

Designing and Editing

2.5-Dimensional Terrain in StarLogo TNG

by

Daniel J. Wendel

S.B., C.S. M.I.T., 2005

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

August, 2006

[September 2006]

© 2006 Massachusetts Institute of Technology

All rights reserved

Author _____
Department _____

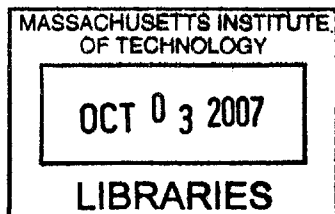
_____ Engineering and Computer Science
August 21, 2006

Certified by _____

Eric Klopfer
Education Program
is Supervisor

Accepted
by _____

Chairman, Department Committee on Graduate Theses



BARKER

Designing and Editing
2.5-Dimensional Terrain in StarLogo TNG

by

Daniel J. Wendel

S.B., C.S. M.I.T., 2005

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

August, 2006

Abstract

StarLogo TNG is “The Next Generation” in block-based decentralized programming for modeling and simulation software. Its aim is to make computer programming more appealing for students in middle school and high school. Part of the draw of StarLogo TNG is its 3-D rendered world called Spaceland where “agents” live on a terrain made of a grid of “patches”. This thesis evaluates and outlines the redesign of Spaceland and its associated terrain editor based on user-task analysis, and discusses the design of new data structures to support the desired features.

Thesis supervisor: Eric Klopfer
Title: Professor, Director, MIT Teacher Education Program

Table of Contents

1.	Introduction.....	7
1.1.	What is StarLogo?.....	7
1.2.	What is StarLogo used for?.....	9
1.3.	Drawbacks of StarLogo	10
1.3.1.	Text-based programming.....	10
1.3.2.	Simple 2D graphics.....	11
1.4.	Solution: StarLogo TNG.....	11
1.4.1.	StarLogoBlocks.....	11
1.4.2.	Spaceland – a 3D world	13
2.	Spaceland, version 0.9, Preview 1	15
2.1.	Moving from 2D to 3D	15
2.2.	Design	16
2.2.1.	Data storage	17
2.2.2.	Rendering.....	18
2.2.3.	Stomp and yank.....	18
3.	Level Editor, Preview 1	21
3.1.	Preliminary requirements.....	21
3.1.1.	Whatever an agent can do	21
3.1.2.	Some things an agent cannot do.....	21
3.2.	Design	21
3.2.1.	Overview and layout.....	21
3.2.2.	Tools	23
3.2.3.	Missing tools.....	26
4.	Evaluation of Preview 1	28
4.1.	Evaluation contexts.....	28
4.1.1.	After school at CCSC.....	28
4.1.2.	Physics class in Massachusetts	28
4.1.3.	Teacher Education Program lab at MIT.....	29
4.2.	User observations and evaluations.....	30
4.2.1.	General Spaceland and editor	30
4.2.2.	Observations	30
4.2.3.	Evaluation	31
4.3.	Terrain/level observations.....	32
4.3.1.	Mazelike.....	32
4.3.2.	Scenic.....	34
4.3.3.	Picturelike	34
4.3.4.	Agent-edited.....	35
5.	Revised Spaceland and editor requirements	38
5.1.	Spaceland features	38
5.2.	Block commands.....	39
5.3.	Editor tools.....	40
5.3.1.	Shaping tools.....	40
5.3.2.	Painting tools	40
5.3.3.	Agent tools.....	41
5.3.4.	General tools	42

6.	Spaceland and Editor, version 0.9, Preview 2	44
6.1.1.	Improved Spaceland speed	44
6.1.2.	Change color tool.....	45
7.	Spaceland, version 0.9, Preview 3	46
7.1.	New data structures.....	46
7.1.1.	Terrain structure.....	46
7.1.2.	Patch structure.....	47
7.2.	Textures.....	47
7.2.1.	Texture specification.....	47
7.2.2.	Combining textures with colors	50
7.3.	Rendering.....	50
7.3.1.	Steady-state rendering.....	50
7.3.2.	Changing terrain.....	52
7.4.	Extensions to blocks	53
7.5.	Missing Features	54
8.	Editor, version 0.9, Preview 3+	55
8.1.	Improved features	55
8.2.	Added features	56
9.	Future work.....	57
10.	Conclusion	58
11.	Acknowledgements.....	59
12.	References.....	60
	Appendix A: Spaceland data structure files.....	61
	Appendix B: Editor features to fulfill requirements	69

List of Figures:

Figure 1-1 The StarLogo Runtime environment. In this example, colored agents move around on a world made of black and yellow patches. 8

Figure 1-2 Some StarLogo Blocks..... 12

Figure 1-3 A portion of code written with StarLogo Blocks. 13

Figure 2-1 Several shapes found in the StarLogo TNG shape library. 16

Figure 2-2 One red patch is divided into nine sub-patches. Two sub-patches are outlined in gray for clarity. 17

Figure 2-3 Vertex Multipliers for stomp and yank operations. 19

Figure 2-4 One red patch that has been yanked, surrounded by green patches. 19

Figure 3-1 Layout of the terrain editor for Preview 1..... 22

Figure 3-2 A region of red patches that has been leveled, in the middle of a terrain full of green hills..... 23

Figure 3-3 A region of red patches that has been raised, in the middle of a terrain full of green hills..... 24

Figure 3-4 A crater drawn in the middle of a mound, forming a ring-shaped ridge..... 25

Figure 3-5 The color palette that pops up when using the “paint” tool. 25

Figure 3-6 A terrain that has been painted with several different colors..... 26

Figure 7-1 A sample of what a patch top source image might look like. The yellow rectangle indicates a possible portion of the image that would be specified by a patch’s texture coordinates. 49

Figure 7-2 An example of a possible patch side source image..... 49

Figure 7-3 A tilted red patch in the middle of green terrain 51

Figure 7-4 A green patch with brown sides raised two units above the surrounding terrain. 52

1. Introduction

StarLogo TNG (The Next Generation) is a 3D graphical programming environment designed to help increase computer literacy, especially in students in middle school and high school. Its goal is to use novel tools and cool display capability to get students to write programs, expanding their powers of expression on the computer. In particular, StarLogo TNG is designed to be a flexible tool to let users with little or no prior programming experience create powerful, graphically attractive games without the need for text-based coding [3].

At the heart of this approach are StarLogoBlocks and Spaceland. StarLogoBlocks are small graphical representations of the various control and data structures in a program, and Spaceland is the 3D projection of a customizable surface on which the game characters live. While the StarLogoBlocks are robust and nearly completely designed, the capabilities of Spaceland are not. Spaceland has now gone through two redesigns, and the editing tools needed to work with it are only beginning to take form.

This thesis project consists of the design and implementation of an editor for Spaceland. The project starts with a simple editor designed to meet the barest minimum requirements, evaluates it in light of actual usage, and concludes with the second iteration, a design that better meets users' real needs.

1.1. *What is StarLogo?*

In order to better understand the workings of StarLogo TNG, it is useful to start with its predecessor, StarLogo version 2, hereafter referred to simply as StarLogo. StarLogo was developed as a tool for decentralized thinking and modeling. With a few lines of code, developers can model bird flocking behaviors, termite mound building, simple ecological systems, or any number of decentralized systems [6]. However, StarLogo's text-based language can be difficult for younger students to grasp, and its 2-dimensional graphics are no longer state of the art. This section discusses some of the benefits of StarLogo, as well as some areas that can be improved to let StarLogo reach a more diverse, less computer-savvy audience.

Two concepts form the core of StarLogo – *patches* and *agents*. The world on which the agents live and interact is made of a 2-dimensional grid of patches. Each patch has certain built-in properties, like color and user-defined properties, called *patches-own variables*, that might include “vegetation cover” for an ecosystem model or “mound height” for a termite model. A palette on the left-hand side of the StarLogo window features brush and shape tools for painting on the grid of patches. The patches’ properties can also be modified programmatically by agents on or around the patches or by the “Observer,” an all-knowing entity with access to global properties of the model. In StarLogo, patches can even run a limited set of commands to modify themselves according to global properties such as the number of agents alive or the number of red patches. Patches are rendered as colored squares in the StarLogo runtime environment, as shown on the right-hand side of Figure 1-1.

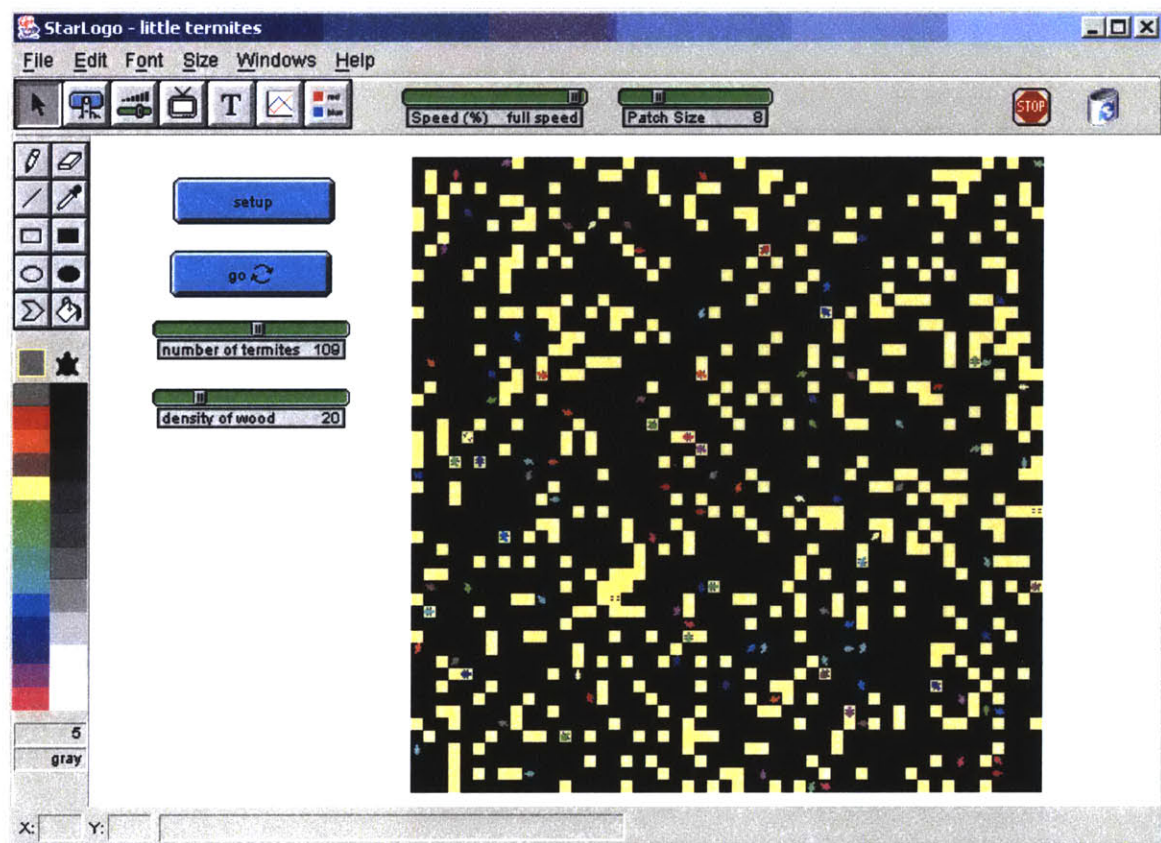


Figure 1-1 The StarLogo Runtime environment. In this example, colored agents move around on a world made of black and yellow patches.

Agents are the entities that can move around and interact programmatically with the patches and each other. Each agent is an instance of an object running “agent code.” The agents are divided into species, which can run different code based on a simple if statement. The agent code specifies the movement and behavior of agents on a decentralized, individual level, and each agent runs the code in its own virtual machine. In other words, each agent knows how it should move and behave based on the agent code for its species, but no central entity controls the group behavior of the agents. Any group behaviors that do emerge come from the interaction between individuals. Agents can have images associated with them, or can be invisible and simply paint patches as a record of their existence. Figure 1-1 shows the StarLogo runtime environment, with agents scattered around a world of yellow and black patches.

1.2. What is StarLogo used for?

StarLogo is a tool for decentralized thinking, aiding in the understanding of complex dynamic systems. Through interaction with StarLogo *models* (i.e. simulation projects), students can gain a better understanding of concepts such as:

- Predator/prey population dynamics
- Forest fire spread and containment strategies
- Bird flocking patterns
- Termite nest building

As well as many more [6]. By interacting with models, rather than simply viewing static information, students can gain first-hand knowledge about the concept being modeled. For more information on StarLogo as an educational tool, see the *Adventures in Modeling* book by V. Colella, E. Klopfer, and M. Resnick, available from Teachers College Press, and *New paths on a StarLogo Adventure* by the same authors [6].

1.3. Drawbacks of StarLogo

Despite its usefulness for many applications, StarLogo also has several drawbacks that prompted the creation of “The Next Generation.” These drawbacks fall into two main categories: difficulty of programming and lack of graphical appeal.

1.3.1. Text-based programming

The difficulty in programming StarLogo models stems primarily from its dependence on its text-based programming language. Although the language is not difficult compared to other text-based languages such as BASIC or PASCAL, it still presents a high entry barrier for users with little or no programming experience, such as most junior high and high school students have.

The first difficulty is syntax. New users must repeatedly look up the proper syntax for commands in order to use them correctly; the commands themselves give no indication of the syntax with which they should be used. Even for users with prior experience with other programming languages, syntax is always a bit of a guessing game without the documentation. For example, the formatting of brackets and parentheses in a StarLogo `if-else` statement, although easy to memorize, is unlike mainstream languages such as C, BASIC, and Java.

A related problem is that compile errors stemming from incorrect syntax are not apparent until the users compile the code. By that time they have often shifted their focus to another aspect of the model and must spend time re-familiarizing themselves with the problematic portion of code.

The final difficulty with the text-based language in StarLogo is its separation from the runtime environment. In order to change the code for a model, i.e. a StarLogo project, users must switch to an editing window, make changes, and then switch back to the runtime interface to test. This also applies when users want to test a command to see what it does – they must switch windows, type the command, save and compile, and return to the runtime interface to run.

1.3.2. Simple 2D graphics

StarLogo's use of 2D graphics rendering for its patches and agents, although sufficient for many models and simulations, may be a drawback for many new users who are used to the advanced 3D graphics of modern computer games. Since even games for young children now use 3D characters and worlds, StarLogo's graphics may not have the appeal or "coolness factor" needed to draw students whose goals are to make games or simply to have fun [5].

Although the above drawbacks do not diminish StarLogo's usefulness for its original purposes, they deserve consideration for a more important reason: the Computing Research Association (CRA) reported in May 2005 that interest in computer science as an undergraduate major in the US dropped 60% between 2000 and 2004. Even more distressing is that only 0.5% of incoming female college freshmen expressed an interest in computer science as a major, an all time low since the early 1970s [9]. Additionally, Gee [2] states that children, although fluent in "reading" video games, are largely inept in the other half of computer literacy, "writing" [5]. There is clearly a need to make programming more accessible to students across the board, and to provide an example of programming as an exciting, interesting activity.

1.4. Solution: StarLogo TNG

Enter StarLogo TNG, "The Next Generation" of StarLogo programming. StarLogo TNG's aims are similar to those of StarLogo: to provide a powerful platform for modeling and learning about decentralized systems. However, StarLogo TNG also addresses the problems and shortcomings of StarLogo, providing a programming environment that is both easy and *fun* for students to use. StarLogo TNG addresses these goals by introducing two significant changes to StarLogo: blocks-based programming and a 3D rendered world [5].

1.4.1. StarLogoBlocks

The StarLogoBlocks concept was developed as a graphical alternative to text-based programming. As in LogoBlocks [1], the graphical language after which StarLogo TNG's language is patterned, StarLogoBlocks commands take the form not of textual words, but of graphical "blocks" as shown in Figure 1-2. These blocks can be snapped

together like pieces of a jigsaw puzzle to form complete programs. Figure 1-3 shows a portion of code written in StarLogo TNG's StarLogoBlocks language.

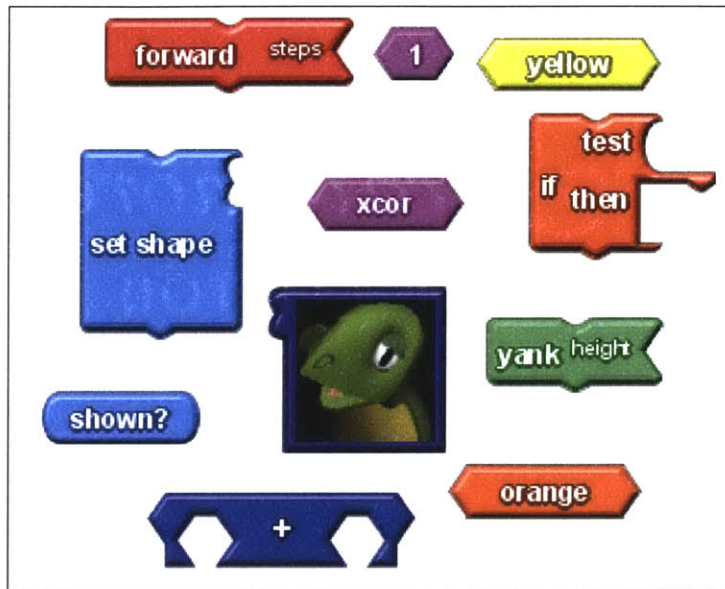


Figure 1-2 Some StarLogo Blocks.

In blocks-based languages, syntax can be given by visual cues, i.e. the shapes of the blocks themselves. It is therefore immediately obvious if two commands do not fit together syntactically, as their shapes are not compatible and hence do not snap together. In the same way, commands that require arguments are given “sockets” that must be filled. Notice, for example, the block labeled “set energy” in Figure 1-3. It has an angular socket on the right side, indicating that it requires a number argument, and conversely, the number block is shaped such that it fits into the socket.

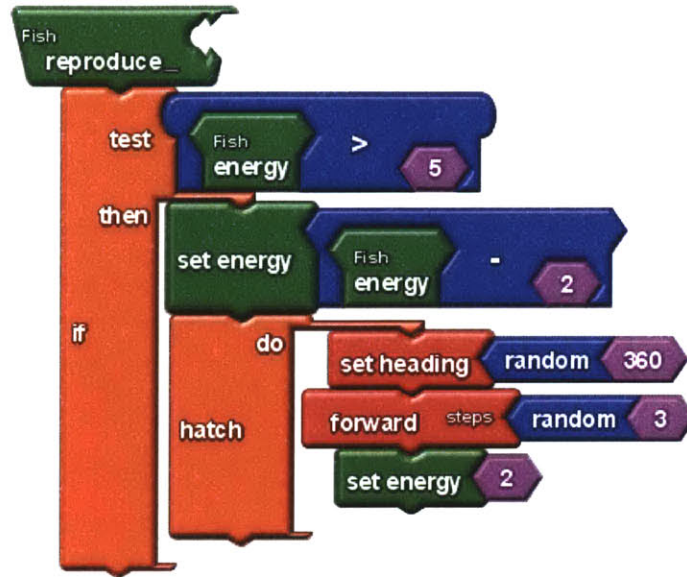


Figure 1-3 A portion of code written with StarLogo Blocks.

This shape-based syntax indication not only provides clues for what arguments are required by commands, but also provides error prevention and feedback [3]. Unlike StarLogo, where it is necessary to compile code in order to find out where the errors are, in StarLogo TNG one can see errors immediately; it is instantly obvious if two blocks placed next to each other do not snap together, and it is also obvious when a block's socket is empty.

1.4.2. Spaceland – a 3D world

The second large improvement in StarLogo TNG is the addition of Spaceland, a 3D rendered world in which the agents live. Spaceland takes StarLogo's concept of a grid of 2D patches and adds another dimension – height. The patches themselves are actually 2.5D. This means that for any (x,y) coordinate in the grid, there is exactly one height. In other words, the patches are rendered as a grid that cannot overlap itself. However, agents in StarLogo TNG can exist anywhere in the 3D space above and below this 2.5D grid of patches. It is our hope that the addition of this 3D graphical capability will make StarLogo TNG more enticing to today's youth [5].

Of course, the introduction of 3D into StarLogo TNG required more than just a change of rendering. Additional height-changing commands needed to be added to the StarLogo language to allow users to control the 3D properties of patches, as well as the

3D properties and movement of agents. Additionally, the old painting tools used in StarLogo are no longer sufficient; painting only covers a small subset of the new needs for editing 2.5D terrain. This thesis project focuses on designing a framework for and editing the new 2.5D terrain.

In this paper I introduce the first version of Spaceland along with a simple terrain editor I developed for it. I then evaluate Spaceland and its editor in light of feedback gained from real world use cases. I analyze some of the tasks these users tried to accomplish in order to develop a specification of requirements for the next iteration of Spaceland and its editor's design. Finally, I propose, implement, and evaluate a design for the next version of Spaceland and its editor.

2. Spaceland, version 0.9, Preview 1

The first version of StarLogo TNG to be released to the public was version 0.9, Preview 1. Preview 1 contains rough versions of most of the target features of StarLogo TNG, including a user interface for block-based programming and a 3D rendered window showing Spaceland, the virtual world in which StarLogo TNG agents live.

2.1. Moving from 2D to 3D

In order to move into 3D, StarLogo TNG had to add several new features to the original StarLogo design. The first step in moving to a 3D world was extending the StarLogo commands and agents to take three dimensions into account. To this end, agents were given several new properties relating to their 3D location and appearance, and several commands were added to the StarLogo language to change the properties. Additionally, a few commands were added for changing the heights of the patches in the terrain.

Each agent has a property specifying its shape. Although this concept is not new in StarLogo TNG, the shapes themselves are. Agents rendered in Spaceland must have 3D shapes, so an entirely new palette of shapes was developed. For Preview 1, the available shapes include basic shapes such as spheres, cones, and cubes, as well as letters, numbers, and more complicated character shapes. The more complicated shapes include .OBJ models of several animals, as well as .MD3 shapes imported from some of the countless Quake 3 websites. Spaceland's ability to draw shapes from both .OBJ and .MD3 file formats also allows users to import 3D shapes of their own into StarLogo TNG's shape library. Figure 2-1 shows agents of several different shapes from the StarLogo TNG shape library.

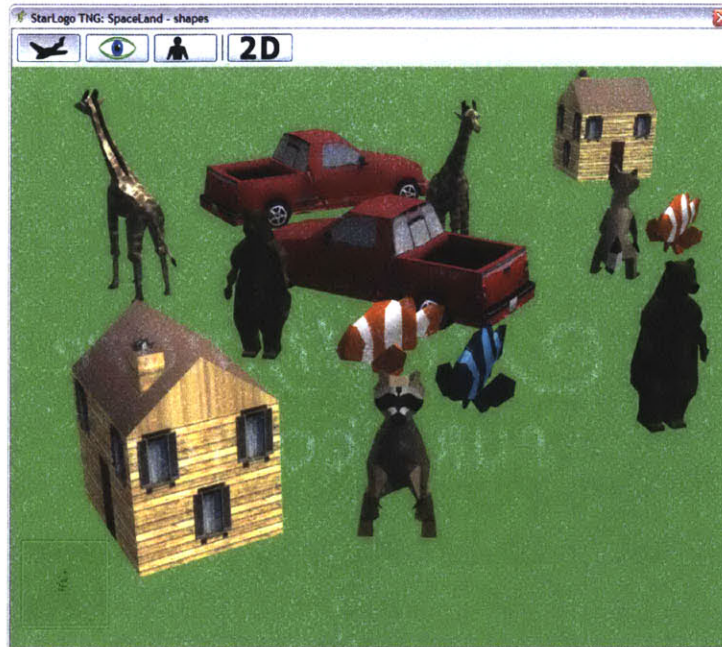


Figure 2-1 Several shapes found in the StarLogo TNG shape library.

The most obvious agent property that was added is altitude, which specifies an agent's height above the terrain. The set-altitude block was added to control this property, allowing users to create programs in which agents appear to be floating above the terrain or even wading through it. A corresponding reporter block was also made, which reports the agent's current altitude.

Similarly, the most obvious patch property that was added is height, which specifies a patch's height relative to the middle of Spaceland, or what would otherwise be known as sea level. This height can be changed and read by four commands that were added to StarLogo TNG's language: `stomp`, `yank`, `patch-height`, and `ph-ahead`. `Stomp` and `yank` are the commands to change a patch's height. When agent executes a `stomp` command, the patch under that agent is lowered by the given amount. Conversely, when an agent executes a `yank` command the patch under the agent is raised. The `patch-height` command reports the height of the patch the agent is standing on. The `ph-ahead` command reports the height of the patch one unit in front of the agent.

2.2. Design

The patches in Preview 1 of Spaceland are arranged in an evenly spaced square grid 101 patches on a side, allowing for patch coordinates from -50 to 50. Patches share

vertices at the corners and sides, making for a smoothly connected terrain. Additionally, each patch is divided into nine sub-patches, arranged in an evenly spaced 3 x 3 grid. The height is stored at each corner in the grid, giving the editor sub-patch height control. Each of the nine squares is rendered as two triangles, which is standard practice for OpenGL rendering. Figure 2-2 shows one red patch surrounded by green patches on the side of a mound. Notice the nine sub-patches each rendered as two triangles. Emphasis is added on two sub-patches to make the distinction clearer.

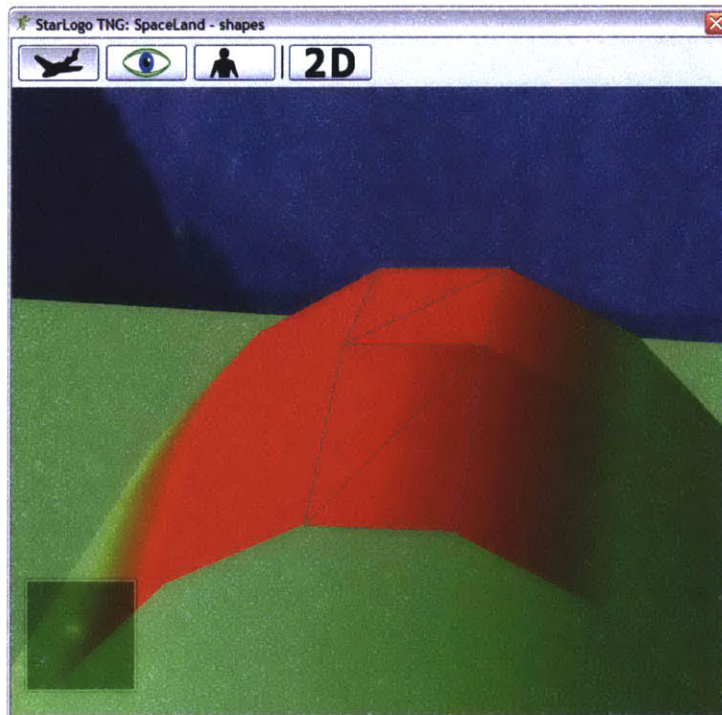


Figure 2-2 One red patch is divided into nine sub-patches. Two sub-patches are outlined in gray for clarity.

2.2.1. Data storage

The data for this terrain of patches is stored in two Java ByteBuffers that can be read both by the Java front-end and the C virtual machine that make up StarLogo TNG. The first ByteBuffer contains data with which the agents can interact. This includes the patch color, the patch height (as an average of its nine sub patch corners), and a pointer to the patch heap, a location in memory where any user-defined variables for that patch are stored.

The second ByteBuffer stores data related to how the patches are rendered, including the nine sub-patch heights. Although this data is modified by the `stomp` and `yank` commands, the data is only ever read by the terrain-rendering code.

2.2.2. Rendering

Spaceland, including agents and the terrain, is rendered using JOGL, a Java binding to OpenGL. The terrain is rendered as a series of 303 triangle strips (because there are 303 sub-patches on a side), each containing 606 triangles. Each of these triangle strips can be converted to an OpenGL display list for much more efficient rendering. However, a display list must be recomputed whenever the data for one of its triangles changes. This means that the execution of common commands, such as `stomp`, to change the color of a patch, or `stomp`, to change the height of a patch, results in substantial recalculation. Sections 4.2.3 and 6.1.1 address this issue in more detail.

2.2.3. Stomp and yank

Due to the connected nature of Preview 1's Spaceland terrain, it is impossible to change the height of a whole patch without changing its neighboring patches. Given the 3 x 3 grid of sub-patches, however, is possible to change the height of part of the patch more than other parts. Preview 1's design of `stomp` and `yank` takes advantage of this fact. Sections 4.3.1 and 4.3.2 discuss how this connected version of `stomp` and `yank`, although good for creating smooth mountainous features, is less than ideal for creating vertical walls.

The design of `stomp` and `yank` in Preview 1 is based on the observation that if all of the patches in a certain region are `yanked` by the same amount, the region should still appear to be flat and connected, but higher than the surrounding terrain. In order to maintain this behavior, a grid of multipliers was devised that indicates what fraction of the total height a given vertex should move. For example, the four corner vertices of the patch are assigned a .25 multiplier so that if the four patches sharing a corner are all `yanked`, the vertex between them will be at the same height as their centers. Figure 2-3 shows the multipliers for each vertex of a patch. Figure 2-4 shows a red patch that has been `yanked` by one unit, surrounded by green patches. Sidebar 1 shows how using this grid, when `yanking` four patches sharing a corner, yields a plateau.

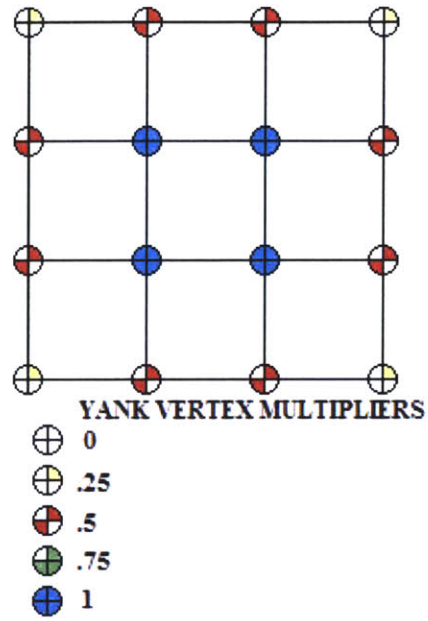


Figure 2-3 Vertex Multipliers for stomp and yank operations.

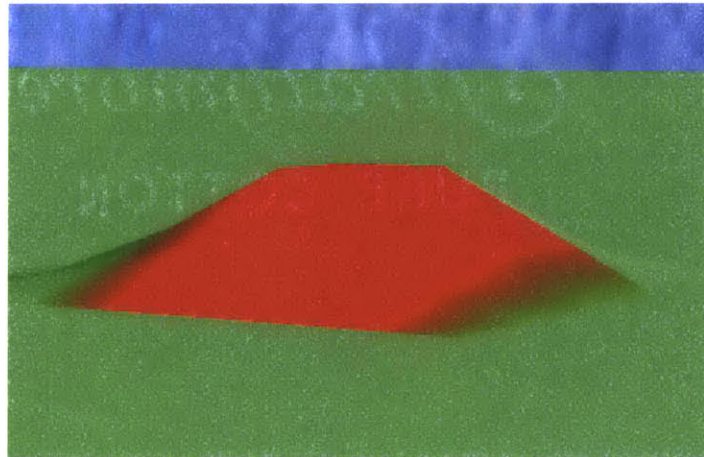
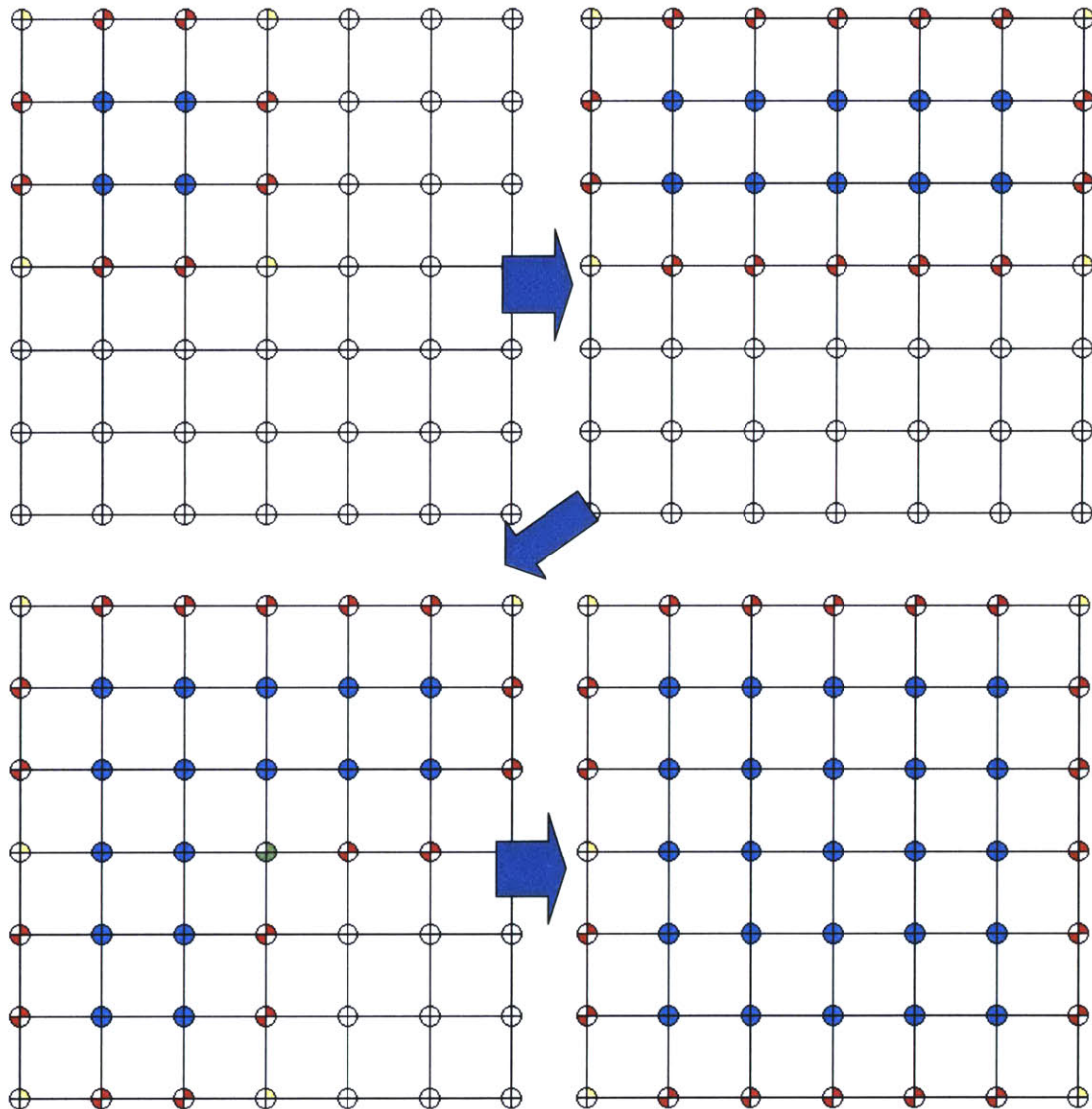


Figure 2-4 One red patch that has been yanked, surrounded by green patches.

Sidebar 1

This progression of images shows the heights of the vertices of four adjacent patches as the patches are yanked one-by-one. Notice that at each iteration, the patches that have been yanked share a middle region (colored blue) with height equal to 1.



3. Level Editor, Preview 1

The design of Spaceland also gave rise to the need for level editor to allow users to shape and paint the terrain. To this end, I developed a simple editor with a bare minimum of tools that we could distribute as a part of this first Preview release of StarLogo TNG.

3.1. Preliminary requirements

The first step in designing this simple editor was to develop some basic requirements. The first set of requirements included actions that could be done by agents running block code, such as setting the height or the color of patch. The second set of requirements was for macroscopic tools that an individual agent could not do.

3.1.1. Whatever an agent can do

The two properties that an agent can control about a patch are its height and its color. However, agents can only affect one patch at a time. Therefore it seemed useful to require tools that could:

1. Control the height of a region of patches, and
2. Control the color of a region of patches.

3.1.2. Some things an agent cannot do

Three other tools were required for editing the terrain in ways agents could not. These high-level requirements were for tools that could:

1. Draw smooth mounds,
2. Place agents around the terrain, and
3. Change the dimensions of the terrain's grid of patches.

3.2. Design

3.2.1. Overview and layout

Figure 3-1 shows the general layout of the terrain editor for Preview 1. The terrain is represented by a grid of squares in the middle of the window. Tools for editing the terrain are along the left-hand side of the window, next to a bar used for choosing heights. Along the top left of the window are buttons for saving and loading terrains, and the bottom right has buttons for changing the view.

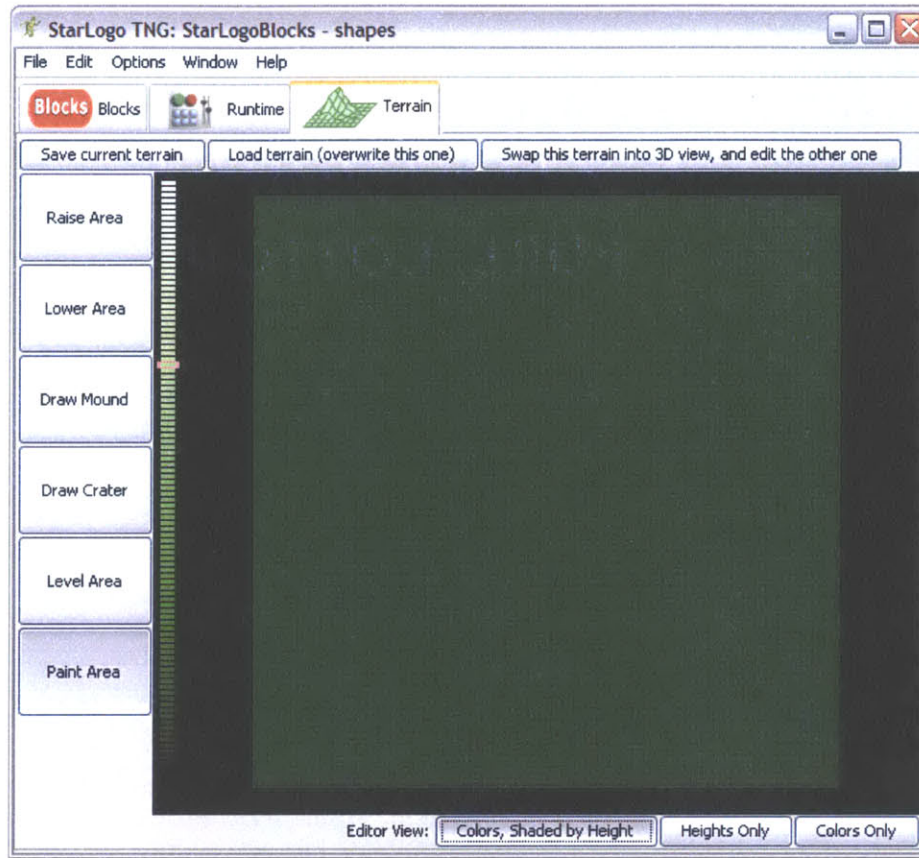


Figure 3-1 Layout of the terrain editor for Preview 1.

The editor for Preview 1 inherited an unfortunate “edit then swap” methodology from its predecessors. In this design, Spaceland displays one terrain, while the editor controls another. In order to make Spaceland show the terrain he/she just edited, the user must click a button to swap the terrains, bringing the terrain from Spaceland into the editor, and putting his/her terrain into Spaceland. The large button on the upper right hand side of the editor window controls the swapping of terrains.

The editor has three selectable viewing modes for the terrain. The default mode, *colored height*, shows the colors of the patches, but shades them according to their heights, with higher patches being shaded more brightly. The *height-only* mode ignores the colors of the patches, shading them instead using a height-dependent gradient. This gradient ranges from black for -50 to white for positive 50, passing through shades of green in the middle for added contrast. The *color-only* mode ignores the height information of the patches, displaying only their actual, unshaded colors.

3.2.2. Tools

The terrain editor for Preview 1 has four tools that, along with the height slider, provide the functionality of the terrain editor. They are listed below along with a description of how they work.

Level: The level tool brings all of vertices of all of the selected patches to the same height, effectively leveling the patches. The position of the height slider determines what height the patches will be brought to. Figure 3-2 shows a region of red patches that has been leveled, in the middle of a hilly terrain full of green patches. Notice that of the two visible “walls” of this plateau, one is green, while the other is red. This is due to the connected nature of the terrain; the south and east edges of any region will slant if necessary to connect to the surrounding terrain.

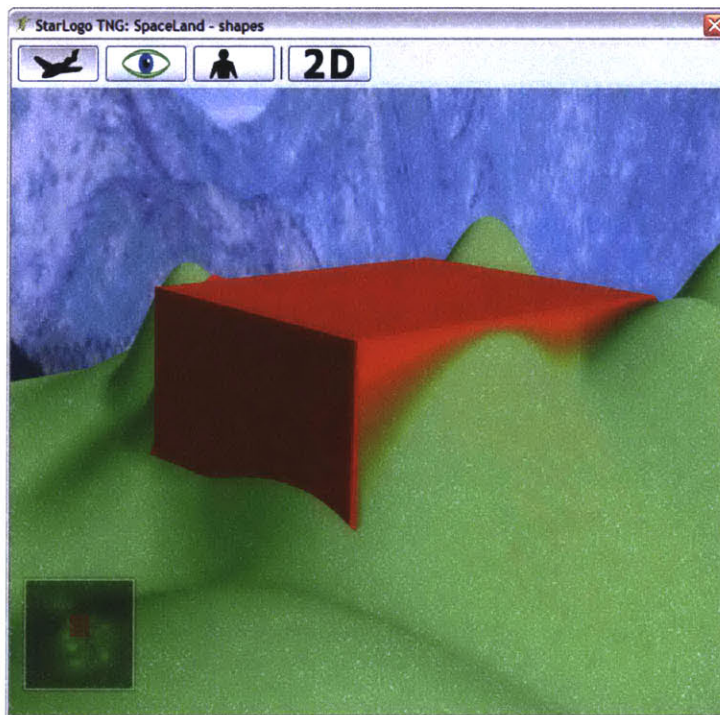


Figure 3-2 A region of red patches that has been leveled, in the middle of a terrain full of green hills.

Raise/lower: The raise/lower tools interpret the height slider differently than the level tool. Rather than reading it as an absolute value, the raise and lower tools interpret it as an offset. The raise tool simply adds the height on the height slider to the height of every vertex of the selected region of patches. Similarly, the lower tool subtracts the height of

the height slider. Figure 3-3 shows a region of red patches that has been raised in the middle of a hilly terrain of green patches.

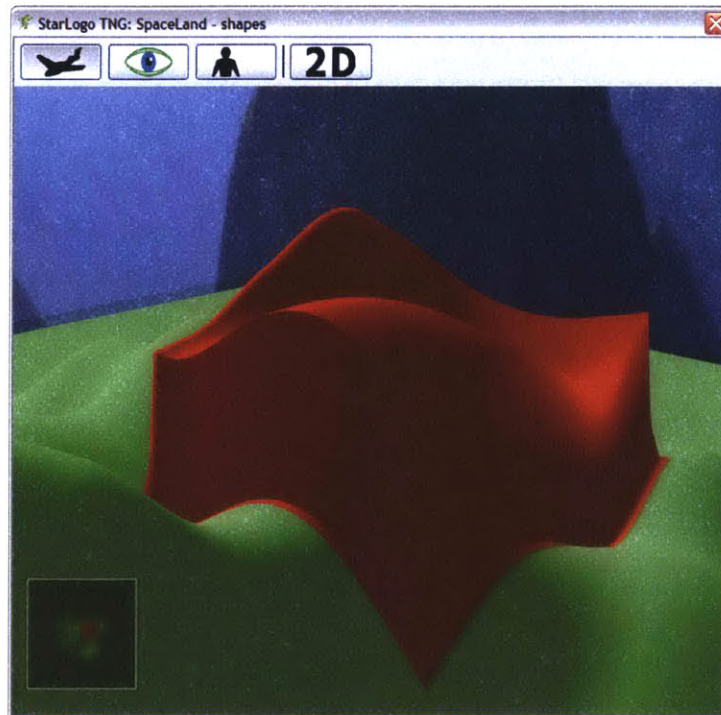


Figure 3-3 A region of red patches that has been raised, in the middle of a terrain full of green hills.

Mound/crater: The mound (and similarly, crater) tool can be used to make curved terrain features. Like the raise and lower tools, the mound and crater tools interpret the height slider's value as an offset rather than as an absolute height. However, unlike the raise and lower tools, the mound and crater tools do not apply this offset uniformly to the selected patches. Instead, they change the center patches by that amount and taper off toward the edges of the selected region, thereby creating a mound or crater. An inverted cosine curve is used to determine the tapered heights, since it yields a rounded shape which transitions smoothly from level terrain to the slope of the mound. Figure 3-4 shows a crater drawn in the middle of a mound, forming a ring-shaped ridge.

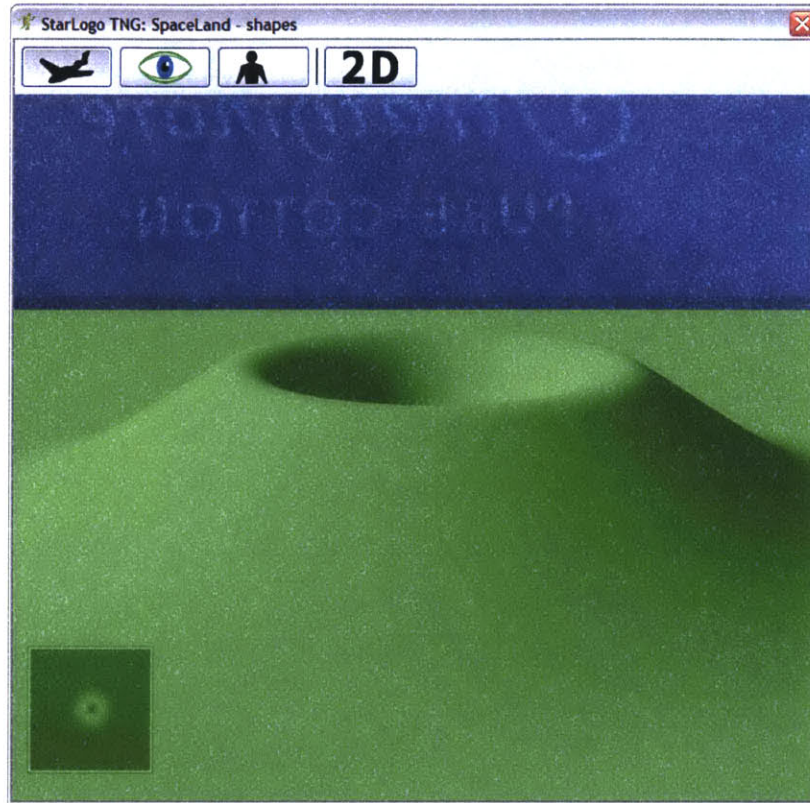


Figure 3-4 A crater drawn in the middle of a mound, forming a ring-shaped ridge.

Paint: The paint tool sets the color property of the patches in the selected region. When the user clicks on the paint tool button, a small dialog box, shown in Figure 3-5, pops up asking the user to choose a color. Subsequently selected patches will be set to the selected color. Figure 3-6 shows a terrain that has been painted with several different colors.



Figure 3-5 The color palette that pops up when using the "paint" tool.

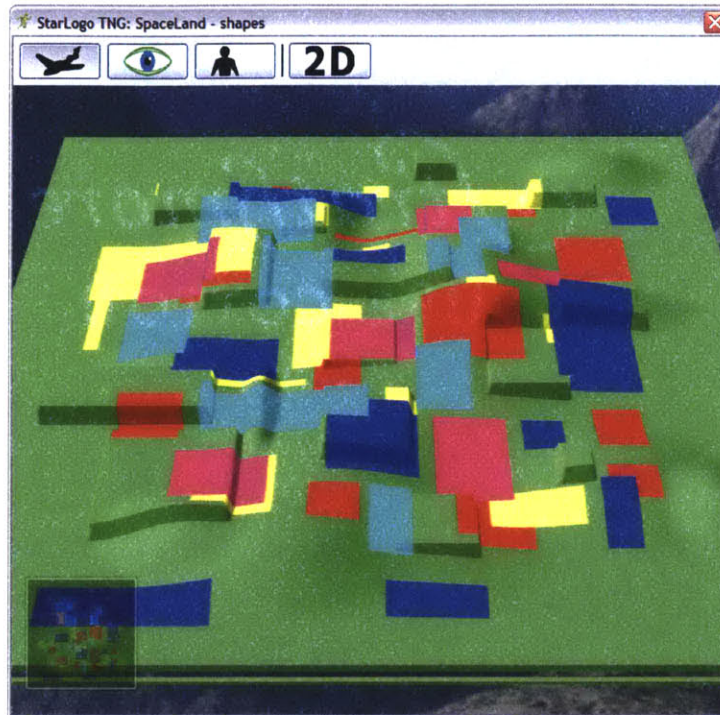


Figure 3-6 A terrain that has been painted with several different colors.

3.2.3. Missing tools

Two of the requirements for the terrain editor are conspicuously missing from the design of the editor for Preview 1: the ability to place agents, and the ability to resize the terrain.

The ability to place agents on the terrain was temporarily implemented for this editor. However, the editor had to store agent placement data internally, since the design of the terrain data structure in Spaceland had no place for storing data. Additionally, agents and agent properties are not stored with project files in StarLogo TNG. Therefore, although users could place agents, they could not recover or save this placement. For this reason, the agent placement tool was disabled until the rest of the StarLogo TNG system could support it.

Terrain resizing faced a similar lack of support in the backend. The size of the terrain, rather than being stored in a data structure, was hard-coded into several interdependent modules of the StarLogo TNG virtual machine and Spaceland. Even if the terrain editor changed its representation of the size of the terrain, Spaceland would

still render the terrain as a grid of 101 x 101 patches. Terrain resizing was also disabled until sufficient backend support for it existed.

4. Evaluation of Preview 1

4.1. Evaluation contexts

The following evaluation of Preview 1 is based on observations of users using the system in three contexts. The first was an after school program at a nearby junior high school. The second was a ninth grade conceptual physics class in Massachusetts. The third was the Teacher Education Program lab at MIT where StarLogo TNG is being developed.

4.1.1. After school at CCSC

The first context in which we observed users interacting with StarLogo TNG was at an after school program at the Community Charter School of Cambridge. At CCSC our users were seventh, eighth, and ninth grade students with a wide range of previous computer experience. They all chose the StarLogo TNG after school program over other options, presumably lured by the idea that they could make their own computer games. The after school program lasted approximately 9 weeks, with most students attending between two and eight sessions.

The curriculum used for the after school program was developed by Kevin Wang, in conjunction with the StarLogo TNG developers. An improved and updated version is available on the StarLogo TNG web site as a tutorial for new users. The idea of teaching computer programming concepts through the use of StarLogo TNG with this curriculum was presented at the International Conference for the Learning Science during the summer of 2006 [4].

The computers used for this after school program were Dell laptops with Intel Celeron processors and 256 MB of RAM. They had an integrated Intel graphics chipset with shared graphics memory.

4.1.2. Physics class in Massachusetts

The second context in which we collected observations about StarLogo TNG was in a ninth grade conceptual physics class at the Governor's Academy in Newbury, Massachusetts. The students were all ninth graders with a general distaste for math and a wide range of previous computer experience.

The observations from this user group were sent to us by their teacher, Hal Scheintaub, who also developed the curriculum that was used in this class. He will be presenting a poster at the International Research Group on Physics Teaching (GIREP) conference in Amsterdam about his success in using StarLogo TNG programming to teach physics concepts. As one of our most active users, Dr. Scheintaub also sent us numerous observations about his own interactions with StarLogo TNG.

The computers that these students used were older Dell OptiPlex desktops with 1.7 GHz Intel Pentium 4 processors and 256 MB of RAM. The computers initially had 32MB ATI Rage 128 Ultra graphics cards, but these proved to be much too slow to display Spaceland at an acceptable frame rate. We purchased and installed NVIDIA GeForce FX 5200 graphics cards instead, which enabled the computers to display models where the agents were simple shapes such as spheres and cubes.

4.1.3. Teacher Education Program lab at MIT

The group of StarLogo TNG users at MIT consisted of a mix of researchers, developers, and college students. The researchers and developers were well aware of StarLogo TNG's capabilities, as each of them had a part in its design. The college students in this group were taking education classes at MIT and had a range of computer backgrounds, although most had strong mathematical skills and several had programmed before. Most of the observations from this group were made by the users themselves and emailed to the StarLogo TNG developers email list.

Most of the users in this group were concerned with making models that demonstrated educational concepts such as osmosis, erosion, planetary orbit, predator/prey population dynamics, decision-making, or emergent properties. However, some were also interested in making sample games such as simple role-playing games or first-person shooters.

The computers that these users used ranged from old laptops with integrated graphics to then state-of-the-art AMD Athlon 64 based systems with NVIDIA 6600 GT graphics cards. Many users in this group also worked on Macintosh computers, and one used Linux on his laptop.

Both the CCSC after school program and the ninth grade conceptual physics class focused mainly on teaching programming concepts through the use of StarLogoBlocks. Although students in both groups made heavy use of the terrain editor, their training consisted of a brief demo after which they were allowed to experiment. The users in the Teacher Education Program (TEP) lab similarly focused predominantly on the blocks, although they did interact often with Spaceland and the terrain, as many of their models had agents interacting with and changing the terrain.

Since the focus of this paper is on Spaceland and the terrain editor, I will focus on observations related to those aspects of StarLogo TNG, though they form only a small subset of the observations we were able to gather from the users. For further discussion regarding other aspects of StarLogo TNG, see the detailed description of StarLogo TNG by lead designers Eric Klopfer and Andrew Begel [3] or Corey McCaffrey's thesis on improving block programming efficiency [7]. Additionally, look for future work by Ricarose Roque on debugging StarLogo TNG models.

4.2. User observations and evaluations

The following sections summarize some of the common observations from the three user groups, and evaluate the design of Preview 1 based on those observations.

4.2.1. General Spaceland and editor

The following two sections discuss observations and related evaluations of General Spaceland and editor functionality.

4.2.2. Observations

Although users rarely paused to make observations about parts of StarLogo TNG that were working well, most of the students seemed to enjoy using the terrain editor to change the appearance of the patches in Spaceland. Once they were shown how to swap terrains between Spaceland and the editor, users tended not to have any trouble with it. In fact, the typical usage pattern was to make a change with a tool, swap terrains to view the change, and then swap back to continue editing.

The users did have a few general complaints. Of these, the most frequent was that StarLogo TNG was simply too slow. On the computers that the middle/high school

students were using, models in which agents walk around Spaceland interacting with each other could run at between 10 and 20 frames per second. However, frame rates for models in which agents were changing the colors or heights of patches dropped to between one and five frames per second. Even on the fastest computers tested, frame rates for such models hovered below 30 frames per second.

The other two frequently mentioned complaints were that: 1) there was no undo function and 2) there was no way to “reset” a terrain to the way it was before the agents in a model modified it. The lack of an undo function had the potential to be very frustrating because it often resulted in lost work due to mistaken edits. The lack of a reset mechanism also resulted in lost work, but in this case, it was due to the running of a model, a purposeful action essential to the StarLogo TNG design.

4.2.3. Evaluation

Based on the observation that users would frequently swap their terrain into Spaceland to see what it looked like, it is apparent that the swapping mechanism is an unnecessary annoyance to the user. Rather than making users swap terrains, it would make more sense simply to view the terrain in both Spaceland and the editor concurrently, updating Spaceland's view after every edit.

The obvious solution to the problem of speed is to improve the efficiency of Spaceland to make it faster on more platforms. This merited a careful look at the rendering code for Spaceland. Radu Berinde, one of our developers, found several places in which the rendering code created unnecessary new variables or did unnecessary recalculations. However, at the heart of the biggest slowdown was the decision to render the terrain as many triangle strips and to internally mark modified strips as needing re-rendering. This design meant that if even one patch in a strip was changed, the entire strip would need to be re-rendered. It did not take very many agents running terrain-modifying code before the entire terrain would need to be re-computed and re-rendered every frame.

The lack of reset functionality was also due to a design shortcoming. Because both edits and agent actions took place on the same terrain data structure, agents running a program with terrain modifying commands would destroy prior editing work that the

user had done on that terrain. Despite there being two terrains in the system, there was no way to copy from one to the other. The only way to reset a terrain after agents had modified it was to save the terrain beforehand and then reload it.

What many users desired in an undo tool was actually the ability to erase terrain features and reset the patches to their initial flat, green state. However, many other users desired a true multi-command undo history. Still, due to direct manipulation style of the editor, users were able to rebuild lost features, indicating that the lack of undo at least did not hinder their ability to create terrains.

4.3. Terrain/level observations

In observing the levels made by the users, it became apparent that most of the levels could be classified into three categories: mazelike, scenic, and picturelike. A fourth type of level, in which agents shaped the terrain somehow, was less common in the student models but more common in the educational simulations. Of course, many levels displayed characteristics of multiple categories of levels. The categorization here is for the sake of analyzing tasks specific to those characteristics. The following four sections describe the characteristics of these levels, as well as problems users encountered with them, in more detail.

4.3.1. Mazelike

The levels that I categorize as mazelike were characterized not by their appearances, although they often looked like mazes, but by the way agents in the models interacted with them. In these models, agents would read the terrain properties, taking different actions if they encountered patches of various heights or colors, for example. The most common usage of patch property differences was to provide boundaries for agents in the model, in the form of walls or colored lines.

Observations: The most noticeable problem that users had with this type of level was that agents would often walk partway up walls that were supposed to form a boundary for them. This problem would appear regardless of whether the agents were checking the height or color of the wall patch as the condition for stopping. A problem whose cause

was much more difficult to trace was that sometimes agents would fail to recognize the boundary colors at all.

Many users also expressed dissatisfaction with the coloring of the walls they would make. In some cases, this was due to the fact that the south and east sides of a wall would be painted the color of the top of the wall, while the north and west sides would be painted the color of the surrounding terrain. However, many users also said they wanted to be able to paint the walls independently of the colors of the tops of the walls or the surrounding terrain.

Evaluation

The problem of agents walking on the sides of walls was related to the connected nature of the terrain. In order for one patch to be higher than its neighbor, either a part of its or its neighbor had to be sloped to connect the two. In Preview 1, the southern and eastern sub-patches would slope as necessary to connect to the surrounding terrain. This meant that it was possible for an agent to be standing on a patch whose reported height was zero, but, by standing on the sloped portion, to appear to be halfway up the side of a wall. For the same reason, agents could be found standing halfway up a green colored wall when the color of the top of the wall, for example red, was a boundary color for that agent.

The connected nature of the terrain was also the reason that users could not paint sides of walls independently of the tops. Since the side of wall was actually the southern or eastern extension of a patch, it was impossible for the side of the wall to be colored differently than the patch itself without redesigning the data structure that held patch properties.

The scenario in which agents would simply not respond to a color was traced back to the way colors are stored in StarLogo TNG. Agents checking for color are checking for numeric matches, since colors are stored as numbers in the virtual machine. However, users editing a terrain would choose a color from the palette presented by the paint tool. The problem occurred when users thought they were choosing green, for example, but were actually choosing a color that would be internally represented as green+1 or green-1.

4.3.2. Scenic

The terrains that I categorize as scenic often had rolling hills, plateaus, walls, and several colors. Their purpose was simply to act as a backdrop for a model; if agents interacted with the terrain I classified it as mazelike or agent-edited. Many of these models used hills or small walls to limit visibility, or large, gentle mounds which seemed more realistic than the purely flat terrain.

Observations

Unfortunately, users found that it was difficult or impossible to make terrains look as realistic as they would like. Neither the editor tools nor the agents were capable of adjusting the slope of an individual patch, so structures such as ramps were impossible to build. Additionally, some users mentioned that a tool that could import heightmap data from other sources would be extremely useful.

Spaceland is rendered inside a “skybox,” a large six-sided shape with images rendered on the inside. Spaceland’s skybox shows images of a mountain range beneath a blue, sunny sky. Many users making scenic terrains expressed a desire to be able to choose a different skybox from a library, or least to be able to turn it off. Similarly, some users, especially in the TEP lab, said that the ability to make the terrain itself invisible would be useful for some of their models.

Evaluation

Unlike the problems encountered by the creators of mazelike levels, the problems creating scenic levels were due to a lack of editing tools and agent program commands rather than core design shortcomings. Additional tools for shaping the terrain can be added to the editor, and commands for changing the skybox and showing or hiding the patches can be added to the StarLogo TNG language.

4.3.3. Picturelike

The defining characteristic of a picturelike terrain is its use of color for artistic or aesthetic purposes. The patches in these terrains might be colored to form shapes, a smiley face, a map, or even a low resolution color photo or drawing.

Observations

Users who wanted to draw pictures on the terrain immediately complained that there was only one painting tool, and that it could only draw rectangles. I therefore immediately changed the editor slightly so that the selected paint color would be applied to all tools. This way, by selecting a zero height and using the mound tool, users could draw circles and ellipses. While this change was a vast improvement, users still had to draw many shapes, such as diagonal lines, one patch at a time.

Some users wanted turn the entire terrain into a map or photo. For these users, neither of the rectangular nor elliptical drawing tools were of much help; they could both be used to fill in large areas of color, but detail work had to be done one patch at a time.

Evaluation

Although the rectangle and ellipse tools were useful, they were simply inadequate for many of the tasks users desired to accomplish. A common task, drawing a line, had no associated tool. Users did not even attempt more complicated shapes such as stars or curves, presumably because the tools did not exist.

Drawing a map or photo on the terrain would also be greatly assisted by the proper tools. For example, users said that the ability to import a picture from an external file would be very useful. An editor tool could be developed to assign patch colors based on an image, assigning patch colors according to the pixel colors of the image.

Fundamentally, though, the one color per patch limitation would still prevent users from drawing detailed pictures on the terrain. For the users, it would be much more useful for each patch to be rendered with a texture, i.e. a small image, rather than simply a color. In the case of importing a picture, it would be ideal to chop the picture into 101 x 101 sub pictures and have each patch display its own sub picture. This ability would also be ideal for maps or cases where users wanted the terrain to look like rock, dirt, or grass.

4.3.4. Agent-edited

Agent-edited terrains occur in models in which the initial features are less important to the user than the features that arise as the result of agents running modifying commands on the patches. Although there are many such models, I will focus on two particularly informative ones, the erosion model and the termite model.

In the erosion model, users could swap a hilly terrain into Spaceland and watch as eroding agents carved away at it. The eroding agents were born randomly near the center of terrain and would flow downhill from there, **stomping** the terrain as they went in an attempt to mimic erosion.

In the termite model, the setup code randomly scattered raised yellow patches (“woodchips”) throughout the terrain. The termite agents would then wander randomly around the terrain. If they bumped into a raised yellow patch, they would “pick up the wood chip” by **stomping** the patch and setting its color to green. They would then wonder until they were next to or on top of another yellow patch, in which case they would “drop the wood chip” by **yanking** the yellow patch. As this process ran, the wood chips would begin to form “piles” (raised yellow mounds) rather than being randomly scattered.

Observations

Two observations arose from the erosion model. First was the observation of how vital resetting the terrain is for some models. It became frustrating having to redraw the terrain every time, despite the fact that no particular terrain shape was required. The second observation was that the “eroded” terrain did not quite look eroded because the slopes of the patches were unchanged, with the exception of the south and east connecting sub-patches.

The most interesting observation that arose from the termite model was that, given enough time, the wood chips would form one pile in a corner of the terrain. This was confusing at first, but we soon determined that it was due to the fact that corner wood chips, being inaccessible from three quadrants, were the least likely to be picked up because they only needed three surrounding yellow patches in order to be completely protected from being picked up.

Evaluation

The erosion model confirmed the need to add the ability to reset terrains to the design of Spaceland, despite there being no existing support for it as discussed in the general evaluation of Spaceland.

The observation about the appearance of the eroded terrain had a less obvious solution. Although `stomp` and `yank` seem to work fine for the termite model, their behavior did not seem quite adequate for erosion. For the termite model, the patches were manipulated largely independently of each other, with each patch having an integral number of wood chips on it, given by its height. In the erosion model, on the other hand, it seemed that nearby patches should be able to affect the slope of a patch, perhaps causing it to slope downward toward lower patches. These seemingly contradictory requirements for `stomp` (and therefore `yank`) indicated a possible need to separate the concepts into unique commands - one for independent patch height changes, and one that would weakly affect surrounding patches.

The corner wood chip migration phenomenon in the termite model led us to realize that for some models, it is inappropriate for agents to “bounce” when they hit the edge of the terrain. Rather, some models might prefer that agents “wrap” around to the other side of the terrain when they walk off one edge.

5. Revised Spaceland and editor requirements

Based on the task analysis and evaluations from Preview 1, as well as a brainstorm of possible future tasks and problems with the StarLogo TNG development team, I drafted a revised set of requirements for Spaceland and the terrain editor for future versions of StarLogo TNG. The requirements are discussed below, organized according to whether they are Spaceland features, additional block commands, or editor tools.

5.1. Spaceland features

Vertical walls: The terrain should be able to support vertical walls, meaning that no portion of a patch should be forced to slope to attach to its neighboring patches. For mazelike levels, this would prevent agents from walking halfway up a wall before registering a patch height change. For the termite model, this would provide height independence, allowing patches to get taller (as they received more woodchips) without affecting nearby patches.

Ramps: The terrain should also allow for ramps, in which patches are connected to one another and share the same slope. Patches should be able to tilt to connect to neighboring patches if desired. This would allow for the creation of more interesting scenic levels, such as tiered levels for first-person video games.

Mounds: Mounds in the new terrain design should look reasonably smooth. Many scenic and agent-edited levels have come to depend on this fact. Additionally, some models, such as the erosion model, require that the terrain remain smoothly connected for realism.

Paintable walls: The sides of walls should have the ability to be painted separately from the patches on the top of or surrounding the wall. This feature was requested by users creating the first three terrain types.

Textures: Patches and walls should be able to display textures (i.e. pictures) as well as colors. These textures should be able to be placed one per patch, stretched across multiple patches, or, in the case of walls especially, tiled to cover the entire surface

regardless of size changes. Both picturelike and scenic terrains would benefit from the added image resolution and realism of textures.

Customizable dimensions: Terrain dimensions should be user-specified rather than simply defaulting to 101 x 101 patches.

Customizable skybox: As many users requested for scenic models, the skybox should be customizable through an included library or, at the very least, the ability to import external skybox files. The skybox should also be able to be turned off entirely.

5.2. Block commands

Improved height control: The design of `stomp` and `yank` in Preview 1 does not allow for the degree of height independence that some models require, nor does it allow patches to affect each other as much as other models require. The height commands should be redesigned to allow better control of both neighbor-independent and neighbor-dependent height changes.

Wall/cliff detection: Rather than checking heights of patches in mazelike levels, agents should be able to detect the presence of a wall or cliff ahead. Commands need to be added for this purpose.

Show/hide patches: Commands should be added for either showing or hiding the patches. These commands should not erase any block properties, which should be readable and changeable regardless of visibility.

Bounce vs. wrap: Since bouncing is not the ideal behavior for some models, commands should be added that can toggle between bouncing mode and wrapping mode. When in bouncing mode, the fence around the terrain should be rendered. Otherwise the fence should be hidden.

Load terrain: The `load terrain` command should reset the terrain to match a previously stored terrain from the editor. Using this command, the user could create a terrain in the editor and be able to use it in a model without fear of it being destroyed; to restore the terrain that was previously created, he/she would simply use this command. Of course this command also requires an editor feature to save terrains. This will be described later.

Load level: Once the virtual machine is modified to allow for the saving and loading of agents, the `load level` command should load not only the terrain associated with a given level, but also the agents on it.

Get/set texture: With the addition of texture to the terrain, commands need to be developed for reading and writing texture values to the patches. However, since one patch's texture maybe only a small part of a multi-patch picture, more work is necessary to determine appropriate behaviors for these commands. For their first iteration, they should at least enable agents to read the texture from one patch and set another patch to have the exact same texture.

5.3. Editor tools

5.3.1. Shaping tools

Mound/crater: The mound and crater tools in future versions of the editor should work similarly to those in Preview 1, which successfully created smooth terrain features.

Raise/lower: In order to enable the creation of vertical walls and cliffs in the terrain, the raise and lower tools should be changed so that they no longer affect surrounding patches.

Ramp: For cases where users want height changes to affect surrounding patches, the ramp tool should allow users to tilt the region of patches to form a connected ramp between areas of different heights.

Import heightmap: In order to allow users to create realistic looking terrain, a tool should be added for importing heightmaps from external files. A good starting point for this tool would be to allow the importing of grayscale bitmaps similar to those used as heightmaps for WarCraft III [10].

5.3.2. Painting tools

Shapes: Tools should be added to the editor for the easy creation of filled or outlined shapes. At the very least, these should include rectangles and ovals, although other shapes should be added as users request them.

Line: A line tool should be added for drawing straight lines. This could be extended in future versions to include tools for drawing curves or polygons. Other extensions, such as line width, could also be added in future versions.

Wall: the wall tool should be able to paint the vertical sides of raised patches. Since the editor currently only displays the tops of patches, this tool has to provide a means to distinguish which sides are currently being edited.

Texturing/coloring: Each of these painting tools should be able to work with both colors and textures. A design needs to be developed for how colors and textures should interact. For example, should they be able to coexist on the same patch? These tools could also be extended to paint any property of the patch, for example height or a variable value, rather than just color or texture.

Import image: The import image tool should be able to import an external image file, for example a drawing or photograph, and use it to either color or texture the patches.

5.3.3. Agent tools

The tools in this category are a result of brainstorming rather than user feedback, as no agent placement tools of any type existed in Preview 1. Additionally, since agent placement depends on fundamental changes to the virtual machine data structures, these tools are not merely improvements to Spaceland itself. For this reason, their addition may be delayed longer than other improvements to Spaceland.

Place agent: The place agent tool should allow users to select a breed and place an agent of that breed on the terrain. The tool should also allow for control of agent properties, such as shape, height, heading, and color. Additionally, agents already placed on the terrain should be able to be moved with a simple click and drag.

Paint/scatter: Another useful tool is one that allows for an army of similar agents to be placed on a region of the terrain. These agents could either be painted in lines or scattered about an area with some user-specified density. User feedback will be required to determine what design works best for this tool.

Camera focus: One agent related tool that was requested by users even in Preview 1 was a tool to control camera focus. Using this tool, users should be able to specify which agents, if any, should be tracked by the camera.

5.3.4. General tools

The tools in this category do not pertain directly to the changing of patch properties. Rather, they are general features of the editor, the editor's interface, and the editor's interaction with Spaceland and the rest of StarLogo TNG.

Improved selection: The highlighting when selecting regions of patches for various tools should take into account the actions of the tools. For example, the raise tool should select rectangular areas, while the mound tool should select elliptical areas. Each of the drawing tools should highlight only the patches they would affect, whether rectangles, ovals, other shapes, or simple lines.

Editor zoom: Some users indicated that it was too hard to select individual patches in the editor due to their small size. On the other hand, users with very low resolution screens complained that the editor was too big, and that parts of it did not fit on the screen. To address these problems, the editor should be able to zoom and scroll its view of the grid of patches in the terrain.

Height slider labels: In addition to showing the range of available heights and highlighting the selected height, the height slider should also have a label indicating the exact value of the current selection. This will help to avoid guesswork in models that rely on patch heights.

New terrain: The level editor should have a simple tool for creating new terrains. When creating a terrain, the user should be able to specify its dimensions. At least for now, it seems reasonable to keep a terrain's dimensions constant throughout its life. However, if user feedback indicates that it is useful to change terrain dimensions even after editing the terrain, that capability could also be added.

Instant updates: Since so many users of Preview 1 would frequently swap terrains to see what their edited terrain looked like in Spaceland, future versions of the editor should

simply edit the active terrain so that edits made in the editor are instantly reflected in Spaceland.

Save to block: In order for block commands to be able to load a terrain or level from a block, the editor must be able to save terrains and levels to memory slots that are associated with and can be accessed by blocks. The design for this should be something that works well for both the editor and the virtual machine that runs the blocks.

Save to file string: The StarLogo TNG project file format consists of strings of encoded data. For this reason, the terrain editor should be able to save all of the existing terrains to a file string which can be included in the save file. In the same way, the editor must be able to read such string to load terrains.

Undo/redo: Many users requested undo capability in the editor. This should work similarly to the undo/redo schemes of other editing or drawing programs, i.e. undoing one tool's actions at a time. Undo and redo in the terrain editor should be unrelated to undo and redo in the StarLogoBlocks interface.

6. Spaceland and Editor, version 0.9, Preview 2

StarLogo TNG version 0.9, Preview 2 was released approximately one month after the end of the programs in which students were using Preview 1. It was almost exclusively bug fixes and additional features for the StarLogoBlocks programming environment, and largely ignored Spaceland and editor issues. However, one improvement in Spaceland and one additional feature in the terrain editor from Preview 2 are noteworthy.

6.1.1. Improved Spaceland speed

The one significant change to Spaceland was its speed. By carefully examining the code, Radu Berinde was able to eliminate several excess variables as well as several unnecessary computations in the rendering code. He also converted many strings of OpenGL calls into OpenGL display lists, which could be sent to the graphics card once and then repeatedly called to avoid unnecessary data transfer.

The results of Radu's work were astounding. Frame rates for simple models in which agents were not changing the terrain jumped from the mid-20s on most computers to well over 100 frames per second. Models that had previously been unbearably slow became fluid and interesting. Additionally, Radu also put a mechanism in place to slow the frame rates of Spaceland while the user was editing blocks in the StarLogoBlocks workspace. In doing this, he was also able to speed up block rendering and dragging by allocating more CPU resources for them while the user was not focusing on Spaceland.

Unfortunately, though, models in which agents changed the terrain were still just as slow. The OpenGL display lists, which saved so much time when nothing was changing, had to be recomputed every time something did change. For example, any time one patch changed, the display lists for the three triangle strips that it was a part of all had to be recomputed and re-sent to the graphics card. With only a handful of agents changing the terrain of a model, it was possible for the entire terrain to need to be recomputed within the span of a few frames.

6.1.2. Change color tool

The change that was made to the terrain editor as an immediate response to student comments from Preview 1 was the addition of a “change color” tool. Previously, the terrain shaping tools had no effect on the colors of the patches. However, multiple users, especially those creating scenic or picturelike levels, wanted to be able to shape and paint terrain at the same time.

To achieve this goal, I added a tool that users could use to change the default color of the editing tools. Subsequent use of terrain shaping tools would cause the shapes patches to take on this default color. The tool is represented as a button whose background was colored according to the selected color. When the user wanted to change the default color, he/she would click on the button, and would be presented with the color palette that had previously been used for the paint tool. Since the paint tool no longer required that users choose a color (since there was now a default), the color palette was removed from the paint tool, which simply painted using the default color.

One minor change was made to the palette in transferring it from the paint tool to change-color tool. As stated earlier in the evaluation of mazelike levels, many users could be confused when they thought they were painting one color on the terrain but then the agents they programmed did not respond to that color. This bug was caused when the color values stored internally were off by one or two from what the user expected. To prevent this problem, the color palette was made to display the internal value of the selected. For example, if the user chose a color two shades lighter than the internal constant for green, the color palette would display “green+2” so that the user could either select a different color or modify his/her program appropriately.

7. Spaceland, version 0.9, Preview 3

StarLogo TNG, version 0.9, Preview 3 was released in August 2006 and contained multiple bug fixes as well as a new version of Spaceland. Preview 3 is the most current release and is both faster and more stable than any of the prior releases. This section describes the design and implementation of Spaceland for Preview 3.

7.1. *New data structures*

Many of the features desired by users of Preview 1 were fundamentally not supported by the Spaceland data structures. For example, some things, such as the terrain dimensions, were hard-coded into the design. Other things, such as vertical walls, required geometry information that was simply not stored in the structure. For these reasons, I completely redesigned the Spaceland data structures with the new requirements in mind, keeping in mind the need for efficiency and high frame rates.

7.1.1. **Terrain structure**

The terrain structure contains general high-level information about each level. In Preview 3 the only data stored in the structure are:

- Terrain dimensions
- Patch top texture sub-grid dimensions
- Patch side texture sub-grid length
- A pointer to the list of patches for the terrain

However, this data structure is meant to be extended as additional features are added. For example, the list of initial agent positions and other information for this level should be stored in the terrain structure once the agent placement feature becomes available. There is also a possibility that each terrain will need to have its own copy of memory for *patches-own* variables so that the value of such variables can be stored in the save file or terrain block for this level. If this becomes the case, a pointer to that memory should be stored in this data structure as well.

The terrain dimensions specify, predictably, the dimensions of the grid of patches that makes up the terrain. The patch top texture sub-grid dimensions specify how many sub-textures are stored in the patch top source image and in what arrangement (see the

Textures section below for details). Similarly, the patch side texture sub-grid length specifies how many side texture strips are contained in the patch side source image. The pointer to the list of patches for the terrain is the terrain structure's link to the patches themselves.

Both the terrain structure and the patch structure, discussed next, can be found in `patch.h` in Appendix A. This data can be accessed from the Java front end of StarLogo TNG through `TerrainData.java`, also available in Appendix A.

7.1.2. Patch structure

Each terrain structure has a pointer to a corresponding list of patches that make up the terrain grid. Each patch stores the following information:

- Four corner heights of the patch
- A pointer to the patch variables
- Color
- Texture
- Texture coordinates of northwest and southeast corners
- Two “side” structures where each side contains
 - Color
 - Texture
 - Two horizontal texture coordinates

The patches list is stored in a shared `ByteBuffer` so that both the StarLogo TNG virtual machine and the Java front end can read and modify the structure.

`TerrainData.java` contains the methods for manipulating the patches from Java. It is included in Appendix A.

7.2. Textures

The design for incorporating textures into the display of Spaceland is not immediately obvious from the data structures, so this section describes it in more detail. The important aspects of the design are the way in which textures are specified and the way in which colors and textures are combined.

7.2.1. Texture specification

In 3D graphics, the term “texture” typically signifies both a source image and the coordinates for what portion of that image to draw on a surface, in this case the top or

side of a patch. In that sense of the word, the texture for the top of a patch is specified by both the texture image and the texture coordinates of the patch.

In OpenGL, texture information can also be incorporated into display lists for speed. However, while experimenting with different portions of the rendering code, Radu discovered that changing the source image for texturing was an expensive operation that would make rendering all of the patches much slower than with colors alone. For this reason, the design that I chose based on his advice uses only two source images – one for the tops of the patches, and one for the sides of the patches. If Radu or a subsequent developer finds a faster way to switch source images, the texture property could be used to specify which source image to use per patch.

These two source images used in the current design are very large, with room for several smaller images to be painted on them. In this way, the system can mimic having multiple source images by using the patch's texture coordinates to specify that only a small portion of the source image should be rendered on the patch. Since each patch stores texture coordinates for both the northwest and southwest corners of the top of the patch, the patch can be drawn with any sub-rectangle of the source image. Figure 7-1 shows an example of what the patch top source image might look like, with a sample rectangle drawn demonstrating what a patch's texture coordinates might be to specify a particular portion of the source image. [Figures 7-1 and 7-2 use photographs from www.free-pictures-photos.com with permission.]

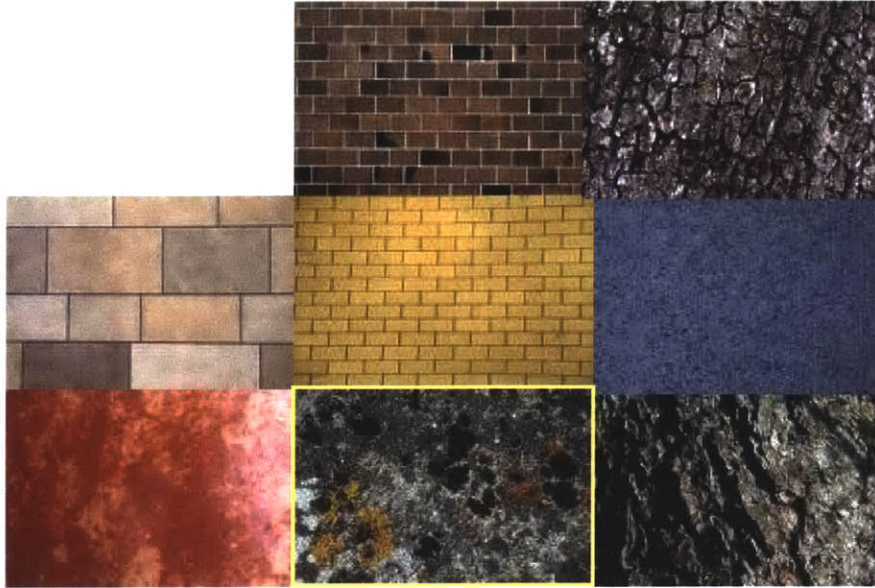


Figure 7-1 A sample of what a patch top source image might look like. The yellow rectangle indicates a possible portion of the image that would be specified by a patch’s texture coordinates.

For additional convenience, the texture property of the patch can be used to specify a sub-texture number that, together with the data stored in the terrain structure, can specify a new origin, letting the texture coordinates be interpreted as being relative to the sub-texture’s corner rather than being relative to the corner of the huge source image.

The sides of patches are textured slightly differently, as the height of a patch may change and thereby stretch the side. With fixed rectangular texture coordinates, this would result in the texture stretching or compressing each time a height changed, which could lead to undesirable visual effects (imagine a brick wall whose bricks could shrink or grow in height). Therefore, patch sides only store horizontal texture coordinates, which define a vertical “strip” in the second source image. As the height of the side changes, it simply shows more or less of this strip, even repeating the image if necessary to cover the whole side. Figure 7-2 shows an example of what the patch side source image might look like.



Figure 7-2 An example of a possible patch side source image.

As with the patch tops, the texture property of the patch side can be used to shift the texture coordinates so that they are interpreted as being relative to a given strip rather than being relative to the source image as a whole. By using this method, it is possible to change the texture of the side of a patch by changing the texture number without having to recalculate its texture coordinates.

7.2.2. Combining textures with colors

In order to render both a color and a texture for each patch, the textures' colors are modulated by the patch's color. For example, if a patch's color is red, only the red portions of the texture will be displayed, leaving the other regions dark. In this design, the colors of the textures and patches are interdependent. For example, a red patch covered by a green texture will be rendered as dark, since the colors cancel each other out. However, a white color allows the texture to be displayed as it looks in the source image, and a white texture allows the underlying patch color to shine through in full brightness.

Using this design, patches that are meant to display only their colors should be given a plain white texture. Similarly, patches that are meant to display only their textures should be set to have a white color. Setting either the color or the texture to black prevents anything from being drawn on the patch.

7.3. Rendering

The rendering of Spaceland was also completely redesigned to support the new data structures. The new renderer was designed with speed as the primary goal, and it met that goal better than any previous version.

7.3.1. Steady-state rendering

Each patch in the terrain is rendered as a triangle fan, a group of four triangles sharing a common central vertex. This design allows the patch to crease along either diagonal, or to tilt at any angle depending on the heights of the four corners. Figure 7-3 shows a red patch tilted, surrounded by green patches.

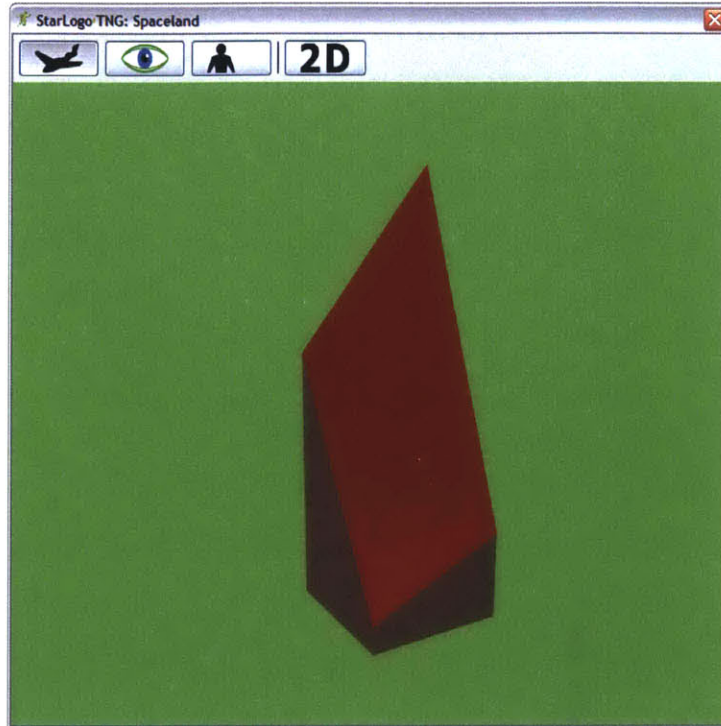


Figure 7-3 A tilted red patch in the middle of green terrain

The patches are rendered in sub-groups in this design as opposed to strips in the old design. The size of these groups can be modified in the rendering code, but groups of 16 (4 x 4) seemed to work best for a variety of different models. These groups are converted to OpenGL display lists so that they can be rendered much more quickly in the case that the terrain is not changing.

The sides of patches are only rendered where the corner heights of neighboring patches do not match up. Each patch only stores data for two sides, but since the patches form a grid, it is more accurate to think of patch sides as shared between the two patches that bound the side. The same side data is rendered regardless of which of the two patches is higher. No sides are rendered around the outside edge of the terrain. Figure 7-4 shows a green patch with brown sides that is raised two units higher than the surrounding blue patches.

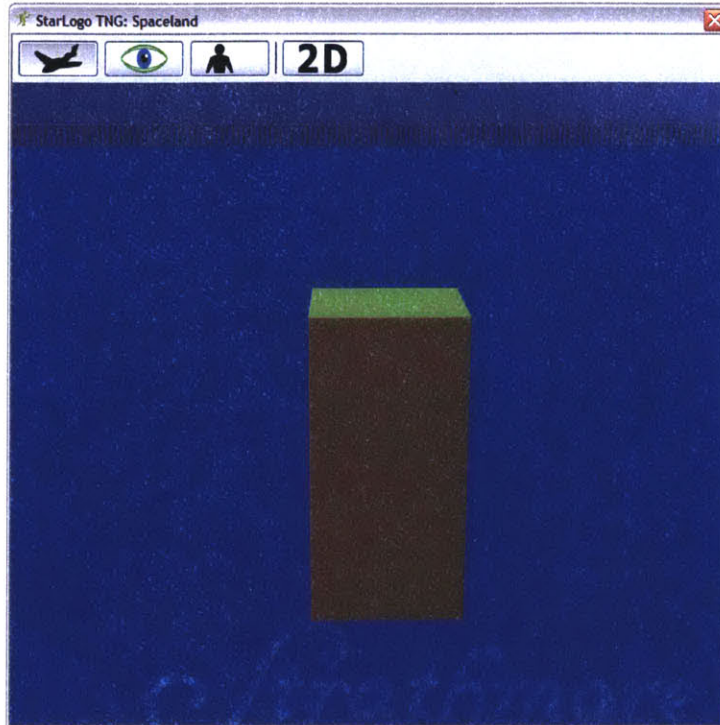


Figure 7-4 A green patch with brown sides raised two units above the surrounding terrain.

7.3.2. Changing terrain

The new design for the terrain and rendering is much better at dealing with terrain changes than the old version of Spaceland was. Rather than marking strips as dirty when any patch in them changes, as Preview 1 did, the new design keeps a list of individual patches that have changed. Additionally, due to the smaller group size (16 patches, as opposed to 101) used in the display lists, the new design can do considerably less work to refresh the terrain surrounding a changed patch.

Additionally, even if enough patches are being changed to require the recalculation of every sub-group, the new design's lower triangle count and more efficient data structures allow the entire terrain to be updated at better than 35 frames per second even with 2000 agents traveling around the terrain, painting and changing the heights of patches as they go. This is approximately a 10x speed increase over Preview 1.

7.4. Extensions to blocks

The following block commands were added to the StarLogo TNG language to allow models to better interact with the new terrain:

Dig/build: These are straight-walled operations, changing the height of a patch without affecting nearby patches. `dig` lowers the patch, and `build` raises it.

Show/hide patches: These commands can be used to turn the rendering of the patches on or off. However, even if the patches are invisible they still retain all of their properties such as height, color, texture, and any variables they might have.

Show/hide skybox: Although Preview 3 does not yet support customizable skyboxes, which require either an import tool or a library of images, it does support toggling the visibility of the existing skybox for models in which no skybox at all might be preferable.

Bounce/wrap: When the `bounce` command is executed, the fence is rendered around the perimeter of the terrain and agents will bounce at the edges. When the `wrap` command is executed, the fence disappears and agents that walk off one edge of the terrain appear on the other side.

Load terrain: This command takes a terrain index, and loads the associated terrain into Spaceland as the new active terrain. Currently there are 15 available terrain indices that users can fill from the terrain editor

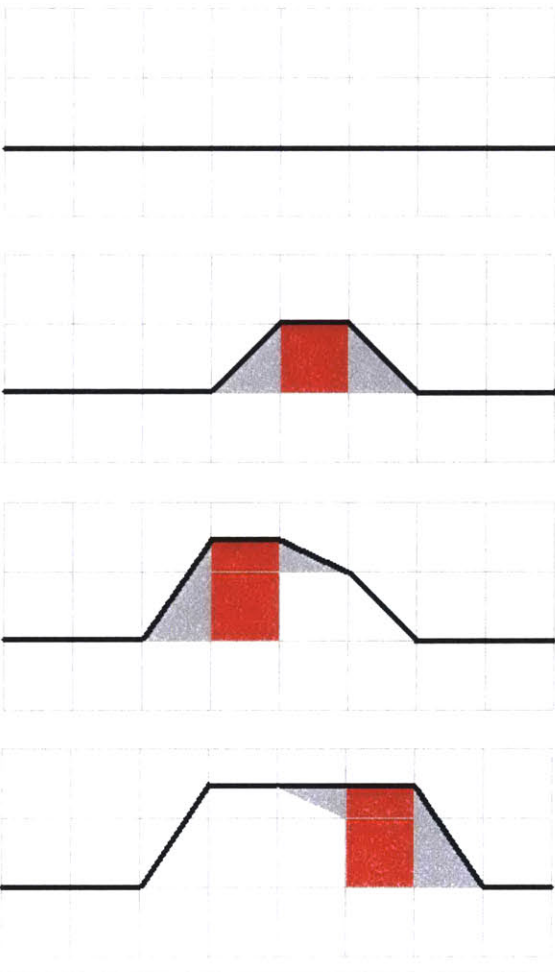
Get/set texture: Currently these commands simply read and write the texture property of the patch. This works well when the sub-texture mechanism is used, but for textures that are meant to be stretched across several patches, this could lead to improper behavior. These commands need more investigation to determine if it is useful to expose the concept of texture coordinates to users or if there is a better abstraction that can be presented.

New stomp/yank: The `stomp` and `yank` commands were designed in this version of StarLogo TNG to be the neighbor-dependent counterparts to `dig` and `build`. Whereas `dig` and `build` modify patch heights without regard to neighboring patches, `stomp` and `yank` behave differently depending on their neighboring patches because they keep the

corners of the affected patch connected to the corners of the surrounding patches. See Sidebar 2 for a detailed description of the new design and behavior for `stomp` and `yank`.

7.5. Missing Features

One command is unfortunately missing from Spaceland and the blocks in Preview 3: `load level`. As stated previously, the StarLogo TNG virtual machine does not currently support importing or exporting agent data from or to different data structures. As no change was made to the virtual machine design for Preview 3, the `load level` command was also excluded. However, with the addition of the terrain data structure to the design, there is now a place to store agent placement data once the virtual machine supports it. Spaceland is ready to support this new feature once a suitable design can be found for the mechanism to interact with the virtual machine.



Sidebar 2

The progression of images to the left shows a side view of several patches being `yanked`. The patch that is being `yanked` in each image is shaded red, while the effect on neighboring patches is shaded gray. In Preview 3, `yank` is given an “auto-leveling” behavior. This means that if the patch being `yanked` starts out sloped, the first portion of the height difference is contributed only to the lower vertex, thereby making the patch level. Once the patch is level, the amount of height left over is applied to the entire patch. This behavior is most clearly visible from the second to third images at left – first the lower vertex is raised (which takes $\frac{1}{2}$ height because it is only $\frac{1}{2}$ of a patch), and then the remaining $\frac{1}{2}$ height is applied to the patch, making its final height equal to 1.5. The third `yank` yields a flat plateau at height 1.5.

8. Editor, version 0.9, Preview 3+

The terrain editor for Preview 3 is nearly the same as that for Preview 1 functionally, although it has been completely reworked internally to support the new terrain format. In anticipation of the next preview release (3+), several small but important features have been added or improved to unlock some of the functionality of the new terrain format.

8.1. Improved features

Change color: The `change color` tool from Preview 2 was updated for the current version of the editor to include a combination of color and texture. In the new editor design, color and texture are only considered independently in this tool. Everywhere else, the “default color” has been changed to mean the color + texture combination defined in the change color tool. In the tool, the color palette has been modified to also show possible textures that the user can choose from. The user chooses both a texture and color. The green color that used to be the patches’ starting state is now a green color and a solid white texture.

Heights-only view: The heights-only view has been modified in the current editor to show a much wider range of colors to improve height contrast. By using height-dependent polynomial equations for the red, green, and blue components of the colors, this view now displays a spectrum from black to white, going through blue, red, and yellow as height increases.

Height slider: The height slider has also been updated to match the color spectrum of the heights-only view. A label has also been added to it which displays the exact value of the currently selected height.

Colored height view: The colored height view, which is the default for the editor, has been modified to display the combined colors of the patch and its texture, all modulated by height.

Instant Updates: The current version of the terrain editor only works on the current terrain, allowing edits to be shown in Spaceland immediately. The swapping feature of

older versions of the editor has been replaced by the ability to load and store terrains to and from this active terrain and an array of 15 available storage locations.

8.2. Added features

Texture-only view: A texture-only view has been added to the editor that behaves much like the color-only view, displaying the textures of the patches unmodulated by height or patch color.

Erase: Many users requested an erase tool for the terrain. Although unsure of what an erase tool should do, I designed it to reset affected patches back to their initial default state, which is level at height 0 with color set to green and texture set to solid white.

New level: Using the new level tool, users can create new terrains and specify their dimensions. When creating a terrain, the newly created terrain is loaded as the active terrain so that edits can be seen immediately. If an unsaved terrain is currently active when a new terrain is created, the user is prompted to save.

Save to block: The save to block tool allows users to save the active terrain into one of 15 available terrain storage locations. These terrains are saved with the project file and can be reloaded into Spaceland using the `load terrain` block.

Load from block: The load from block tool allows a user to load a terrain from storage into the active terrain, overwriting the active terrain. If the active terrain is unsaved, the user is prompted to save before loading.

9. Future work

Although the Spaceland data structures have been redesigned to support many more of the features that users need, many of those features are not yet included as tools in the editor. Appendix B contains an outline of features for the editor that would allow it to fulfill the requirements described in Section 5.

The ultimate goal in my opinion is to allow direct manipulation of Spaceland itself, letting the editor exist as an overlay to the Spaceland window. Further investigation and user study would need to be done to determine if editing in the 3D environment itself is as intuitive as the use of a 2D grid, but I suspect that at least for tools such as the wall painter, the added dimension would greatly increase the simplicity to the user.

In general, all of the features of Spaceland and the editor need to be continually tested and evaluated, leading to a cycle of iterative design focused on the user's needs and preferences.

10. Conclusion

StarLogo TNG is “The Next Generation” in graphical, decentralized programming, and may be a way to engage students in programming who would never be interested in it otherwise.

One of StarLogo TNG’s main draws for users is its ability to render agents in a 3D world called Spaceland, where the agents live on a terrain formed by a grid of patches with which the agents can interact. In order to design a version of Spaceland that supported the features required by users of StarLogo TNG, I first evaluated older designs in light of user feedback and observations and then designed and implemented new data structures to support the desired features.

Part of the fun and usefulness of Spaceland comes from the ability to edit the terrain to form different types of levels or worlds. These levels might include features with which the agents interact, or they may simply be aesthetic additions that make the levels more appealing to users. I designed and evaluated the level editor associated with the first Preview release of StarLogo TNG, and then updated it to work with the new and improved data structures made for the current version. Exciting future work exists in this area, with the possibility of extending the editor to fulfill all of the users’ requirements, perhaps blending the editor into Spaceland itself as an overlay allowing direct manipulation of the terrain from Spaceland.

As with all user-centered designs, Spaceland and the editor need to be continually evaluated in light of user feedback and observations. As the expectations of users change, so should the design of Spaceland and the editor, forming a cycle of iterative design that can help to keep StarLogo TNG appealing and useful to children and adults in the future.

11. Acknowledgements

This thesis would not have been possible without the help of many friends and colleagues. I would like to thank Eric Klopfer for his guidance and leadership throughout the course of this project. A team is largely defined by the leader, and the StarLogo TNG team is the best I have ever worked on. Corey McCaffrey patiently explained countless details of StarLogo TNG to me when I started work on the project, and quickly became a good friend. It was an honor to work with him on several aspects of StarLogo TNG as well as on other class projects throughout the year. Ricarose Roque and Brett Warne made this summer my favorite of my MIT career because of their humor and uncanny ability to accomplish anything they set their minds to. I wish them success in all they do. Evelyn Eastmond, Anna Wendel, and my mom, Donna Wendel, all took the time to carefully edit this paper, providing more pages of feedback than I had a of text. I could not have written this without them. My dad, Tom Wendel, provided several figures and kept me productive when I had no motivation left.

Of course, writing the paper is only a small portion of what goes into a thesis. I would like to thank my friends, especially Lawrie, Matthew, Joyce, Jessica, and Mindy, who kept being nice to me while I neglected to return calls and emails for months on end, and reminded me that life continues with or without the thesis. I would like to thank my girlfriend, Monica, who put up with my incessant stress-outs and fed me yummy dinners while I was working. Finally, I would like to thank my family for the constant love, peace, encouragement, wisdom, and joy that they continue to pour into my life.

12. References

- [1] Begel, A. 1996. *LogoBlocks: A Graphical Programming Language for Interacting with the World*. MIT Advanced Undergraduate Project.
- [2] Gee, J. P. 2003. *What video games have to teach us about learning and literacy?* New York: Palgrave Macmillan.
- [3] Klopfer, E. and A. Begel. 2006. *StarLogo TNG. An Introduction to Game Development*. In Press for the Journal of E-Learning.
- [4] Klopfer, E., A. Begel, K. Wang, C. McCaffrey, and D. Wendel. 2006. *Teaching Game Programming through StarLogo TNG*. Paper presented at the International Conference for the Learning Science 2006.
- [5] Klopfer, E. and S. Yoon. 2005. *Developing Games and Simulations for Today and Tomorrow's Tech Savvy Youth*. Tech Trends. 49(3) 33-41.
- [6] Klopfer, E., V. Colella, and M. Resnick. 2002. *New paths on a StarLogo Adventure*. Computers & Graphics, 26(4) 615-622.
(<http://www.sciencedirect.com/science/article/B6TYG-461XFW7-3/2/272cbf1349ce3d80dc79a8d872c39793>)
- [7] McCaffrey, C. 2006. *StarLogo TNG: The Convergence of Graphical Programming and Text Processing*. Master's thesis, Dept. of Electrical Eng. and Computer Sci., Massachusetts Inst. of Technology.
- [8] Scheintaub, H. 2006. *Programming: A Powerful Physics Learning Tool*. Paper presented at the International Research Group on Physics Teaching (GIREP) Conference 2006.
- [9] Vesgo, J. 2005. *Interest in CS as a Major Drops Among Incoming Freshmen*. Computing Research News, 17(3).
(<http://www.cra.org/CRN/articles/may05/vegso>)
- [10] WarCraft III. Blizzard Entertainment. (<http://www.battle.net/war3/>)

Appendix A: Spaceland data structure files

TerrainData.java

```
package starlogoc;

import java.awt.Color;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.LongBuffer;
import java.util.ArrayList;
import java.util.List;

public class TerrainData {
    /*
     * each patch contains:
     * slnum color; // SL Color of the patch
     * slnum *heap; // pointer to the heap where this patch's variables are stored
     * int texture; // the index of the texture of this patch, -1 for no texture
     * float corner_heights[4]; // NW, SW, SE, NE i.e. UL, LL, LR, UR
     * Tex_coord tex_coords[2]; // NW, SE texture coordinates
     * float x;
     * float y;
     * Patch_side north_side;
     * slnum color; // SL Color of the side
     * int texture; // index of the texture of this side, -1 for no texture
     * float tex_x[2]; // horizontal texture coordinates (y values are calculated by
renderer)
     * Patch_side west_side;
     * slnum color; // SL Color of the side
     * int texture; // index of the texture of this side, -1 for no texture
     * float tex_x[2]; // horizontal texture coordinates (y values are calculated by
renderer)
     */
    /*
     * Note that the following offsets are measured in bytes
     */
    public static final int COLOR_OFFSET = getColorOffset();
    public static final int HEAP_OFFSET = getHeapOffset();
    public static final int TEXTURE_OFFSET = getTextureOffset();
    public static final int CORNER_HEIGHTS_OFFSET = getCornerHeightsOffset();
    public static final int TEX_COORDS_OFFSET = getTexCoordsOffset();
    public static final int NORTH_OFFSET = getNorthSideOffset();
    public static final int WEST_OFFSET = getWestSideOffset();
    public static final int SIDE_COLOR_OFFSET = getSideColorOffset();
    public static final int SIDE_TEXTURE_OFFSET = getSideTextureOffset();
    public static final int SIDE_TEX_X_COORDS_OFFSET = getSideTexXCoordsOffset();
    public static final int PATCH_SIZE = getPatchSizeInBytes();

    public ByteBuffer patches;
    private int height, width;
    private int index;
    private float patchSize;
    private static StarLogo sl;
    private int numPatchesOwn;
    private LongBuffer patchHeap;
    private List<Variable> varList = new ArrayList<Variable>();

    /*
     * Native terrain functions
     */
    private static native int getPatchSizeInBytes();
    private static native int getHeapOffset();
    private static native int getColorOffset();
    private static native int getTextureOffset();
    private static native int getCornerHeightsOffset();
}
```

```

private static native int getTexCoordsOffset();
private static native int getNorthSideOffset();
private static native int getWestSideOffset();

private static native int getSideColorOffset();
private static native int getSideTextureOffset();
private static native int getSideTexXCoordsOffset();

private native void initTerrain(int index, int width, int height, ByteBuffer patches);
private native void setPatchHeap(int index, int numPatchesOwn, LongBuffer patchHeap);
private native void clearPatches0(int index);
private native void scatterPC0(int index, double red, double green, double blue,
double black, int reseed);

public TerrainData(int index, int width, int height, StarLogo thesl) {
    this.index = index; // this terrain's index in the terrains array
    this.height = height;
    this.width = width;
    sl = thesl;
    patchSize = 3; //TODO: let this be variable

    patches = ByteBuffer.
    allocateDirect(width * height * getPatchSizeInBytes()).
    order(ByteOrder.nativeOrder());

    initTerrain(index, width, height, patches);
}

/**
 * <em>Note:</em> this class does not support bounds checking.
 * @return the number of patches
 */
public int getNumPatches() {
    return width * height;
}

public int getHeight() {
    return height;
}

public int getWidth() {
    return width;
}

public float getPatchSize() {
    return patchSize;
}

public void setPatchSize(float newSize) {
    patchSize = newSize;
}

private int getPatch(int patchX, int patchY) {
    return (patchY * width + patchX) * PATCH_SIZE;
}

/**
 * Use this method if you need to interact with starlogoc.Colors directly.
 * @return the StarLogo color number of patch
 */
public double getColorNumber(int patchX, int patchY) {
    return StarLogo.slnumToDouble(patches.getLong(getPatch(patchX,
patchY)+COLOR_OFFSET));
}

/**
 * Use this method if you only need the Color of patch.
 * @return the Color of patch
 */

```

```

public Color getColor(int patchX, int patchY) {
    return Colors.colorarray[(int)(getColorNumber
        (patchX, patchY)*32.0)];
}

private static float[] colorVector = new float[]{0.0f, 0.0f, 0.0f};

public float[] getColorVector(int patchX, int patchY) {
    int index = (int)(getColorNumber(patchX, patchY)*32.0);
    colorVector = Colors.colors[index];
    return colorVector;
}

public void setColor(int patchX, int patchY, double color) {
    patches.putLong(getPatch(patchX, patchY)+COLOR_OFFSET,
        StarLogo.doubleToSlnum(color));
}

public void setColor(int patchX, int patchY, Color color) {
    double colorNumber = Colors.mapColorToStarLogoColor(color.getRed(),
        color.getGreen(),
        color.getBlue());
    if (colorNumber < 0.0 || colorNumber > (Colors.numcolors + 0.0)) {
        System.out.println("Set PC to illegal color: " + colorNumber + ". Substituting
gray 5.");
        new Exception().printStackTrace();
        colorNumber = 5.0;
    }

    setColor(patchX, patchY,
        Colors.mapColorToStarLogoColor(color.getRed(),
            color.getGreen(),
            color.getBlue()));
}

/**
 * @return the height of patch as the average of the corner heights
 */
public float getHeight(int patchX, int patchY) {
    int base = getPatch(patchX, patchY) + CORNER_HEIGHTS_OFFSET;
    patches.position(base);
    float height = 0;
    for (int i = 0; i < 4; i++) {
        height += patches.getFloat();
    }
    height /= 4.0;
    return height;
}

public float getHeight(int patchX, int patchY, int corner)
{
    patches.position(getPatch(patchX, patchY) + CORNER_HEIGHTS_OFFSET + corner * 4);
    return patches.getFloat();
}

public void setHeight(int patchX, int patchY, int corner, float height)
{
    patches.position(getPatch(patchX, patchY) + CORNER_HEIGHTS_OFFSET + corner * 4);
    patches.putFloat(height);
}

public float[] getHeights(int patchX, int patchY) {
    int base = getPatch(patchX, patchY) + CORNER_HEIGHTS_OFFSET;
    patches.position(base);
    float[] heights = {0,0,0,0};
    patches.asFloatBuffer().get(heights);
    return heights;
}

public void getHeights(int patchX, int patchY, float[] heights)

```

```

    {
        int base = getPatch(patchX, patchY) + CORNER_HEIGHTS_OFFSET;
        patches.position(base);
        patches.asFloatBuffer().get(heights);
    }

    public void setHeight(int patchX, int patchY, float height) {
        float[] heights = {height, height, height, height};
        setHeights(patchX, patchY, heights);
    }

    /**
     *
     * @param patchX
     * @param patchY
     * @param heights - the heights of the 4 corners in this order: NW, SW, SE, NE
     */
    public void setHeights(int patchX, int patchY, float[] heights) {
        if (heights.length != 4)
            throw new RuntimeException("Bad array passed to setHeights. 4 elements
expected.");
        int base = getPatch(patchX, patchY) + CORNER_HEIGHTS_OFFSET;
        patches.position(base);
        patches.asFloatBuffer().put(heights, 0, 4);
    }

    public void addHeight(int patchX, int patchY, float height)
    {
        float[] heights = getHeights(patchX, patchY);
        for (int i = 0; i < 4; i++) {
            heights[i] = heights[i] + height;
        }
        setHeights(patchX, patchY, heights);
    }

    public void addHeights(int patchX, int patchY, float[] heights) {
        float[] heights2 = getHeights(patchX, patchY);
        for (int i = 0; i < 4; i++) {
            heights2[i] = heights2[i] + heights[i];
        }
        setHeights(patchX, patchY, heights2);
    }

    /**
     * @return the texture of patch
     */
    public int getTexture(int patchX, int patchY) {
        return patches.getInt(getPatch(patchX, patchY) + TEXTURE_OFFSET);
    }

    public void setTexture(int patchX, int patchY, int texture) {
        patches.putInt(getPatch(patchX, patchY) + TEXTURE_OFFSET, texture);
    }

    /**
     * fills the fields of a TerrainDataPatch
     */
    public void getTerrainDataPatch(int patchX, int patchY, TerrainDataPatch p)
    {
        int base = getPatch(patchX, patchY);
        patches.position(base + COLOR_OFFSET);
        p.color = patches.getLong();
        //skip the heap pointer
        patches.position(base + TEXTURE_OFFSET);
        p.texture = patches.getInt();
        patches.asFloatBuffer().get(p.heights, 0, 4);
        p.heights[0] *= patchSize; // multiply by height scale, since
        p.heights[1] *= patchSize; // these DataPatches are used for
        p.heights[2] *= patchSize; // rendering rather than for interaction
        p.heights[3] *= patchSize; // with agents. */
    }

```



```

    patches.position(base + TEX_COORDS_OFFSET);
    patches.asFloatBuffer().get(p.tex_coords, 0, 4);
    patches.position(base + NORTH_OFFSET);
    p.north_color = patches.getLong();
    p.north_texture = patches.getInt();
    p.north_tx0 = patches.getFloat();
    p.north_tx1 = patches.getFloat();
    p.west_color = patches.getLong();
    p.west_texture = patches.getInt();
    p.west_tx0 = patches.getFloat();
    p.west_tx1 = patches.getFloat();

    // temporary hack to make walls brown =P FIXME
    p.north_color = StarLogo.doubleToSlnum(36);
    p.west_color = StarLogo.doubleToSlnum(36);
}

public void reallocateVariables(List<Variable> newVarList) {
    // Shortcircuit if no changes have been made
    if (varList.equals(newVarList))
        return;

    LongBuffer newPatchHeap = ByteBuffer.
        allocatedDirect(getNumPatches() * newVarList.size() * 8).
        order(ByteOrder.nativeOrder()).
        asLongBuffer();

    // Generate a list that maps new var positions to old positions.
    // -1 signals that this is a new variable that will be initialized to 0.
    List<Integer> positions = new ArrayList<Integer>(newVarList.size()); // Indexes new
positions to old positions
    for (int newPosition = 0; newPosition < newVarList.size(); newPosition++)
        positions.add(new Integer(varList.indexOf(newVarList.get(newPosition))));

    int oldPosition;
    for (int i = 0; i < getNumPatches(); i++) {
        for (int newPosition = 0; newPosition < positions.size(); newPosition++) {
            oldPosition = positions.get(newPosition).intValue();
            if (oldPosition == -1) {
                //System.out.println("new variable at pos: " + newPosition);
                newPatchHeap.put(0);
            }
            else {
                //System.out.println("oldvar: " + oldPosition + " to newvar: " +
newPosition);
                newPatchHeap.put(patchHeap.get(i * numPatchesOwn + oldPosition));
            }
        }
    }

    varList = new ArrayList<Variable>(newVarList);
    numPatchesOwn = newVarList.size();
    patchHeap = newPatchHeap;
    setPatchHeap(index, numPatchesOwn, patchHeap);
}

public void clearPatches() {
    synchronized(sl.getLock()) {
        clearPatches0(index);
    }
}

public void scatterPC(double red, double green, double blue, double black) {
    synchronized(sl.getLock()) {
        scatterPC0(index, red, green, blue, black, sl.seedThisThread());
    }
}
}

```

patch.h

```
#ifndef _patch_h
#define _patch_h

#include "slnum.h"
#include <stdlib.h>

/* Patches Notes:
 * *****
 * Each Patch has a top, which is really the "patch," and two sides -
 * north_side and west_side. These should never be rendered on the
 * edge of the terrain. Otherwise, they're rendered between patches
 * whose corner heights do not match for shared corners.
 *
 * The "height" of a patch is equal to the average of the four corner
 * heights. This should also be rendered as a vertex in the center
 * of the patch, but it is not stored since it is a calculated value.
 * *****/
typedef struct patch_side {
    slnum color; // SL Color of the side
    int texture; // index of the texture of this side, -1 for no texture
    float tex_x[2]; // horizontal texture coordinates (y values are calculated by
renderer)
} Patch_side;
#define PATCHSIDE_LENGTH (sizeof(Patch_side)) // =24 on PPC Mac

typedef struct tex_coord {
    float x;
    float y;
} Tex_coord;

typedef struct patch {
    slnum color; // SL Color of the patch
    slnum *heap; // pointer to the heap where this patch's variables are stored
    int texture; // the index of the texture of this patch, -1 for no texture
    float corner_heights[4]; // NW, SW, SE, NE i.e. UL, LL, LR, UR
    Tex_coord tex_coords[2]; // NW, SE texture coordinates
    Patch_side north_side;
    Patch_side west_side;
} Patch;
#define PATCH_LENGTH (sizeof(Patch)) // =96 on PPC Mac

typedef struct terrain {
    int width; // the width of the terrain in patches
    int height; // the height/depth/whatever you call it of the terrain
    Patch *patches; // the patches in this terrain
    /*
    Space here to add other saved state for the terrain, such as agent
    starting positions, custom texture info, level name, skybox texture,
    etc., etc. These will come soon hopefully!!
    */
} Terrain;

typedef struct mod_patch {
    short x;
    short y;
} Modified_patch;

#define MAX_PATCH_TERRAINS 16

/* Patches heap is an array of slnums, allocated by the compiler
when it informs StarLogo of how many patches-own variables
there are. */
extern int num_patches_own;

extern Terrain *current_terrain;
extern Terrain **terrains;
extern Modified_patch *changed_patches;
extern int num_changed_patches;
```

```

extern int max_changed_patches;
extern int patch_colors_shownp;

extern int changedPatchHeight, changedPatchColor,
    changedPatchTexture; // from vm.h
extern int screen_width, screen_height, cell_size;
extern int screen_half_width, screen_half_height;

#ifdef WIN32
#define min(x, y) (((x) < (y)) ? (x) : (y))
#define max(x, y) (((x) > (y)) ? (x) : (y))
#endif

// returns pointer to the patch, or NULL if out of bounds. Maybe will
// change this later to make it faster, but hopefully for now will cause
// NPE's if an algorithm is wrong and tries to go out of bounds
#define GET_PATCH_AT(x, y) ((x >= 0 && y >= 0 && x < current_terrain->width && y <
current_terrain->height) ? &(current_terrain->patches[(y)*current_terrain->width + (x)])
: NULL)

INLINE slnum patch_heap_fun(int x, int y, int index) {
    Patch *p = GET_PATCH_AT(x, y);
    return p->heap[index];
}
#define PATCH_HEAP_MEMBER(x, y, index) patch_heap_fun(x, y, index)

INLINE void set_patch_heap_fun(int x, int y, int index, slnum val) {
    Patch *p = GET_PATCH_AT(x, y);
    p->heap[index] = val;
}
#define SETPATCH_HEAP_MEMBER(x, y, index, val) set_patch_heap_fun(x, y, index, val)

// from vm.h
INLINE slnum logoToInternalXcor(slnum logo_xcor) {
    return FLOAT_TO_SLNUM(screen_half_width + SLNUM_TO_FLOAT(logo_xcor));
}

INLINE slnum logoToInternalYcor(slnum logo_ycor) {
    return FLOAT_TO_SLNUM(screen_half_height - SLNUM_TO_FLOAT(logo_ycor));
}

INLINE slnum internalToLogoXcor(slnum internal_xcor) {
    return FLOAT_TO_SLNUM(SLNUM_TO_FLOAT(internal_xcor) - screen_half_width);
}

INLINE slnum internalToLogoYcor(slnum internal_ycor) {
    return FLOAT_TO_SLNUM(screen_half_height - SLNUM_TO_FLOAT(internal_ycor));
}

void set_modified(int x, int y);
void clear_patches_i(int index);
void clear_patches();
void set_current_terrain(int num);
void scatter_patches_i(int index, double red, double green, double blue, double black,
int reseed);
void scatter_patches(double red, double green, double blue, double black, int reseed);
void init_patches(int width, int height, int cellsize);
void set_num_patches_own(int num, slnum *new_patch_heap);
void set_patch_heap(int index, int num, slnum *new_patch_heap);
int patchat(double x, double y);
int patchtowards(double angle, double magnitude, double xcor, double ycor);
void set_patch_terrain(int num);

slnum get_patch_height(int x, int y); //used in turtle.c, from vm.h

// from vm.c
void stamp(int x, int y, slnum color);
void set_patch_height(int x, int y, double height);
void change_patch_height_smooth(int x, int y, double delta_height);

```

```
void change_patch_height(int x, int y, double delta_height);
slnum get_patch_texture(int x, int y);
void set_patch_texture(int x, int y, slnum texture);
slnum get_pc(int x, int y);
```

```
#endif // _patch_h
```

Appendix B: Editor features to fulfill requirements

General

- Height slider improved (w/ label)
- Grid location displayed (even if over Spaceland, eventually)
- Multiple terrain blocks, “active” one can be swapped in.
 - Always edit active, so changes visible (eventually edit IN SpaceLand)
 - Can store to a block or load from one
- Views (all zoomable, scrollable)
 - Height-only
 - Colored height (also includes texture)
 - Color only
 - Texture only

Tools

- Raise/Lower area
- Wall/Trench
- Mound/Crater
- Level area
- Erase (sets height to 0 and reset to green for now)
- Paint Shapes
- Change Color (where color is color + texture)
 - Includes color/texture editor
- Wall painter
 - Choose color, texture (different palette of textures from patch tops)
 - Choose which wall(s) of 4 possible to affect
 - Choose – edges, or all internal walls too?
- Place/move agents (one-by-one, at mouse pointer, locations stored in sub-structure under the terrain structure)
 - Agents under mouse highlight
 - Delete deletes them, dragging moves them.
 - Hover long enough, and an agent monitor pops up to change properties.
- Import heightmap
- Import colormap

Meta-tools

- New level (and set dimensions there)
- Save to block/load from block
- Undo/redo (based on log)
- Save to file/load from file (ALL terrains go to project save file already)

For future versions

- Add agents to the file format
- Add skybox to format
- Add tools for changing agent properties when placing them
- Extend agent adding tools – density scatter/formations/etc.
- Where useful, add overlay so editing works on terrain in Spaceland as well!
- (Goal is direct manipulation on Spaceland)
- Keep evaluating and iterating design!