in order to create richer 3-D environments and games within StarLogo TNG.
Table of Contents

1. Introduction ................................................................................................................. 6
  1.1 What is StarLogo? .............................................................................................. 6
  1.1.1 Patches, Agents and the User ........................................................................... 7
  1.1.2 The Result ........................................................................................................ 8
  1.1.3 Problems ....................................................................................................... 8
  1.2 What is StarLogo TNG? .................................................................................. 9
  1.2.1 StarLogoBlocks ............................................................................................. 9
  1.2.2 Spaceland ...................................................................................................... 11
2. Previous Work .............................................................................................................. 12
3. User Classes .............................................................................................................. 13
  3.1 Students ............................................................................................................. 13
  3.2 Teachers .............................................................................................................. 13
  3.3 Researchers ....................................................................................................... 14
4. Feature Proposal ........................................................................................................ 15
  4.1 Creation/Placement of Agents ............................................................................. 15
  4.2 Editing Dimensions of Spaceland ....................................................................... 16
  4.3 Undo / Command History .................................................................................. 16
  4.4 Reset .................................................................................................................. 16
  4.5 Copy / Paste Terrain ......................................................................................... 17
  4.6 Texture Terrain .................................................................................................. 17
  4.7 Graffiti Patch Walls ........................................................................................... 18
  4.8 Custom Patches .................................................................................................. 18
  4.9 Load Level ......................................................................................................... 18
  4.10 Edit Skybox ...................................................................................................... 19
  4.11 Other Features ................................................................................................... 19
5. Measuring a Good Interface ...................................................................................... 20
  5.1 Learnability ......................................................................................................... 20
  5.2 Memorability ....................................................................................................... 20
  5.3 Efficiency ............................................................................................................ 20
  5.4 Errors .................................................................................................................. 21
  5.5 Satisfaction ......................................................................................................... 21
  5.6 What is Important for Us? .................................................................................. 21
6. A Tour of Spaceland .................................................................................................. 23
7. Lawrence Workshop .................................................................................................. 32
  7.1 Changes to Spaceland in Response to Lawrence Workshop .............................. 33
  7.2 Reception .......................................................................................................... 35
8. Load Level .................................................................................................................. 37
  8.1 Design Iteration .................................................................................................. 37
  8.2 Final Solution – Spaceland Side .......................................................................... 42
  8.3 Final Solution – StarLogoBlocks Side ............................................................... 46
  8.4 Reception – Teacher Workshop .......................................................................... 49
  8.5 Responses to Unforeseen Issues ......................................................................... 49
9. Saving and Loading Agents ....................................................................................... 52
List of Figures

Figure 1 -- A Simple StarLogo TNG Program ................................................................. 10
Figure 2 -- Default Spaceland View .............................................................................. 23
Figure 3 -- Pan, Rotate and Zoom ................................................................................... 25
Figure 4 -- Edit Terrain Menu ....................................................................................... 26
Figure 5 -- Color Region Menu ..................................................................................... 27
Figure 6 -- Raising and Lowering the Terrain ............................................................... 28
Figure 7 -- Mounds and Pits ............................................................................................ 29
Figure 8 -- Agent Monitor Window ................................................................................ 30
Figure 9 -- Example Terrain .......................................................................................... 31
Figure 10 -- Lawrence Changes .................................................................................... 33
Figure 11 -- Swap Views Button ................................................................................... 34
Figure 12 -- Block Drawers ........................................................................................... 38
Figure 13 -- Forever and Run Once on Canvas ............................................................ 40
Figure 14 -- Forever and Run Once in Runtime Workspace .......................................... 41
Figure 15 -- Levels Tab .................................................................................................. 42
Figure 16 -- Editing a Level’s Name .............................................................................. 44
Figure 17 -- Levels Blocks ............................................................................................. 47
Figure 18 -- Set Level Blocks ....................................................................................... 51
Figure 19 -- Creating Two Agents (Step 1) .................................................................. 56
Figure 20 -- Creating Two Agents (Step 2) .................................................................. 57
Figure 21 -- Creating Two Agents (Step 3) .................................................................. 58
Figure 22 -- Edit Breeds Window ................................................................................... 59
Figure 23 -- Agents Tab ................................................................................................. 60
Figure 24 -- Bear Highlighted ....................................................................................... 61
Figure 25 -- Open and Close Hand Cursor ................................................................... 62
Figure 26 -- Deleting an Agent ..................................................................................... 64
Figure 27 -- Changing an Agent’s Shape ....................................................................... 65
1. Introduction

For my MEng project, I will work with the StarLogo TNG ("The Next Generation") group. StarLogo TNG, evolved from an older program, simply titled "StarLogo". My project will revolve around building a more functional three dimensional terrain editor for use within StarLogo TNG. However, before we can talk about StarLogo TNG, we will first explore its predecessor, StarLogo.

1.1 What is StarLogo?

StarLogo can be explained neatly with the following quote:

StarLogo is designed especially for helping people create models of decentralized systems—that is, systems in which patterns arise from interactions among lots of individual objects. For example, StarLogo is well designed for exploring how bird flocks arise from interactions among individual birds, or how traffic jams arise from interactions among individual cars [1].

There are many more uses for StarLogo [6]. In essence, StarLogo was designed to model decentralized systems. StarLogo’s user base included both children and adults. Children would use StarLogo in order to learn about models and adults would typically create the models. The aforementioned adults could either be the children’s teachers or even researchers who were modeling something of interest to them. At the center of StarLogo lied three sources of interest, patches, agents and the ubiquitous observer (commonly referred to as "you, the user") [9].
1.1.1 Patches, Agents and the User

Patches made up the world in which everything lives. The world consisted of a two-dimensional array of patches and these patches had several pertinent properties. For example, a patch had a notion of its own color. A patch's internal state could be acted upon by either agents or the observer (i.e. the user could change a patch's color). Patches represented physical things such as mounds or hedges (if we were designing a maze).

Agents are the objects that populate the system. Each agent had a set of rules which it followed, independent of any other agent in the system. For example, an agent could be programmed to walk forward until it hits a wall, then spin around in a circle five times and then say "I don't like spam". However, an agent had a notion of a species (e.g. a turtle or a bird) and it knew about all of the other species in existence. A species roughly translated to a Java class object. An agent also had a mechanism for dealing with collisions between other agents of any species. We could have very easily modified the above example to make our agent say "Hello" if it were to collide with an agent of the same species instead of spinning around and doing silly things. Models could be created through the interactions of individuals. However, there was no being that controlled the group as a whole. Agents typically had a visual representation, but they needed not have one. Since agents could modify the patches that they stood on, they could mark the ground as a record of their passing over a particular patch and remain invisible.

The observer was you. You knew all about the state of the system. You had access to all of the terrain data (and could change it at will, even during simulation). You could also view properties of agents and change them on the fly. For example, an agent may have had a particular x and y coordinate associated with it. The observer could freely change that data and make the agent move somewhere else. However, the agents still ran their code so their internal state still changed every time tick.
1.1.2 The Result

The end result is a totally decentralized, and grossly parallel, system. The benefit of a parallel system is that it can be run on a multi-processor system and that’s exactly what the earliest version of StarLogo was designed for [7]. However, the fates intervened and the code was switched to the Macintosh system (in a version not surprisingly called “MacStarLogo”) [7]. Soon, the creation of the Java programming language allowed StarLogo to be re-implemented in a platform and processor independent manner [7]. StarLogo still exists today at version 2.2 [3].

1.1.3 Problems

The single largest problem with StarLogo was that the programming interface was textual. This was very problematic for younger children who had no prior programming experience. There was a steep learning curve involved. Also, with any text based interface, errors were very common. Users needed to consult the reference manual for a variety of reasons. For example, they may be new and were learning the system. They could also have been very experienced and simply had a slip of memory and had to recall the right keyword to perform the task at hand. One might try to argue that this isn’t a problem as children are very smart and since they play video games often, they should have some idea how to write them as well. Unfortunately, this is not the case as children generally have no idea how to create a video game [5].

The other contributing factor to StarLogo’s demise was the fact that the graphics (representing the terrain and agents) were 2-dimensional, small and rather unpleasant on the eyes. One of StarLogo’s goals was to entice children to learn and to let them experiment with the models. However, many children were interested in learning how to program and make their own games. StarLogo’s graphics simply didn’t have the “coolness factor” needed to latch onto this demographic of students [5].
1.2 What is StarLogo TNG?

Development of StarLogo TNG started around 2002 [7]. It was designed to address the problems of StarLogo. The goals of StarLogo TNG are as follows [4]:

1. Lower the barrier to entry for programming by making programming easier.
2. Entice more young people into programming through tools that facilitate making games.
3. Create compelling 3D worlds that encompass rich games and simulations.

There are two main components to StarLogo TNG: StarLogoBlocks and Spaceland [5]. They are described below.

1.2.1 StarLogoBlocks

StarLogoBlocks was designed to remove the need to use a command language in order to create a program. Textual commands were eliminated in favor for a set of puzzle pieces that represented the set of all things you could do with StarLogo TNG. These puzzle pieces could be stitched together in order to create programs. This eliminates all syntax errors. If two puzzle pieces don’t fit together, then they can’t be used together. Likewise, if a particular piece has an empty socket, then a parameter is missing and one should be supplied. Furthermore, the blocks are arranged in several bins and can be accessed from there. This means that a user can find every block that exists in StarLogo TNG without having to consult outside sources. Tooltips accompany the blocks so the “documentation” is easily accessible from within StarLogo TNG.
Figure 1 -- A Simple StarLogo TNG Program

The above figure shows a very simple StarLogo TNG program. In this program there is a forever block. This block means that its body will be executed forever (until otherwise told to stop). Because the label (denoted at the upper right hand corner of the forever block) says “Turtles”, this means that this body will be executed only for Turtles. If, for example, we had another breed, such as bears, then bears would not follow this program. They would have to be programmed separately. The program itself simply checks for keyboard input (either the up, down, left or right arrows) and if an arrow is pressed, all Turtles will move accordingly (either forward, back, or turn to the left or right).
1.2.2 Spaceland

Spaceland is the world in which the agents live. However, unlike the world in StarLogo, Spaceland is rendered three-dimensionally using OpenGL. This means that patches have a height in addition to their x, y coordinate pairs. However, for any given x, y pair, there is exactly one height associated with it so Spaceland is actually a 2.5-D grid of patches [9]. The hope was that the addition of a three-dimensional world, children would be more interested in using StarLogo TNG [5].

Of course, middle school aged children using StarLogo TNG won’t become expert game programmers overnight. However, they can learn many concepts about programming without being bogged down by syntax and graphics, which typically involve heavy math skills. As a matter of fact, I used a similar program when I was in middle school (on my own, personal, time) to tinker with game programming. It was called RPG Maker 2000 [2] (the reference is to a similar, but newer program). Even though the graphics were two-dimensional, reminiscent of the SNES era, event scripting was done with a point and click interface (albeit not with blocks). By using that program, I taught myself many basic programming concepts such as: if statements, for loops, and variables. I never actually learned a formal programming language (C++) until I was halfway through high school and even then, my projects weren’t nearly as cool as the games that I made years earlier with the help of RPG Maker 2000. At least from a personal standpoint, I believe that creating tools that will excite children into learning programming concepts and creating small games will eventually turn them into excellent programmers at a later age.
2. Previous Work

Daniel Wendell created the Spaceland editor that is currently in use today [9]. Daniel created functionality for several things. For example, he created a mechanism to change a patch’s color [9]. He also added functionality for raising or lowering an individual patch [9]. However, the most useful tool he made was one that created a mound (or hill) from a selection of patches [9]. These tools persist today in the Spaceland editor.

Daniel also created the backend for supporting textures, meaning that the user could use a texture to paint a terrain instead of just using a solid color [9]. There is a full implementation of saving a terrain. A user can sculpt their terrain as they see fit and save it with their project file. There is support for saving up to 15 terrains, but this feature is turned off in the current version of StarLogo TNG [9]. Also, there is a backend for saving a level [9]. A “level” is defined as the terrain data as well as all the agent data but there is no mechanism for actually saving, or loading, an agent’s data. Finally, there is also backend support for changing a terrain’s structure, but Spaceland’s OpenGL frontend always renders a grid of 101x101 patches. Lastly, there is currently no way to add an agent to Spaceland via Spaceland [9]. The user must create a new agent using the StarLogoBlocks interface (blocks exist for creating new agents).

An early version of his terrain editor required the user to exit from the Spaceland view and go into a separate editor (which was two-dimensional) [9]. The user would then make changes and hit a button to apply them. However, he noted that students didn’t like this implementation so he changed it so that the tools could be accessed directly from the Spaceland window [9]. This implies that users could edit the terrain directly, even while the agents were running. The conclusion is that a good editor should allow the user to directly edit the three-dimensional world. This will allow them to know what the effects of a particular change will be without having to swap contexts.
3. **User Classes**

Before an updated terrain editor can be created, one should first think about the different groups of users that will ultimately use the system [8]. By reflecting about the different types of users, then one can think about some of the early design considerations that may come into play. For example, if one were building a voice-mail interface, they may make one set of design decisions. However, if they were building a voice-mail interface for the deaf, then different design decisions may need to be made. In this case, there are three main user classes: students, teachers and researchers.

### 3.1 **Students**

Students using StarLogo TNG are most likely around middle-school aged, meaning they are likely to be somewhere between the ages of nine and thirteen. They will typically have had exposure to computers before but have had no experience with computer programming. This user class is likely to use StarLogo TNG for using models, running simulations and creating their own, simple, games. This group is likely to use both StarLogoBlocks and Spaceland.

### 3.2 **Teachers**

Teachers will be much older than the students. They can be anywhere from the ages of twenty two to sixty five. However, they may or may not have programming experience. For example, a biology teacher is unlikely to have programming experience but a computer science teacher will. Teachers will serve as mentors, supervising children in their journey to creating games. But teachers will also need to create models in order to teach the children something new. Teachers are very likely to use both StarLogoBlocks and Spaceland.
3.3 **Researchers**

Lastly, we have researchers as the third user class. Researchers are going to typically be in the age range of twenty to sixty five, but they are very likely to have a great deal of programming experience. This group is likely to have used other modeling software before. They will typically look to StarLogo TNG as a tool for creating simulations in order to view the steady-state behavior. This group would be more interested in StarLogoBlocks and not Spaceland. The reason is because this group wants to create simulations so the functionality of StarLogoBlocks is important (i.e. they need to be able to actually create the simulation) and the graphical result is not that important, so long as they can make a justifiable conclusion (i.e. by looking at some quantitative data in a plotted table or graph rather than the qualitative data that is the state of the terrain).
4. Feature Proposal

There are a plethora of features that can be added to Spaceland in order to make terrain editing more powerful. There are a few considerations that must be kept in mind while making a new user interface.

Daniel’s thesis taught us that the terrain editor should exist within Spaceland directly [9]. This means that the interface should be a direct mapped one. A direct mapped interface is one where you interact with the system directly [8]. For example, CAD programs are examples of direct mapped interfaces.

A consequence of a direct mapped interface is that patches and agents will be directly modified by the mouse. Spaceland is rendered in three dimensions. However, the mouse pointer only exists in two dimensions. Great care will have to be taken when designing the interactions between the mouse and/or keyboard and Spaceland.

4.1 Creation/Placement of Agents

As it stands, there is no way to create new agents via Spaceland controls. Agents have to be created programmatically via StarLogo Blocks. Agents can be moved around, but the current interface is a window popup (i.e. a form). The user may be restricted and only allowed to create agents while no Virtual Machine (VM) code is running. Right now, a user can edit the terrain while agent code is running. It is likely that agent code will be paused when the user edits the terrain. We may do this to avoid consistency problems. For example, agents have an id associated with them. If there is agent code which can hatch a new agent and the user can create agents via a toolbar in Spaceland, then there may be a problem where two agents accidentally share the same id. This is not a desirable effect so the easy solution will be to prevent the agents from running code while the user is editing the terrain.
4.2 Editing Dimensions of Spaceland

It is desirable to modify the dimensions of Spaceland. Currently, the terrain is limited to a 101 by 101 grid of patches. The backend currently exists to change the size of Spaceland, but the OpenGL frontend only renders a 101 by 101 grid. It is reasonable to allow a user to change the dimensions of a terrain when they create a new one (a “level”), but not allow them to resize the terrain once it is created. Task analysis is needed in order to determine if users typically would try to change the terrain’s size once it has been instantiated.

4.3 Undo / Command History

As of today, there is no way to undo any changes that are made to the terrain. If the user were to make a mistake, they would have to fix it manually. This can be quite tedious and frustrating for people who are experimenting with the editor and may not know what would happen if they used a particular tool. It would be useful to be able to undo actions. We could also go one step further and implement the ability to view a history of all the actions that a user made and undo any particular action in the command history.

4.4 Reset

Currently, it is possible to flatten a selection of terrain (reducing it to its “default” state of being solid green with a height of zero). However, there is no mechanism to reset the entire terrain to its default state. This feature would be useful for when a user just wants to start everything over and have a blank slate. One design consideration that needs to be addressed is the exact nature of the default state. For example, the default state can mean a flat terrain initialized to the same color (green). However, agents can modify the terrain during execution (changing a patch’s height, color, etc.). It may be better to somehow
encapsulate this type of terrain change and allow the user to reset the terrain to its state before the last VM execution.

### 4.5 Copy / Paste Terrain

Let's assume that the user spent ten minutes or so making a cool layout in their terrain. It would be nice if the user could highlight a selection of patches and then paste them somewhere else. One design consideration when coming up with a UI for this feature would be to figure out where the pasted terrain should go to. For example, when I copy and paste text, the destination of the pasted text is easily determined by the location of my cursor. However, since Spaceland is three dimensional and there is no notion of a “current location” like a blinking text cursor, the destination of the pasted features is not immediately clear. This is a design issue that will have to be worked out when doing task analysis. Also, what characteristics should be copied? For example, we can copy a patch’s height, color and a few other things. Which of these should be pasted? The user could have no control over what details get copied (i.e. copy everything); they could have complete control or they could have some degree of freedom in-between.

### 4.6 Texture Terrain

Currently, there is a backend to support texturing the terrain. However, there is no interface to the backend. This support should be added. However, some design considerations need to be made. For example, should the user be allowed to place a giant picture over all of Spaceland or should they only be allowed to tile one patch at a time? Can we allow the user to find a happy medium somewhere in the middle (for example, being able to tile up to a 5x5 grid of patches)? And if we do support multiple tiling sizes, then how should we present this interface to the user so that they know exactly what to expect then they try to tile Spaceland?
4.7 Graffiti Patch Walls

On a related note, when one raises a patch, a set of walls get drawn. Currently, there is no way to change the color or texture these walls. The color of the walls is determined by the color of the top of the patch. It would be nice if the user could directly manipulate the color of the walls independently of the color of the patch top. Likewise, the user should be able to texture the walls independently of the patch top.

4.8 Custom Patches

Right now the user can only raise and lower individual patches. The user may wish to create a shape out of the terrain (i.e. star, heart, etc.) and support for those particular shapes could be made. However, it would be much more useful if the user could create their own shapes and use them to quickly create patterns in the terrain. For example, the user could define a heart shape, which would raise or lower the proper patches to form a heart and then use that shape to make hearts in the terrain. This feature may or may not be useful so task analysis is needed in order to make that decision.

4.9 Load Level

Currently, there is a mechanism for saving the terrain. There is support to save multiple terrains. However, the user has no way to create another terrain and switch freely between the two. Also, there is no way to save and load agent data. For example, when a user loads a project file, they must run “setup code” in order to create all of the necessary agents and place them accordingly. This is very cumbersome and it would be nice to eliminate the need for this setup code. This feature would be useful for users that are creating games. Many adventure games have multiple levels with different sets of characters (bad guys) in each level. This feature would allow users to create multi-level games more easily.
4.10 Edit Skybox

Currently, the skybox is static. It can be turned on and off, but the user cannot change the skybox. Ideally, the user should be able to import any image (of a given size) and be able to use that as the skybox. However, it may be easier from a programmatic point of view to only restrict the user to changing the skybox among a set of given resources. The reason for this is because the current version of StarLogo defines all of its resources (i.e. agents) internally. Adding support for custom skyboxes creates problems if users attempt to share projects with each other. They would also need a way to share resources with each other as well.

4.11 Other Features

It is plausible that, after task analysis, users may demonstrate the need for a feature not listed here. It is possible that other features may be implemented in order to meet the needs of the user. It is also possible that features currently listed may be determined as useless and thus dropped from the feature list. All of these questions will be answered when a thorough task analysis is performed with real users of StarLogo TNG.
5. Measuring a Good Interface

It isn’t enough to simply build this user interface and proclaim success just because all of the features were implemented. The goal is to build a usable interface. Usability is defined as how well users can actually use a system’s functions [8]. Luckily, usability can be measured quantitatively. Below are the five measures of usability.

5.1 Learnability

Learnability is defined as how easy (or not) the user interface is to learn [8]. Typically, learnability is measured by observing users whom have never used the interface before. In a sense, learnability is just like the learning curve associated with the interface. Learnability is important for interfaces that are designed to be used by many different types of people. For example, the interface to an airplane’s flight controls need not be learnable because pilots receive extensive training on how to use the flight controls. But a more common interface should be learnable to accommodate many more users.

5.2 Memorability

Memorability is the measure of how well a user can recall an interface once they learn it [8]. This is usually measured by asking a user to perform a task, then return for a second session (possibly a day or so later) and then ask the user to repeat that task. If the interface is memorable, then they will be able to recall how to use the interface.

5.3 Efficiency

Efficiency is the measure of how fast a user can interact with the interface once they learn how to use it [8]. Efficiency can be measured by asking a user to repeat the same task
repeatedly and measuring how long they take. Efficiency is very important for interfaces that users will deal with on a persistent basis. For example, air traffic controllers need to be efficient at using their software because they are directing planes all day.

5.4 Errors

Actually, when we say "errors" we are really referring to how few errors are and, when they do occur, how easy it is to recover from them [8]. One can simply measure errors by observing all types of users interacting with the interface and observing how many mistakes they occur and how they attempt to resolve those errors.

5.5 Satisfaction

Last, but not least, an interface should provide satisfaction. Satisfaction is the measure of how enjoyable an interface is to use [8]. An interface like Gmail should have a large amount of satisfaction because it's a novelty interface. However, the air traffic controller interface need not provide much satisfaction because it's the user's job to interact with the interface. Satisfaction is somewhat subjective but a measure of it can be quantified by surveying users about their experience once they have completed a usability study.

5.6 What is Important for Us?

In an ideal world, all interfaces should have all of the above characteristics. However, the world is not ideal and some traits are more important than others. For instance, learnability is probably the most important characteristic because StarLogo TNG targets children whom have never programmed before. It is important that they can learn the interface quickly in order to enjoy it. This leads us to conclude that satisfaction is also of importance. If users do not like the system, then they simply won't use it. Memorability
is important, but not as important as learnability or satisfaction. The user should be able to recall how to use the tools, but if they cannot, there should be sufficient help (i.e. tooltip, example projects) to remind the user as to how to use the interface. Also, the interface is directly mapped so if the user forgets what a tool does, it is relatively safe to just try to use it and re-learn. Whatever they do should be easy to undo. So for that reason, error prevention should be important too, but not of utmost importance. Some user models will use blocks that edit the terrain (change colors, raise patches, etc.) so the user is likely to be adept at recreating features that they made once before. Lastly, efficiency is the least important trait of this interface. The students are encouraged to take their time, explore and play with the system.
6. A Tour of Spaceland

Before we explore the changes made to Spaceland, it is a good idea to first examine what Spaceland looked like, and behaved like, when I first started with the StarLogo TNG group.

Figure 2 -- Default Spaceland View
The above figure illustrates what a user would see (minus the StarLogo Blocks window) upon launching StarLogo TNG. Two turtles are initially positioned in the center of the screen and Spaceland is initialized to a flat, green surface. The buttons at the top of the screen control the camera. The square area in the lower left hand corner of the screen is the mini-map. The view inside of the mini-map is a two-dimensional representation of Spaceland and all of the agents inside of it. The Edit Terrain button is pretty self-explanatory. The arrows in the lower right hand corner are used to move the camera in the four cardinal directions. The plus and minus buttons zoom the camera. The status bar below the Edit Terrain button shows the frames per second and the rate at which the VM (Virtual Machine) is running. The slow and pause button advance simulation one frame and stop simulation respectively. The slider next to those buttons controls how fast the VM runs. Lastly, the space below the slider is reserved for various blocks created via StarLogo Blocks (for example, setup and runtime blocks).

There are four views, or cameras, in Spaceland. First, there’s the Aerial camera. With this camera, you can freely move around Spaceland and position the camera wherever you like. When you place the camera, it doesn’t move until you move it again. The Agent Eye and Agent View cameras are similar. Both of these cameras follow a particular agent as they move about Spaceland. The user has no control over this camera. The only difference between these two cameras is that Agent Eye is a first person view (literally through the agent’s eyes) and Agent View is an over the shoulder camera view. Lastly, the Overhead view, which can be seen in the mini-view, is a two dimensional representation of Spaceland. The camera is literally overhead and views everything from above. The user has no control over this camera either. Pressing the Aerial, Agent Eye or Agent View buttons causes the view in Spaceland to change (the mini-view stays the same). Pressing the Overhead button is a bit of a misnomer as the views in the Spaceland and the mini-view simply switch.

The image below (figure 3), shows the red cone that appears when the user clicks on Spaceland (either with the left or mouse button).
Unfortunately, the screenshot above doesn't show the mouse cursor, but it is positioned under the red cone. This cone appears when you click either mouse button and is used for maintaining orientation and position when panning, rotating and zooming. Panning is as simple as holding the left mouse button and dragging the mouse around. The invariant that is preserved is that the place you initially clicked (as denoted by the cone) remains
under your mouse cursor. As a result, when the mouse is dragged to the left, the view actually shifts to the right. In order to rotate, we click with the right mouse button and similarly drag the cursor around. The invariant in this situation is that the camera rotates about the point that you initially clicked. Lastly, we can simply scroll the mouse wheel up and down to zoom in and out respectively. The invariant preserved in this case is that the center of the zoom is set to where the mouse cursor is positioned.

Figure 4 -- Edit Terrain Menu
The above figure shows a forever block icon. This icon is what would appear if the user inserted a forever block into the StarLogo Blocks workspace (see figure 1). Clicking the forever block icon here will allow the user to run the body of the actual forever block, located in StarLogo Blocks. The above figure also shows the current terrain editing menu, which appears when the user presses the Edit Terrain button. Currently, the feature set for terrain editing is quite sparse. The mouse’s function changes when the user presses the Edit Terrain button. Left clicking no longer pans the camera. Now, it allows the user to highlight a square region of terrain. In Figure 4 above, this is denoted by the purple rectangle. After the user highlights a rectangular region of Spaceland, they can level the terrain, change its color, raise the area, lower the area, create a mound or create a pit. Leveling the terrain is pretty self explanatory; the user can simply set the highlighted area of patches to have zero height. Clicking the Color Region button causes the below menu to pop up (see figure 5 below).

![Choose a Color](image)

**Figure 5 -- Color Region Menu**

The top bar is a list of hues and the bottom bar is a list of shades. Changing the shade will have the effect of changing the hues. For example, if the user clicked on a darker shade, then all of the hues will darken as well.

Pressing either of the two buttons below the Color Region icon (the two in the bottom right) will raise the patch and lower the patch respectively. This simply uniformly changes the height of the entire highlighted rectangular region. In Figure 6 below, you can see a raised patch (colored pink) and a lowered patch (colored brown). They’re located on the left side of the image.
Also located in Figure 6 are a mound and a pit. They could have equivalently been called hills and valleys. The mound is the blue area and the pit is green (the same color as the rest of Spaceland). A better view of the mound and the pit can be seen in the figure below.
If one were to look back at Figure 4, the mound and pit buttons are the ones below the Level Region button (the ones on the lower left). The user can use the mouse to perform any of the four aforementioned functions. Raising and lowering a region can be performed by using the mouse scroll wheel when a region is highlighted. Likewise, a mound or a pit can be created by holding the SHIFT key and scrolling the mouse wheel.
The figure below illustrates what would happen if the user were to click on an agent. The Agent Monitor window pops up. From this window, the user can change any of the agent's variables (i.e. the agent's x and y coordinates). The user cannot move an agent with the mouse (i.e. drag and drop an agent somewhere). Clicking on any of the graph icons to the left of a label will pop up a nifty graph of that variable over time. For example, a user can graph an agent's x coordinate over time. The eye and the silhouette button simply change the camera to the Agent Eye and the Agent View cameras.

Figure 8 -- Agent Monitor Window
With a little bit of time and patience, one can create very pretty scenes. However, with the current toolset, this is a more of a pain than a pleasure. The figure below is an actual terrain created for use in one of the current curriculums. The house, trees and the brown bear in the middle of the screen are all agents.

Figure 9 -- Example Terrain
7. Lawrence Workshop

On May 16th, 2008, I had the pleasure of attending my first StarLogo TNG workshop. It was the last in a series of workshops being held at Lawrence High School. There were two groups: sophomores and seniors. Both groups were taking some form of Biology (the seniors were taking Ecology). The goals of the workshops were to teach students something related to their current curriculum and to give them some exposure to programming with the blocks language.

The purpose of this particular week’s workshop was to illustrate producer-consumer relationships by programming a bunny. The goal was to program bunnies such that they ate carrots, to survive, while they avoided wolves, also to survive. This particular model was very good at illustrating concepts such as population growth (unconstrained and constrained). This is beneficial because the rabbit population can be modeled with a simple linear differential equation. However, most high school students have had zero exposure to differential equations. Rather than trying to teach them the complex math, they were allowed to just run the simulation and make both qualitative and quantitative observations. Since StarLogo TNG has support for graphing variables, it was quite easy for students to look at a graph of the rabbits’ total population and draw conclusions.

Speaking of observations, I made a few of them myself. I did not lead the workshop so I had the opportunity to watch the students and observe how they use StarLogo. They didn’t need to edit the terrain at all, but they did have to interact with Spaceland (i.e. to position the camera so that they could watch the simulation). The first thing I noticed was that students were able to move the camera away from Spaceland and they had no way of getting it back. The first plausible conclusion drawn was that a reset button was necessary to revert the camera to a default state. Another thing I observed was that the aerial camera was sometimes too restrictive. At times it would fail to move when I believed that it should have moved. This led me to believe that the constraints on the camera were too rigid and it was getting in the way of normal operation. The conclusion drawn was that the camera’s movement needed to be examined in further detail. Lastly, I
observed that students didn’t seem to be too fond of the buttons overlaying the Spaceland OpenGL view. It appeared that they were sometimes distracting. The students apparently valued having as much screen real estate as possible to move and rotate the camera. They also seemed to be very confused by the “Overhead” button, even though they had been working with TNG for several weeks. Many students clicked on the Overhead button and then became unsure of what to do next. Because students value screen space, I needed to be very careful to not pollute the screen with buttons for terrain editing. If it was at all possible, I should remove buttons and provide a different interface to their functionality, preferably one that doesn’t overlap the view of Spaceland.

7.1 Changes to Spaceland in Response to Lawrence Workshop

After attending the Lawrence workshop, I began work on fixing the usability issues that were observed. The image below shows the final effect of all of my changes.

Figure 10 -- Lawrence Changes
The two most notable changes in the above image are the inclusion of a "Reset Camera" button as well as a "Swap Views" button. The "Overhead" button is also missing. The function of the overhead button was very confusing to students. The view in the bottom left corner represents an overhead view of the two turtles in the above figure. When a user pressed the overhead button, that view would be switched with the regular Spaceland window. Pressing the button again would switch the views back. However, the label "Overhead" on the button didn't exactly convey this operation very concisely. The fix was to simply change the label on the overhead button to say "Swap Views" and to move it to be below the mini-view. The button was moved in order to place it within the locus of attention of the user. The locus of attention refers to the area on the screen that the user is giving the most attention. In this case, pressing the swap views button causes something to happen within the vicinity of the mini-view; therefore the swap views button is justified for being placed there. Figure 11 below shows the effect of pressing the swap views button when we are in the state represented by Figure 10, above.

Figure 11 -- Swap Views Button
The other large change made was the inclusion of a Reset Camera button. This button makes a best effort attempt to restore the camera position to some known default state. When a user saves a project, one of the things that are saved is the camera position. When a user loads a project, the camera view saved in the project file becomes the default camera view. A problem can arise if the user somehow manages to navigate away from Spaceland and then saves their project. This did happen during the workshop, but it turned out that the camera was able to move away from Spaceland due to a bug. The bug was fixed and we concluded that the user could no longer make the camera move away from Spaceland so the potential problem should never happen. As a result, the Reset Camera button is a convenience function rather than a means to fix a bug.

Lastly, an effort was made to improve the camera functionality, but I was unable to qualitatively improve the camera functionality. The camera is currently over-constrained (i.e. it may not move when one believes that it should), but this approach has the benefit of not allowing the camera to move away from Spaceland.

### 7.2 Reception

I was unable to observe students using these new changes until another workshop in Lawrence. This time, it was a summer workshop aimed towards gifted middle school students. These students were considered beginners to StarLogo TNG. During the earlier Lawrence workshop, I noticed that many students would press the Overhead button and then become stuck; they wouldn’t know what to do next. With the new change, users were less likely to become stuck. I believe this has more to do with the movement of the button rather than renaming it. Typically, students would be fiddling with the cameras when they would experiment by pressing the Overhead button. As mentioned before, the other three buttons act like a group of radio buttons; any time one is pressed, it becomes the active selection and changes the camera view. The Overhead button acts like a toggle button; the main view either contains Spaceland, or the contents that would normally be in the mini-view. Students typically became stuck when they pressed the Overhead button and then tried to change it back by pressing one of the other camera buttons. The
correct way to change the camera back was to press the Overhead button again. The real problem was that two separate groups of buttons were grouped together, even though they had different functionality. Students are “chunking” the four buttons together and perceiving them as having similar function [8]. By moving the button away from the group, students were less likely to press it. Also, the more informative label helps to identify its function so if students were to press the button, they would have a much clearer expectation of what would happen. The overall feeling to the change was very positive.

Few students pressed the Reset Camera button. It was regarded as a more advanced feature and one that probably shouldn’t be used too often anyway. It was intended as a convenience function to quickly move the camera. Most, if not all, users would prominently manipulate the camera manually with their mouse.
8. **Load Level**

StarLogo TNG used to only allow the user to create and edit a single level. This caused problems because agents have the ability to modify the terrain; they can change the heights and the color of the individual patches. Users would create a level and then run a simulation, which could potentially destroy their nice level. Furthermore, users only had the ability to create one level so if they wanted to make a game, then they were limited to single-level games. In order to remedy these problems, I created an interface to allow users to make multiple levels.

In order for this feature to be implemented properly, users would first need to be able to create a new, blank level. They would also need the ability to delete levels. Furthermore, they would require the means to change levels, as Spaceland only displays the current level. I also addressed the problem of agents modifying the terrain and created a way to restore a level to a previous state (i.e. before the simulation was run and agents modified the terrain). This is analogous to the “Reset” feature that was documented above in section 4.4. All of these features needed to be accessed via controls in the Spaceland editor but some of them also needed an interface via the blocks language.

### 8.1 Design Iteration

The main problem I faced when implementing this feature was to decide where to place all of the UI components that I was going to add. I didn’t want to place them on top of the Spaceland view because that would clutter the screen. Design iteration was performed via a series of pencil and paper sketches.

The first potential design would be to place a single button somewhere on the Spaceland window. This would have been a minimal addition and thus wouldn’t clutter the view too badly. When this button was pressed, a popup box would appear with the necessary controls to edit levels. This interface would appear similar to the popup box to color a
terrain (see Figure 5) or the popup that displays the Agent monitor (see Figure 8). This idea was ultimately rejected because StarLogo TNG already takes up a lot of space on the screen and already contains two windows (one for StarLogoBlocks and one for Spaceland). A third window could only exist for very temporary tasks. For example, picking a color to paint the terrain with is a temporary task. The act of creating, editing, navigating and deleting levels was deemed to not be temporary enough to allow for a popup box. The user is likely to frequently want to perform any of the operations associated with levels. This design would have lent itself to errors as users would be distracted by the jarring popup [8].

Another design proposal would be to mimic the drawer function in StarLogoBlocks. When a user clicks on a drawer in StarLogoBlocks, it pops out to show a list of all the blocks that are contained in that drawer.
In the above figure, the Logic drawer has been selected. Blocks related to logic functions are present in this drawer. For example, an “if” block is displayed at the top of the drawer. There are also the Boolean “and”, “or” and “not” functions. The proposed level design would use this concept of drawers and apply it to the Spaceland window. However, since Spaceland is docked to the right of the StarLogoBlocks window, such a drawer would have to open to the left, to not increase screen real estate used. One great benefit to this approach is that it’s scaleable. We could easily add new drawers as we implemented new functionality. For example, the agent monitor (the screen that pops up when you click on an agent) could go away and we could have a notion of an agent drawer. There are a few obvious problems with this approach. First, the drawers would slide to the left and overlap the StarLogoBlocks canvas. This is undesirable and can be confusing as students would likely expect a drawer to open to the right. While it wouldn’t increase the amount of horizontal screen space used, it would increase the overall clutter of the screen. A second problem arose when we considered what would be placed in the drawers. In StarLogoBlocks, it makes sense to have the blocks themselves be in the drawer but how could we graphically represent a list of terrains? Furthermore, if we added other “drawers”, how could we graphically represent whatever is to be contained within those drawers? We were all hard pressed to come up with a good way of graphically representing levels in a drawer so this idea was abandoned as well.

A third proposal involved using the space already occupied by the Runtime Workspace. The Runtime Workspace refers to the area below the Spaceland window where forever and setup blocks live. In Figure 9, the area with the slider, play/pause button and the two red blocks, labeled “forever” and “setup” depicts the Runtime Workspace. A project may have one or more Forever or Setup blocks as well as any number of graphed variables. For example, one can graph the number of living turtles over the course of time. All of these blocks would be located in the Runtime Workspace. A block with each level’s name would be placed in this area and users could navigate the levels simply by pressing blocks. A button would have to be made to allow users to create and delete levels. This seemed like the best idea because it re-used existing space rather than trying to take up more screen real estate. The UI challenge here would be to figure out how to create and
delete levels. The way a user causes a Forever or Setup block to be placed in the Runtime Workspace is to place a Forever or Setup block on the canvas. All runtime blocks appear in the Runtime Workspace after they have been placed on the canvas. The other problem is that there are other runtime blocks. For example, there is a Run Once block. Running a forever block causes its body to be executed forever, or until the user stops execution. A Run Once block simply executes its body once.

Figures 13 and 14 show the relationship between runtime blocks on the canvas and the Runtime Workspace. In Figure 13, there are two runtime blocks, a forever block and a run once block. Both runtime blocks contain the same blocks within their bodies. Pressing the green triangle on either block would cause all of the Turtles to move forward one step. The difference is that the forever block will cause the VM to run indefinitely, whereas the run once block only runs the VM for one step. The net effect is that when we
press the Run Once block, our turtles will only move forward once and then stop, whereas with the Forever block, they would continue to move forward until we told them to stop.

Figure 14 shows the blocks that are added to the Runtime Workspace. Since we have a Forever block and a Run Once block in the canvas, we have the same two blocks in the
Runtime Workspace. Pressing either of the blocks on the Runtime Workspace causes the pressed block to execute its body; if the Forever block is pressed, then the Turtles will move forward indefinitely.

The biggest problem with this seemingly perfect idea is that the abstraction of the Runtime Workspace breaks down. There is a one-to-one correspondence between the runtime blocks on the canvas and the runtime blocks on the Runtime Workspace. Also, the Runtime Workspace is for runtime blocks and a block representing the various levels is not a runtime block. Since we had no desire to break the abstraction, we had to abandon this idea but we used it as inspiration for the final solution.

8.2 Final Solution – Spaceland Side

Figure 14, above, gives away the final solution. Instead of re-using space in the Runtime Workspace, we noticed that the Runtime Workspace doesn’t need to be visible when a user wishes to create, edit, rename or delete levels. For that reason, the final answer was to make a “Levels” UI component and have it exist in a tabbed pane with the Runtime Workspace.

Figure 15 shows the Levels tab. When a user presses the Levels tab, the entire Runtime Workspace, the speed slider, the play/pause button and the area for runtime buttons,
disappears. The graphic design of the levels tab is quite simple. The majority of the screen is taken up by a list of the all the levels. The bottom comprises of four buttons: new level, save level, restore level and delete level. New Level and Delete Level are complimentary actions but since delete is a dangerous operation, it is placed far away from creating a new level. Also, since it is at the far edge of the screen, users are unlikely to press it by accident.

Pressing the New Level button causes a new, default level to be created with the name "Level X" where X is a number between 0 and 15 such that no two levels share the same name. The reason X can only be between 0 and 15 is because we only allow the user to create up to 16 levels. This is an artificial limitation and could be trivially increased at a later time if necessary. The upper bound of 16 was chosen simply because no one knows how many levels the user requires. The "default" terrain is a 101 x 101 grid of green patches with zero height. We could have allowed the user to specify a default name and a default height and width via a popup but this was pushed off until a later date. The size of the terrain is un-editable, though backend support for this exists. However, there is full support for renaming a terrain. A "default" name is chosen simply to avoid using a messy popup box whenever the user presses the New Level button. A default name was given for efficiency and if a user chooses, they have full power to rename a level.

The user can rename a level by double clicking on any of the levels in the list. This causes a highlight box to be drawn around the row and a blinking cursor appears at the end of the level’s name. The user can then type in, almost, any level name and press enter, press an arrow key or click outside of the box to make the change take effect. This behaves exactly like table cell editing in other applications, such as Microsoft Excel or Apple’s iTunes. As a result, users are familiar with the interface and very likely to learn it quickly. No two levels can have the same name, so if the user types in an existing level name, then it is rejected and the old name is restored. The figure below shows the effect of double clicking on a level's name. Unfortunately, screen capture software does not pick up the mouse cursor, but there is a blinking text cursor at the end of the level's name.
A user changes the level by single clicking on a level’s name. This is very intuitive and extremely learnable. The user can also change level by using the arrow keys on the keyboard to move up and down the list. The invariant that is preserved is that the highlighted level in the list is the one that is displayed in the Spaceland window. For example, in Figure 15, Level 0 is highlighted; therefore the user is currently looking at Level 0 in the Spaceland window.
Deleting a level is as simple as pressing the Delete Level button. Unfortunately, there is a tradeoff between error prevention and user satisfaction. Currently, pressing the Delete Level button deletes the current terrain (i.e. the one that is highlighted in the list). There is no confirmation dialog and there is no way to undo the action. Deleting the highlighted terrain is a sensible one. For example, if a user wishes to delete a file on their computer, they will highlight the file and then press the delete button. The same thought process works for deleting a level so it is learnable. Confirmation dialogs do tend to reduce errors at first because they prevent accidental presses of the delete button. Unfortunately, confirmation dialogs interrupt a user’s thought process and work flow. As a result, they will internalize the confirmation dialog as part of their deleting process and most power users will press the Ok button without even reading the dialog. The confirmation dialog is useless at this point and does nothing to prevent an error [8]. A much more powerful option would be to allow a user to undo deleting a level. It is likely to be implemented as part of undoing and redoing terrain edits in general.

The other two buttons, Save Level and Restore Level, attempt to address the problem of agents messing up the terrain during simulation by changing a patch’s height and/or color. Each terrain has its own backup copy located in memory. Whenever the user presses the Save Terrain button, the current terrain configuration is saved to that backup slot, overwriting whatever configuration located there. The Restore Level button copies the backup configuration for the current level into the active configuration. After pressing the Restore Level button, both the active and backup configurations for a given level are equal. The idea behind these two buttons is that a user can create a level and press the Save Level button just before executing the simulation. When the user runs the simulation, agents may or may not modify the terrain. If the user’s simulation allows agents the edit the terrain, the user has the freedom of pressing the Restore Level when they are done running the simulation and have observed the results. This will restore the terrain to its state just before the simulation was run (i.e. when the user pressed the Save Terrain button). Previously, if a user ran a simulation which modified the terrain, the only way to restore a terrain would be to close and open the project again, but only if the user
did not save the project before closing it. If the user saved the project after running the simulation, then the changes to the terrain would be saved and it would be impossible to restore the terrain to its state before the simulation was run. These two buttons allow for a very coarsely grained undo function. This only works for terrain edits, however.

In order to implement these features, some changes needed to be made to the existing framework. Previously, each terrain had a number associated with it, which was an index into an array of terrains. The user shouldn’t have to know that the terrains are stored in an array, nor should they be concerned with indices. From a usability perspective, it was much more learnable and memorable for users to be able to name terrains. Therefore, existing code had to be refactored to allow other code modules to access a terrain by name or index value, as there was old code that referred to a terrain by its index. Also, the notion of a backup terrain, one for each slot, was new. Code needed to be added to allow other modules to save a terrain to a backup slot and to restore a terrain with its backup copy. Also, the code that saved terrains to the project file did have the ability to save multiple terrains, but it saved them by index. A name field had to be added. Also, the save file had no notion of a backup copy so the save format was appended to not only save all of the terrains, but to also save the backup copies. This neatly solves the problem described above: if a user saves a terrain to memory before running a simulation, then runs a simulation that modifies the terrain, and then saves the terrain, then closes the project and finally opens the project, they can still restore the terrain to its state before they ran the simulation by pressing the Restore Level button. Previously, the user would have to re-create the terrain.

8.3 Final Solution – StarLogoBlocks Side

There are six operations that a user can perform related to levels: creating a new level, changing from one level to another, renaming a level, saving a level to its backup copy, restoring a level from its backup copy and deleting a level. After some task analysis, it was deemed that users definitely need to change levels from the blocks language. In a game, a character will run through one level and when they finish, they should have a
means to go to another level. This involves changing the level. There was some debate, but it was also decided that users should probably have the ability to save and restore terrain backup copies. The belief was that students likely wouldn’t need to use the Save Level and Restore Level buttons in the editor if there were blocks that saved and restored a level for them whenever they ran a simulation. In this sense, the creator of the simulation (i.e. the teacher) could use the blocks to abstract away the feature of saving and restoring a level from the student. The belief was that saving and restoring a level was an advanced operation. Creating a new level, renaming a level and deleting a level are operations that only needed to exist in the editor, as they are geared towards the creation of a game/simulation and probably don’t need to occur during runtime. The result is shown in the figure below.

Figure 17 -- Levels Blocks
In Figure 17, the drawer of note is the Terrain drawer, which is five drawers from the top. Near the top of the drawer is the Change Level block, followed by Save Level and Reload Level. The top block, “current level”, is used to report the name of the current level. This is explained in a later section. Change Level requires a string argument, hence the square cutoff at the right side of the block. The string is the name of the level. If the name doesn’t exist, then no operation is performed. Clearly, this isn’t the best choice of error handling, but TNG currently only supports compile errors, not runtime errors. For example, if a user tries to execute the Change Level block without attaching a string, then the VM will give a compile error and nothing is run. However, there is no support for handling runtime errors or exceptions. In the case of changing a level, the absence or an error isn’t catastrophic because the user will clearly be able to see that their level didn’t change and they can debug it accordingly.

The Save Level and Reload Level blocks take in no arguments. Much like the corresponding buttons in the editor, they operate on the current level. The alternative was to allow these two blocks to take in a string argument. The argument was that if there was a string argument, then there would be reduced errors. For example, if I have a Save Level block without the argument and I change level, then when the Save Level block is executed, it will save a different level. This could potentially cause errors when the user attempts to restore a level and the result is not what they expect, because the save point was never created. However, this approach wouldn’t be very learnable and confusing. If a user saw a Save Level block with a level name attached to it, they’re likely to interpret that as, “Save the current level to the slot of the level attached to this block.” It was undesirable to have ambiguity surrounding any block’s function because that decreases learnability and memorability, two of our most important measures of a good user interface. Furthermore, the no-argument block is consistent with the Save Level and Reload Level buttons in the editor; they both operate on the current level.
8.4 Reception – Teacher Workshop

These features were first demoed for other members of the lab. Reception was quite positive. People were greatly excited to be able to create multi-level games. Unfortunately, at the time of the demo, there was no notion of associating an agent with a level. For example, if I created two turtles on Level 0 and then switched to Level 1, those two turtles would still be there. If one wanted to create a game, they would have to manually kill and create the proper agents whenever the player reached the next level. The conclusion was that agents would have to be associated with a level and agents not living on the current level shouldn’t execute any code (i.e. they shouldn’t move). In user testing, no one had difficulties performing any of the six operations associated with levels.

At the end of August 2008, there was a teacher workshop, Imagination Toolbox. The purpose of this workshop was to teach StarLogo to classroom teachers so that they could use it as a part of their curriculum. Previously, the workshops were run by members of the lab. They created the models and taught the lesson to the students. Teachers would give up teaching time to allow us to run a workshop for a class period, which is usually somewhere around 45 minutes for a group. Naturally, the limiting factor in running workshops is the amount of people in the lab who can run these workshops. All of the teachers at this workshop were beginners. Most of the teachers were able to experiment with the Levels tab and they were able to create and delete levels with no difficulties; that is to say no major usability issues were uncovered. Unfortunately, the Save and Reload Level functions were largely unused. This was to be expected as these two functions are advanced features. However, testing among lab members seems sufficient to conclude that the current interface is satisfactory.

8.5 Responses to Unforeseen Issues

After the teacher workshop, some work was done to address the issues that were discovered. As stated previously, agents previously had no notion of an existing only on
one level. As a result, when the user switched between levels, all agents would be rendered. The original design of StarLogo TNG didn’t account for the possibility of multiple levels. Various modules, including the underlying VM code, had to be refactored to allow agents to only exist on one level at a time. Agents were updated to include a level number. This is the index of the level that they are associated with. When an agent is created, it is associated with the current level. The rendering code was updated to only draw the agents that live on the current level. This step aids in the creation of multi-level games.

Another issue that arose when multiple levels were implemented had to do with StarLogoBlocks. Agents were fixed to only be rendered if they lived on the currently displayed level. However, when agent code is run, it runs for all agents, including the agents that live on different levels. The VM could have been modified to only run code for the agents that live on the current level. However, it was decided that model creators should have control over whether or not agents on different levels are updated. For a game, the creator may not want agents on different levels to run their code. However, if the creator could wish to run multiple simulations at once (i.e. on different levels), but subject each level to different initial conditions. In this scenario, the model creator would prefer to have all the agents run their code at once.

The response to this issue was to create three new blocks. The first block can be seen in Figure 17. It is the “Current Level” block, in the Terrain drawer. This block is a string block that returns the name of the currently displayed level. The other two blocks can be seen in Figure 18, below. The two blocks are “Set Level” and the “Level” block. There are located in the Traits drawer. The “Set Level” block allows an agent to move from one level to another and the “Level” block returns the string name that the particular agent lives on.
These blocks are meant to be used within an if block. For example, if the creator wanted to only allow agents on the current level to run code, then they could use an if block to test if the agent's level is equal to the current level. If so, then the body of the if block would contain the agent code. This mechanism allows the model creator to have the flexibility to decide which agents should run their code.
9. Saving and Loading Agents

One of the major hindrances to game creation is the inability to save agents to the project file and have their data restored upon loading the project. Terrain data persisted to the project file so when a project was loaded, the terrain would be restored, but there would be no agents. For years, creators of simulations would have to use a runtime block, called Setup, to execute code that created and placed all of the agents on a level. The Setup block is a runtime block, so it appears in the Runtime Workspace (and can be seen in Figure 9). It is similar to the Run Once block with the exception that it can run privileged instructions that normal runtime blocks cannot. The Setup block typically creates agents and places them somewhere in the terrain. Unfortunately, it is very tedious to set the x and y coordinates of each agent manually, so there is a shortcut block called “Scatter X” where X is the name of a breed (i.e. “Turtles”). The Scatter Turtles block would take all of the Turtles and place them randomly about the terrain. For simulations, it’s usually okay to allow agents to be placed randomly via setup code. However, games typically revolve around a series of scripted events where agents are placed in specific locations. In order to facilitate this, the agent data needed to be saved to the project file.

9.1 Implementation

The save file for a project consists of many different parts. For example, one part is a series of XML data representing all of the blocks on the canvas in StarLogoBlocks. Another part consists of the blocks in the Runtime Workspace. Another corresponds to all of the terrain data. A new part was added to support the agent data. The simplest format was to write a separate XML stream to the project file containing the necessary data. StarLogo has a hard limit of only allowing 4096 agents. The agent data lives in a contiguous block of memory. When an agent is created, it is assigned a “who” number. This number corresponds to one of the 4096 slots that an agent can reside in. It is important that agents retain their who numbers so they had to be saved to the XML file. Also, an agent needed to know its breed and shape. Breed and shape aren’t always the
same. For example, we can have a Bear breed, but make it look like a fish. All Bears have the shape of a bear, but this one particular Bear may have the shape of a fish. These are the only three things we require to be saved so they were placed as attributes within an AGENT tag in the XML file. Everything else is optional and treated as data. So, a sample save file may look like:

```xml
<AGENTS>
  <AGENT who="1" breed="Turtle" shape="animals/turtle">
    <X>5</X>
    <Y>42</Y>
    ... more data here
  </AGENT>
  <AGENT ... >
  ... 
  </AGENT>
</AGENTS>
```

The above XML code is compressed when it is saved to the project file because there could potentially be a lot of agents created (up to 4096). The three attributes, who number, breed and shape are all required to create an agent (at the VM level). Once we create an agent, we can simply parse the rest of the XML tree and set the agent’s attributes as necessary. For example, when we create an agent, it’s X and Y coordinates are set to 0. However, if we save this as data in the XML structure, we can set the agent’s X and Y coordinate (at the VM level) when we obtain this information. Other things that are saved include an agent’s size, heading and height above the terrain (i.e. for flying or swimming object).

One other important thing that is saved in the project file is agent variables. Agents come with a series of pre-defined attributes, many of which were mentioned above. However, the user has the ability to create a variable and assign it a value. This is to allow the user to extend their agents to contain more information. For example, agents are only uniquely identified by their who number. Agents don’t have a name associated with them. A user could create a Name variable which holds the string value of an agent’s name. Variables have two types of scope: Agent and Global. Agent variables are like member variables in
a Java class, there is one variable for each agent and each agent has its own value for that variable. Global variables are like static member variables in a Java class, there is one variable with one value and everyone can access it. There are also three types of variables supported in StarLogo TNG: numbers (double), strings and booleans. Agent variables are saved as part of the XML schema described above. When a project is loaded, the variable is initialized to its saved value in the project file. Consider that the user wishes to run a simulation for a very long time, say on the order of days but they do not have the time to sit around and watch the simulation run. They now have the ability to run a simulation, stop it when they wish, save the file and close everything. When they return to StarLogo TNG and load their project, all of the agents are restored to their last saved state, including any user variables. As a result, almost all setup code is eliminated. This feature was well received among the people who create the various projects for StarLogo TNG workshops.

### 9.2 Saving Agents in Memory

In section 8.2, a mechanism was described to save a backup copy of a level. This backup would be used to restore the level after running a simulation where the agents may destroy the terrain (for example, by changing the colors of the patches, or the heights of the patches). When the user pressed the “Restore Level” button, only the terrain would be restored. After implementing saving agents to the file, the team discovered that agents should also be saved and restored when the “Save Level” and “Restore Level” buttons are pressed. This normally would have been a straightforward implementation, but there were two complications.

Assume that an agent is created and their who number is number five and it is saved to memory via the “Save Level” button. When that agent dies, who number 5 is freed. Later during simulation, a new agent may be created and it may be assigned who number five. The problem arises when the user presses the “Restore Level” button. The dead agent with who number 5 should be restored and its who number should be five. However, another agent already exists with that who number. Therefore, a simple reference
counting system had to be implemented to ensure that when an agent dies, its who number is only freed up if there are no remaining references to that agent.

The other issue arose when dealing with agent variables. Agent variables have a name associated with them. Variables can be created, renamed and deleted. Some care had to be taken when dealing with these operations. For example, if a variable is created after a level is saved to memory, then nothing will be added to the save state. This is logical because the variable was created after the state was saved to memory. A variable may also be renamed after a level was saved to memory. In this scenario, a reasonable action was to update the save state so that it knows of the variable renaming. This implies that when the level is restored, the renamed variable will have its value restored as well.

Lastly, a variable may be deleted after a level is saved to memory. In this case, we chose to ignore the deleted variable when the user presses the “Restore Level” button.

After taking care of the aforementioned issues, the “Save Level” and “Restore Level” functions were updated to save and restore both the terrain features as well as the agent data. The saved agents in memory also had to be persisted to the project save file in order to be consistent with the operation of the “Save Level” and “Restore Level” buttons. Remember that the backup terrains were also saved to the project file. As a result, a user can load a project and still have their backup levels retained in memory, including the saved agents. This is a great feature to have, especially for game creation. Users can now restore a level and truly have the level reset to some known, user-defined, default state.
10. Creation and Placement of Agents

Previously, the user had no ability to create and place agents via Spaceland controls. As described in section 4.1, when a user wished to add an agent, they had to do so by dragging a block on the canvas, double clicking on it and then using the Agent Monitor to place the agent. The images below illustrate this user experience.

![Figure 19 -- Creating Two Agents (Step 1)](image)

After dragging the “Create Turtles” block onto the canvas and indicating that one would like to create two turtles, the user then must double click the block to execute it. The result is that two turtles are created in the center of the screen, but they are placed on top of each other.
The camera has been moved and rotated to illustrate that the two turtles are indeed on top of each other. The user then must click on a turtle to have its Agent Monitor pop up. From this screen, the user can finally set the agent’s x and y coordinates. This experience is not very intuitive and definitely not user friendly. There is a clear coupling between StarLogoBlocks and Spaceland; the user must issue a VM command in order to create an agent on their level (that they’re presumably in the process of building). More troubling, the user has no real control over where the agent is placed. Every time the Create Turtles block is run, it will create some number of turtles and place them in the center of Spaceland. This is not very efficient if the user wishes to create 10 turtles and place them throughout the level. Finally, the method of moving a turtle is quite inefficient. The user must type in x and y coordinates. However, they don’t really know too much about x and y coordinates (they aren’t visible anywhere on the screen). When a user is placing an
agent, they are more likely to use landmarks when describing where to place an agent. For example, they are likely to think “I need to put this turtle near the tree” and not “I need to put this turtle at (-42, 5).”

Figure 21 -- Creating Two Agents (Step 3)
10.1 *The Solution*

Since the best solution for solving the problem of creating multiple levels was to create a new tab in the Runtime Workspace, the same solution was applied here. A new tab, named “Agents”, was created near the bottom of the Spaceland window. This tab was designed to closely resemble the “Edit Breeds” window that had already existed in StarLogo TNG. The Edit Breeds window is how a user manages their breeds; they can create, rename or delete breeds from here. The figure below shows the Edit Breeds window.

![Edit Breeds Window](image)

*Figure 22 -- Edit Breeds Window*

The Edit Breeds button can be seen near the top of the picture (it’s highlighted as being pressed). At the top of the Edit Breeds window, there are the New and Delete buttons,
which allow the user to create a new breed and delete the selected breed respectively. The horizontal bar with the picture of the turtle, bear and fish represents all of the known breeds. The Turtles breed is selected, indicated by the non-transparent picture and text box, the highlight around the text box and the blinking icon in the text box. The window near the bottom represents all of the known shapes (models that can be rendered in Spaceland).

Figure 23 -- Agents Tab
The figure above illustrates the Agents tab. When the user creates new breeds (or renames them or deletes them) via the Edit Breeds window, their corresponding UI components in the Agents tab get modified accordingly. In the figure, no agent is selected so if the user were to click on Spaceland using the left mouse button, the normal camera controls would apply. The raised bevel around the UI components affords clicking (meaning that the UI components look “clickable”). If the user were to click on the Bear button, they would see the below figure.

Figure 24 -- Bear Highlighted
There are a few points of interest in the figure above. First, the Bear UI component is now depressed and the icon is displayed without any transparency. This is consistent with the Edit Breeds window in indicating that the Bear component is selected. The mouse icon is displayed over Spaceland (screen capturing software doesn’t capture the mouse icon). Since the mouse is displayed over Spaceland and a button is pressed, the status bar displays a helpful status message, “Click to add Bear at X: -6, Y: -2.” These two indicators allow the user to clearly understand that left clicking will now add a Bear at that position in Spaceland. Ideally, it would have been perfect to have Spaceland render a semi-transparent bear over the mouse cursor, but this feature proved to be too time consuming as much of the rendering code is strongly coupled together. The other two affordances allow the user to learn this feature very easily.

Once the user clicks on Spaceland, the bear is created and placed at that position. If the user then decides that they wish to move the bear, they don’t have to open up the Agent Monitor at all. They can simply click on the bear and drag it to the desired location on Spaceland. The user can drag any agent while the Agent tab is selected. Unfortunately, there is no easy way to discover this feature. The feature itself is quite intuitive and easy to learn and remember but discovering it for the first time is difficult. When the user moves their mouse over an agent, the cursor changes to a pointing finger (much like when one hovers over a link on a web page). This cursor indicates to the user that the hovered object is clickable (much like a link to a web page), but say nothing about whether or not the object can be dragged. The image below shows the cursor that affords grabbing.

![Figure 25 - Open and Close Hand Cursor](image)

When the user clicks, the open hand closes, as denoted by the image on the right. Unfortunately, the user can both click on the agent and drag it. Clicking on the agent opens up the Agent Monitor whereas dragging it would move the agent around. Therefore, one cursor cannot be used when hovering over an agent. One plausible
solution would be to use the status bar to display a message indicating that the hovered agent can be dragged. This area needs more exploring and user studying to obtain an optimal solution.

Since the mouse is used for moving the camera, editing the terrain and potentially during simulation (if the user programmed the mouse to do something), extra care had to be taken to make sure that the new operation of adding an agent didn’t interfere with other operations. A user can only add an agent if they are not running a simulation, the agent tab is open, they have selected an agent, and they are not editing the terrain. Unfortunately, the last condition is a bit of a hack because the Edit Terrain button shouldn’t be located on top of Spaceland. Given the new set of tabs at the bottom of the screen, the Edit Terrain features should really be moved to its own tab. If this were to happen, there would be less confusion over the conditions in which an agent is added. All of the other conditions are reasonable and somewhat intuitive. Currently, if the user presses the Edit Terrain button while in the Agents tab, the selected agent is automatically unselected, the status bar message goes away and the left mouse button is used to select a region of patches to edit. In this scenario, the user is unable to press any of the buttons in the Agents tab. When the user presses the Edit Terrain button again to exit editing mode, the previously selected agent is automatically selected again and the status message reappears in the status bar. These steps are taken to avoid errors and confusion over the operation of the mouse.

There were some concerns over how to delete an agent. The UI for creating an agent is nice because it allows the user to create many agents in one operation. All they have to do is select the breed that they wish to create and click on the proper areas on Spaceland. If they miss their target, they can simply drag the agent to the correct place. This design is very learnable, memorable, efficient and guards against errors. One potential solution was to create a UI component for deleting agents and place it in the Agents tab, much like the rest of the breeds. This would be nice if the user wished to delete many agents at once, but it would be inconsistent with the design of the tab. The Agents tab holds representations of breeds, not tools for deleting breeds. Clicking on anything but the
delete tool would add agents but the delete tool would delete them. This is inconsistent behavior. Therefore, the alternative was to add a delete button to the Agent Monitor. This allows users to delete individual agents. It is not very efficient if the user wishes to delete many agents at once, but it gives them control over which agents they would like to delete and thus is less error prone (i.e. the user is not likely to accidentally delete an agent because they need to click on the agent first to open up the Agent Monitor). The figure below shows the changes made. Note that the bear has been created at (-6,-2).

Figure 26 -- Deleting an Agent
11. Changing an Agent's Shape via Agent Monitor

One unforeseen feature was the need to change an agent's shape via Spaceland controls. In StarLogoBlocks, there is a block that will allow the user to change an agent's shape. However, the corresponding controls did not exist within Spaceland. After brief analysis, it was determined that since the user only needs to change the shape of one agent, then it was acceptable to place this tool within the Agent Monitor. Also, since the Agent Monitor already displayed an icon representing the agent's current shape, no UI changes were needed to the Agent Monitor. This icon was made clickable. When the user clicked on the icon, they see the following:

![Figure 27 -- Changing an Agent's Shape](image)
In the above figure, the top level dialog is the one of interest. The UI design mimics the design of the Edit Breeds window. This was done because the Edit Breeds window is very familiar to current users. The user can navigate through the various shape categories, select the new shape by clicking on it and press the Ok button to change the agent’s shape. The familiarity of the UI comes at a cost of potentially cluttering the screen. In the above figure, there are 4 windows: StarLogoBlocks, Spaceland (the two that always exist), the Agent Monitor and the dialog to change the agent’s shape. Some further research and user testing is necessary in order to determine whether or not this UI design is sufficient. For example, the Agent Monitor could be moved to the tabbed pane near the bottom of Spaceland (just like the Agents tab). Clicking on an agent would spawn a new tab and this tab would contain all of the same information as the Agent Monitor. Users would then have to be given the ability to close the tab. This would remove one window from the screen and help to reduce clutter.
12. Conclusion

One year ago, I sought out to make an impact on the Starlogo TNG project and leave my footprint. The project has been in existence for over ten years and I am fully confident that it will exist for at least ten more.

The measurable properties of a good user interface were discussed and they were fully taken into consideration when designing the various features that were added to Starlogo TNG.

I intended to add a multitude of features that would facilitate game creation. However, as the year went on, priorities did change and not all of the features made the cut. However, the ones that were implemented were done so deeply. They are in a fully usable and functional state.

Because of Starlogo’s anticipated longevity, not only was it important to follow good UI design principles, but it was also very critical that I followed good coding practices so that future UROPs and MEngs can read my code, understand it fully and have the capability to change and improve upon it.

Before I took on my project with the TNG group, the uses for Starlogo TNG were mostly limited to simulations (biological, physical, etc.) and very simple games. Now that I have had a year to work with the group, the scope of projects that can be created have expanded enormously. This is very exciting as it will entice children to use more of their creativity when working with the program and it will hopefully lead to greater interest in math and the sciences.
13. Acknowledgements

First and foremost, I would like to thank Professor Eric Klopfer for allowing me to be a part of this wonderful project. I would also like to thank Daniel Wendel for working so closely with me this past year and keeping me on track as well as helping me discover the right scope for my thesis. He was a good friend and very supportive of my work, as well as a great person to seek advice from. Without these two individuals, this thesis would have been impossible.

I would also like to acknowledge Professor Steve Ward and Professor Chris Terman for being gracious enough to allow me to be a TA for their course, 6.004 for the entire year. I wholeheartedly believe that being a TA was the single most rewarding experience I have had at MIT and I am thankful for being granted the opportunity (twice) to do this. I also would like to recognize all of the TAs that I have worked with over the past year. They were all great and enjoyable to work with.

My thanks go out to all of the other members of the TNG group. It was great working with the other MEng, Ling Tu, and I hope she had an experience as wonderful as mine. Also all of the UROPs (Peter, Lily, Bianca and others) have been great and they made my summer in the lab enjoyable.

I would also like to thank my girlfriend and roommate, Christina, for putting up with me all year. She had to deal with my late night hack-a-thons and had to constantly assure me that I would finish my thesis in time.

Lastly, I acknowledge all of the users of TNG. They really were my motivation. The users are the reason why I put so much work into the project. I thoroughly enjoyed being able to improve their experience with the program, help them learn more, and become interested in math and science.
14. References


