Custom and Interactive Environments in StarLogo Nova for Computational Modeling

by Malcolm X. Wetzstein

B.S., Computer Science and Engineering,
Massachusetts Institute of Technology (2019)

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 2019

© Massachusetts Institute of Technology 2019. All rights reserved.

Author: ____

Department of Electrical Engineering and Computer Science
August 23rd, 2019

Certified by: ____

Eric Klopfer, Professor, Thesis Supervisor
August 23rd, 2019

Accepted by: ____

Katrina LaCurts, Chair, Master of Engineering Thesis Committee

Custom and Interactive Environments in StarLogo Nova for Computational Modeling

by Malcolm X. Wetzstein

August 23rd, 2019

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

Abstract

StarLogo Nova is an agent-based simulation and game programming application geared towards classroom learning. In many cases, *StarLogo Nova* can be an effective computational modeling tool, allowing students to apply their knowledge and gain a greater understanding of scientific concepts. However, there are many computational models that are difficult to implement in *StarLogo Nova* or cannot be implemented. This can be partly attributed to the fact that agents are the only programmable and customizable entity in *StarLogo Nova* and are limited to interactions with other agents. We expand the set of computational models possible within *StarLogo Nova* by adding new features and capabilities; namely an editable and programmable 3D environment for agents to interact with. We add the capability for users to program real-time interactions between agents and their environment. A dynamic and custom environment opens up the possibility for new computational models and simplifies the implementation of others.

Thesis Supervisor: Eric Klopfer
Title: Professor

Table of Contents

1 Introduction 8

1.1 Purpose of *StarLogo Nova* 9

1.2 What is *StarLogo Nova*? 9

1.2.1 Agent-Based Simulation 11

1.2.2 Block-Based Programming 11

1.3 Need for Additional Functionality 13

1.3.1 Computational Models That Are Difficult to Implement in *StarLogo Nova* 13

1.3.2 Solutions in Previous Iterations of *StarLogo* 15

2 Environment Representation 16

2.1 Terrain 17

2.2 Patches 19

2.2.1 Floor Patches 19

2.2.2 Wall Patches 21

2.3 Commands 22

2.3.1 Undo/Redo 22

3 Terrain Editor 23

3.1 User Interface 23

3.1.1 Selection 23

3.2 Tools and Terrain Modification 24

3.2.1 Raise 25

3.2.2 Mound 26

3.2.3 Level 29

3.2.4 Close Seams 30

3.2.5 Ramp 31

3.2.6 Copy 33

3.2.7 Move 33

3.2.8 Stretch 34

3.2.9 Grow 35

4 Agent-Terrain Interaction 36

4.1 Terrain Traversal 37

4.2 Terrain Modification 39

4.2.1 Build / Dig 40

4.2.2 Yank / Stomp Grid 40
4.2.3 Yank / Stomp 42
4.3 Patch Data 43
4.3.1 Get / Set Attribute 43
4.3.2 Interpolate Attribute 44
5 Software Design and Implementation Details 44
5.1 Terrain Class 44
5.1.1 Methods and Interface 45
5.1.2 Commands and the Undo/Redo System 51
5.2 3D Visualization 52
5.3 Patch Class 54
5.3.1 Methods and Interface 56
5.3.2 Patch Data Structure 59
5.4 Picking System 60
6 An Erosion Model Using Our New Features 61
7 Conclusion 62
8 Future Work 62
8.1 User Interface 63
8.2 Agent Movement 64
8.3 Additional Features 64
9 Acknowledgements 66
10 References 66
Appendix: Terrain Editor Keyboard and Mouse Inputs 66
General 66
Tools 67
Special 68
Camera 68

List of Figures

Figure 1: Top: StarLogo Nova viewport and workspace. 10
Figure 2: A program in StarLogo Nova described graphically using blocks. 12
Figure 3: Editable terrain from StarLogo Nova's predecessor, StarLogo TNG. 16
Figure 4: Previous terrain in StarLogo Nova. 17
Figure 5: An example of a triangle mesh. 18
Figure 6: Location of points in a floor patch. 21
Figure 7: Wall patches. 22
Figure 8: Selection in the Terrain Editor. 24
Figure 9: The raise command. 25
Figure 10: The mound command. 26
Figure 11: The level command. 29
Figure 12: The close seams command. 30
Figure 13: The ramp command. 31
Figure 14: Diagram of ramp algorithm for computing linear transitions. 32
Figure 15: Agent terrain traversal and altitude. 38
Figure 16: Definition of terrain height. 39
Figure 17: The yank grid command. 41
Figure 18: Template for yank grid weights. 42
Figure 19: Correspondence between floor patch points and underlying triangle vertices. 55
Figure 20: Diagram of data structure for Patch class instances. 60

1 Introduction

StarLogo Nova is an agent-based simulation and game programming application geared towards classroom learning. In many cases, *StarLogo Nova* can be an effective computational modeling tool, allowing students to apply their knowledge and gain a greater understanding of scientific concepts. However, there are many computational models that are difficult to implement in *StarLogo Nova* or cannot be implemented. This can be partly attributed to the fact that agents are the only programmable and customizable entity in *StarLogo Nova* and are limited to interactions with other agents. We expand the set of computational models possible within *StarLogo Nova* by adding new features and capabilities; namely an editable and programmable 3D environment for agents to interact with. We add the capability for users to program real-time interactions between agents and their environment. A dynamic and custom environment opens up the possibility for new computational models and simplifies the implementation of others.

Section 1 provides more background information on what the *StarLogo Nova* application is, the intended purposes of the application, and how it is used. Section 2 elaborates on *StarLogo Nova's* previous representation of the environment and the role that it plays in a simulation. More importantly, Section 2 explains how the work of this thesis changes the representation of the environment so that new features can be added. Sections 3 and 4 explain the new features added by the work of this thesis, with Section 3 focusing on the new features for editing the environment and Section 4 focusing on the new features that allow programmable interactions between agents and the environment. Section 5 sheds light on how the software that implements the new features is organized and explains the interface that it exposes to the rest of *StarLogo Nova* that make the new features possible. Section 5 also explains some key implementation details. Section 6 provides an example of a concrete use case for the new features we propose. Section 7 summarizes how the new features fulfill the purpose of this

thesis. Section 8 provides several starting points for future work to either extend or improve the work of this thesis.

1.1 Purpose of *StarLogo Nova*

StarLogo Nova is a flexible tool that can be used to create many kinds of simulations, including games. However, *StarLogo Nova* is primarily a tool for creating computational models. A computational model is a computer simulation used to study the behavior of a system. The design of *StarLogo Nova* is particularly geared towards creating models of complex systems, systems with many components that interact with one another to produce complex behavior [Melnik 2015]. An example of a complex system is the relationship between populations of a predator species and their prey. The relationship between the two populations can be modeled in a computer simulation. The computer simulation can then be studied to better understand how the size of one population affects the other.

The target audiences of *StarLogo Nova* are secondary school students and educators. As stated previously, *StarLogo Nova* is a powerful tool for teaching scientific concepts to students. By having students create computational models of a scientific phenomenon, they gain greater understanding of that phenomenon and dispel misconceptions. Working in *StarLogo Nova* also allows students to start developing computational thinking skills, the ability to express problems in a way that computers can solve, earlier in their education [Lee et al. 2014].

1.2 What is *StarLogo Nova*?

In this section we explain the nature of the *StarLogo Nova* application; what platform the application is on, the user interface of the application, and how a user uses the application. *StarLogo Nova* is a web-based application that can be reached at slnova.org. It is used to program 3D simulations. After a user creates an account on *StarLogo Nova*, they can create a 3D simulation by making a project. Once a project is opened, a user interface containing a viewport and a workspace is presented to the

user [Figure 1]. These are the two main components of *StarLogo Nova* that a user engages with.

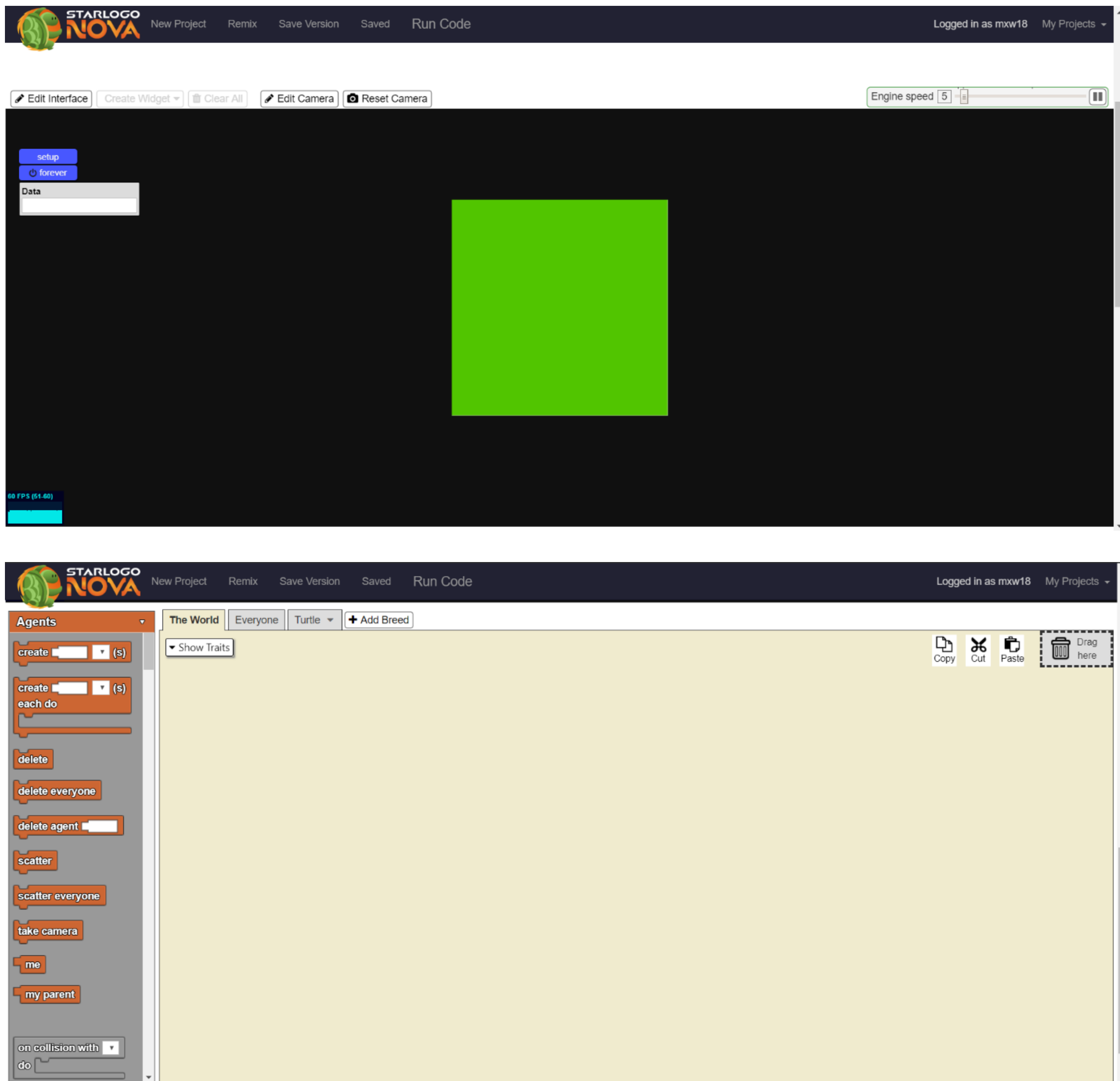

Figure 1: Top: StarLogo Nova viewport. Bottom: StarLogo Nova workspace.

The viewport is a window showing a 3D visualization of the current state of the simulation. The 3D visualization itself is called Spaceland. The viewport has several tools for interacting with Spaceland and providing user input to the underlying simulation. Among these tools are camera controls to manipulate the view of Spaceland and explore the 3D environment of the simulation.

The workspace is the area where a user programs the behavior of their simulation. Here, a program that defines the behavior of the simulation is constructed using blocks, components used to produce a graphical representation of a program, which we discuss further in Section 1.1.3. Users can compile and execute this program by pressing the "run code" button at the top of the web-page. Once the program in the workspace has been compiled and is running, the user can interact with the simulation through the viewport.

To learn more about how to use the *StarLogo Nova* application, we suggest that the reader explore the resources page of the slnova.org website.

1.2.1 Agent-Based Simulation

Simulations in *StarLogo Nova* are agent-based. Agents are entities that possess several traits, including a position in 3D space. Users can add custom traits to an agent in addition to the existing traits built into *StarLogo Nova*. Agents are organized into breeds; a breed is similar to classes in other programming languages. When a user creates a program in *StarLogo Nova*, they program the behavior of each breed, so that all agents of the same breed follow the same behavior. Users can create as many breeds as they desire, as well as any number of agents from each breed. The behavior of a breed can be programmed to alter the traits of other agents upon certain events, allowing agents to interact with one another. Breeds can even be programmed to create or destroy other agents upon certain events as well. The agent-based design of *StarLogo Nova* is what makes it well suited for computational modeling of complex systems. Agents are the individual components that make up a complex system.

1.2.2 Block-Based Programming

StarLogo Nova does not make use of a written programming language or syntax. Instead, programs are designed using a graphical representation in the workspace. The elements used to construct this graphical representation are called blocks. Control statements, intrinsic functions,

operators, event callbacks, and other elements commonly found in written programming languages are instead represented by individual blocks. A syntax exists dictating how blocks can be connected together or socketed inside one another in order to define a procedure [Figure 2]. The procedures constructed from blocks determine the behavior of agents, instantiate agents, and can respond to user input from the viewport, among other things. For more information on what blocks are available in *StarLogo Nova* and what they can do, we suggest that the reader visit the resources page of the slnova.org website.

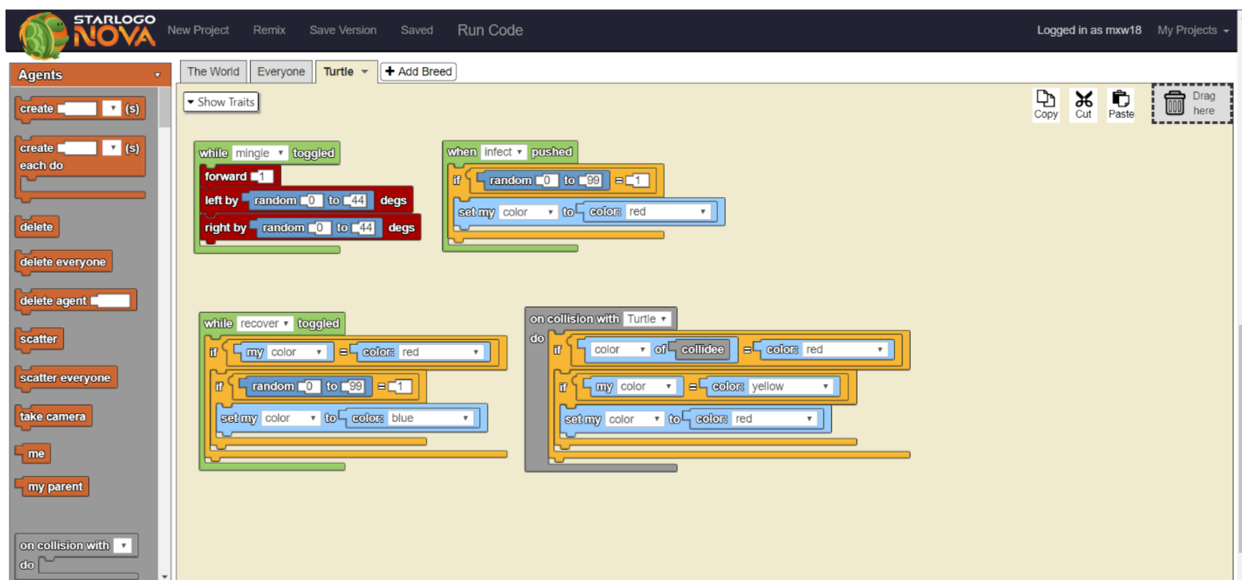

Figure 2: A program in StarLogo Nova described graphically using blocks.

Using a block-based programming language gives *StarLogo Nova* several advantages over using a written programming language that are ideal for its intended target audience. Students using *StarLogo Nova* in secondary school are novices when it comes to computer programming. Novices learning to programming in a written programming language have to learn both the general concepts behind designing computer programs, such as computational thinking skills, and the syntax of the new language they are using to program. Having to remember the proper syntax used to write code leaves novices open to make mistakes that prevent their code from even compiling. With a block-based language, the blocks provide a graphical representation of the language's syntax and enforce that syntax when a user

tries to connect blocks to form a program. Syntax errors cannot be made in block-based languages, and the user must recognize the syntax of the language rather than memorize it, which increases the learnability of the language by allowing novices to focus on learning programming concepts [Bau et al. 2017]. This increased learnability makes *StarLogo Nova* easier for novices to approach and allows them to spend more time learning computational thinking skills rather than the syntax of a particular language, making *StarLogo Nova* more ideal for use in classroom settings.

1.3 Need for Additional Functionality

Although the agent-based design of *StarLogo Nova* is well suited for creating computational models, there are still limitations to the kinds of computational models that can be created. The agent-based design alone is not enough to implement models of some complex systems, may not provide enough capabilities to produce accurate models, or does not provide enough capabilities to make the implementation of the models feasible for secondary school students.

1.3.1 Computational Models That Are Difficult to Implement in *StarLogo Nova*

A thorough analysis of what kinds of complex systems are difficult to model in *StarLogo Nova* could be the research topic of an entire thesis, so we only discuss the kinds of complex systems that are difficult to model in *StarLogo Nova* that the work of this thesis makes possible to model with relative ease.

A complex system can be modeled in more than one way, depending upon what simplifications are made. Usually models of real-world complex systems must make simplifications because, in many cases, the components of a complex system can itself be seen as complex systems. For example, the predator-prey model discussed in Section 1.1.1 uses individual predator and prey animals as the components of the complex system. However, an animal itself is a complex system comprised of cells,

and each cell is a complex system of biomolecules, and so forth. To say individual animals are the basic components of a predator-prey complex system is a simplification of the model.

A common case where simplifications are necessary are complex systems that involve modeling large environments. For example, if we want to model the erosion of a mountain due to runoff, modeling the mountain with individual dirt particles as the components of the system would not be feasible. The choice of simplification is not obvious however. The mountain must be broken down into simpler components of some kind. The common approach to making a simplification of an environment is to section the environment into discrete parts that become basic components of the system. The mountain's surface can be sectioned into separate parts in our example, and erosion can take place at individual parts of the surface, with neighboring parts possibly influencing one another. A basic component that can be represented by an agent, for example, a rain drop that becomes runoff, can now interact with the simplified model of the mountain, and the mountain can update the section of surface area where the interaction took place. A solely agent-based design has difficultly implementing this kind of model. Using individual agents to model the sections of the mountain's surface would be difficult because the surface agent would not only have to know how to interact with agents modeling the runoff, but also which agents are its neighboring surface sections which it influences. There is no straightforward way in an agent-based design to program an agent to interact with specific instances of other agents. If a user wants to make changes to the environment in a model to change the starting conditions of their simulation, this is also hard to do if the environment is comprised of agents, as a procedure must be created with blocks in order to produce a new arrangement of agents. Also, computational models in *StarLogo Nova* are visualized by depicting individual agents in 3D space, so there is also the question of how an environment such as the mountain can be properly visualized by a collection of individual agents. An adequate visualization might prove harder to implement than the model itself, especially for a secondary school student with limited knowledge and experience. A

different representation than agents must also be available to model large environments, one that is comprised of components that are spatially aware of one another, is configurable by the user, and has a visualization that is already suitable for its purpose.

A set of features to expand *StarLogo Nova* to support computational models involving large environments should provide an environment representation capable of interacting with agents in in a generalized way that can be used to model a wide variety of complex systems. It should also provide an adequate visualization of the environment, and should allow a user to configure the environment to meet the needs of their model and to change starting conditions. We will demonstrate how the work of this thesis meets these requirements revisiting the erosion complex system later in Section 6, showing how the new features we present in this thesis can be used to create several models for the complex system that can be implemented with ease.

1.3.2 Solutions in Previous Iterations of *StarLogo*

The predecessor to *StarLogo Nova* is *StarLogo TNG*, a desktop application. *StarLogo Nova* is very similar to *StarLogo TNG* and brings much of the same features to a web-based platform. *StarLogo TNG* addressed the issue of computational models that require a different representation of the environment aside from agents. In *StarLogo TNG*, an object called the Terrain represents the ground and is sub-divided into square sections called patches. These patches could be manipulated by both agents through blocks and by the user through an editor in order to change the shape of the Terrain [Figure 3]. The work of this thesis used the Terrain representation in *StarLogo TNG* as an inspiration and starting point to develop the Terrain representation that is now a part of *StarLogo Nova*. For those familiar with *StarLogo TNG*, there are several features contributed by the work of this thesis that were not present in *StarLogo TNG*. The purpose of this thesis is not necessarily related to the goals that the designers of *StarLogo TNG* had, and so many design choices are made differently in the work of this thesis regarding

the Terrain. That being said, the work of this thesis does bring *StarLogo Nova* closer to the wider set of features and capabilities that its predecessor, *StarLogo TNG*, had.

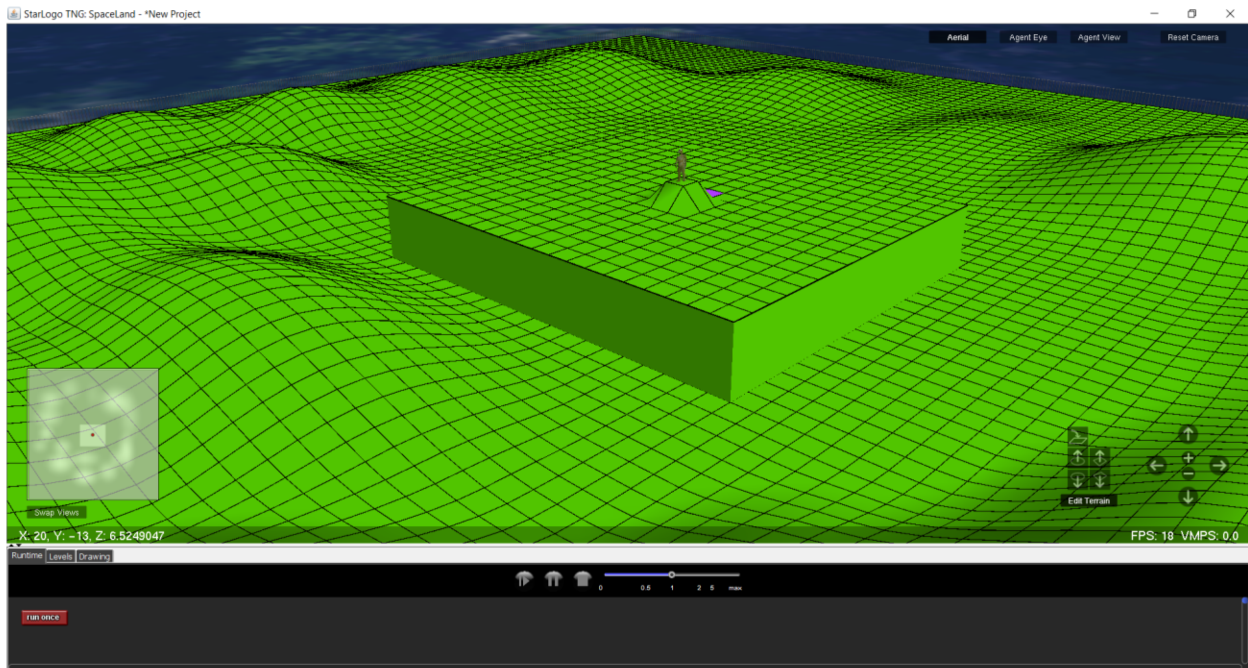

Figure 3: Editable terrain from StarLogo Nova's predecessor, StarLogo TNG.

2 Environment Representation

The environment in which *StarLogo Nova* simulations take place is a 3-dimensionsal space called The World. The World is a rectangular region bounded on six sides; agents in a simulation stay within this bounded region at all times. The positive z-axis is the up direction, with movement in the x and y axes being sideways. The World possesses a standard unit of measure, which we refer to as the world unit in later sections.

Aside from agents, The World contains one special object called the Terrain. Previously in *StarLogo Nova*, the Terrain was a flat plane that served as a visualization for ground level [Figure 4]. Agents at height z equals zero could be thought of as walking along the Terrain at ground level as they

move in the xy-plane. Agents could interact with the Terrain by drawing on it, changing its color locally underneath the agent.

The work of this thesis is to expand upon the functionality of the Terrain, making the Terrain a feature of the environment that agents can interact with and in turn can interact with the agents. Users should also be able to make changes to the Terrain themselves in order to setup a unique environment for agents to interact with that can serve their own purposes.

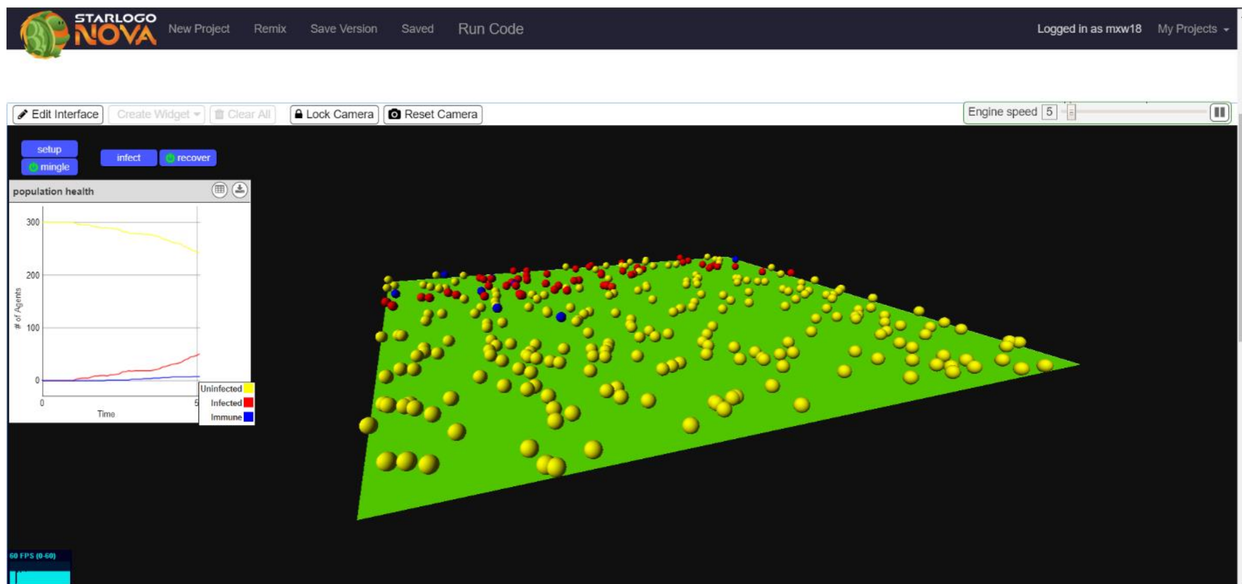

Figure 4: Previous terrain in StarLogo Nova; a simple visualization of the ground that has no impact on the simulation.

2.1 Terrain

The Terrain is an object within The World that serves as a visualization for ground level in *StarLogo Nova* simulations. However, ground level is no longer restricted to the z equals zero plane, as it was in previous versions of *StarLogo Nova*. Instead of a plane, the Terrain is now visualized by a triangle mesh.

A triangle mesh is an open or closed geometric surface approximated by a collection of triangles. These triangles are connected by sharing edges; an arbitrary shape is assembled from triangles by

connecting them in this manner [Figure 5]. For a more detailed explanation of triangle meshes, we refer the reader to [Foley 1994]. By using a triangle mesh, the terrain can be morphed from a plane into an arbitrary surface, allowing the height of the terrain at any point to vary in the z-dimension. Ground level can therefore change locally as an agent moves over the Terrain. The shape of the Terrain is manipulated by manipulating the geometry of the individual triangles that comprise the Terrain, which can be done either by an agent or a user.

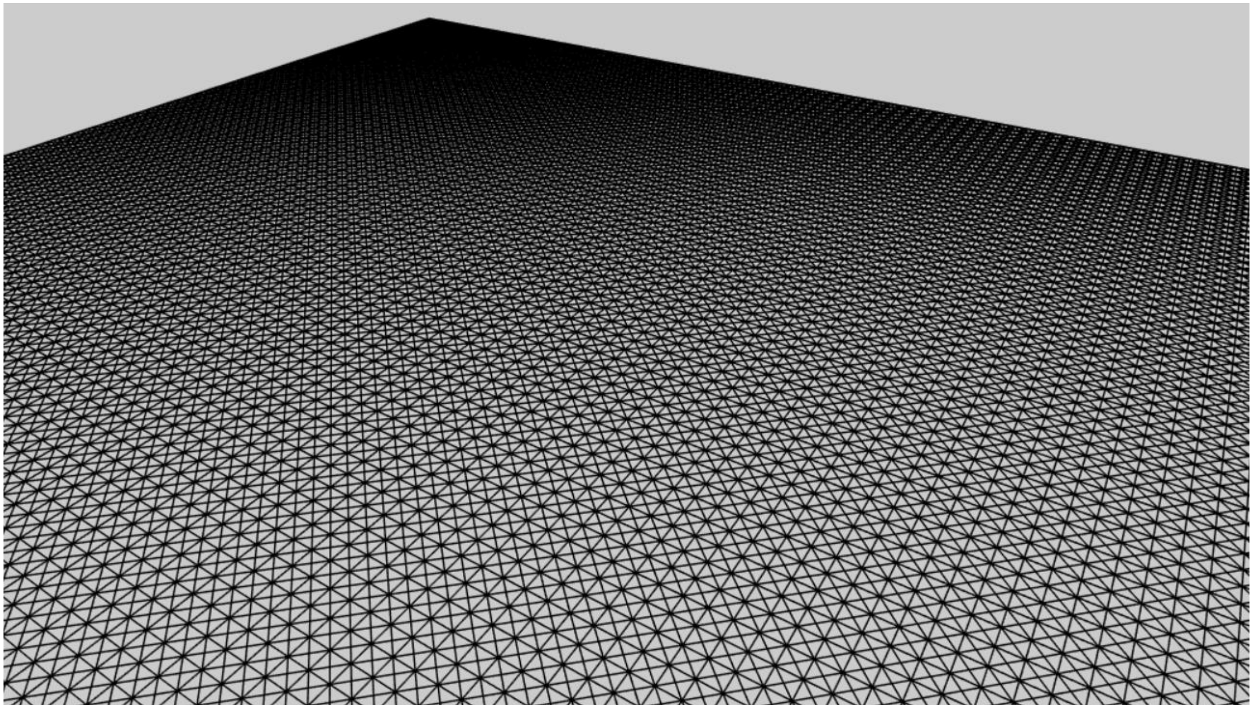

Figure 5: An example of a triangle mesh.

The Terrain now also serves as a spatial data store. Numerical data can be stored at discrete locations spread over the Terrain in the pattern of a regular grid. Data is stored under a name in a lookup table, allowing more than one numerical value to be stored at each grid point.

The Terrain can now be manipulated by agents in two ways, changing the geometry or shape by editing the underlying triangles in the triangle mesh, and writing data to the Terrain at the grid points. The Terrain in turn can manipulate the agents; agents walking along ground level follow the height of

the Terrain. Changing the height of the Terrain therefore affects how agents move about The World. Agents may also be programmed to act differently depending upon what data they read from the grid points on the Terrain, so the Terrain's data store can directly affect the behavior of the agents. These features provide a mechanism for agents and their environment to interact with and influence one another.

2.2 Patches

Users and agents don't interact with the entire Terrain. Instead, they interact with a local region. Patches are an organizational structure for subdividing the Terrain into discrete units that can be manipulated by a user or agent. When an agent interacts with the Terrain, they only interact with a single patch. Similarly, when a user makes edits to the Terrain, they select patches to edit, with a single patch being the smallest element they can edit. Patches also serve as the grid points at which data is stored; each patch has a data store that an agent can read from and write to. Triangles within the Terrain's triangle mesh are arranged such that they are contained within a unique patch. Triangles are assigned to the patch that they are contained in. When a patch's geometry is manipulated, it is responsible for updating its triangles accordingly. Section 5 discusses more the relationship between patches and their underlying triangles, but for now we can think of patches as being the objects that are geometrically manipulated by agents and users.

The patches which the Terrain is subdivided into can be distinguished into one of two types, floor patches and wall patches. We discuss the difference between these two types in the following subsections.

2.2.1 Floor Patches

Floor patches are the patches that agents walk over when they traverse the Terrain. Any patch that has a visible cross section when looking down the z-axis is a floor patch. Floor patches are directly

manipulated by the user or by agents. Agents can read data from and store data at a floor patch. Both users and agents can directly manipulate the geometry of a floor patch as well.

The Terrain is divided into floor patches such that when the Terrain is planar, the floor patches form a square grid. Each floor patch has four corner points and a center point. These five points serve as handles to manipulate the geometry of the floor patch [Figure 6]. To maintain an aligned grid of floor patches and preserve the location of the center of each floor patch, the corner and center points of each floor patch remain fixed in the x and y dimensions. This helps to preserve the boundaries of the The World, which typically match up with the outer edges of the Terrain. This also helps to maintain a sense of distance and makes floor patches a reliable unit of measure. Moving 20 patches left and then 20 patches right should bring you back to the same location, even if the Terrain changed shape during the movement. The center point of a floor patch also determines the spatial location of its data store from the perspective of the agents. Data stored at a particular location should not migrate to a new location due to Terrain changing shape. When a user or agent manipulates the geometry of a floor patch, only the z dimension is affected. By changing the z coordinate of the corner and the center points of one or more floor patches, the Terrain can be molded into a wide variety of shapes.

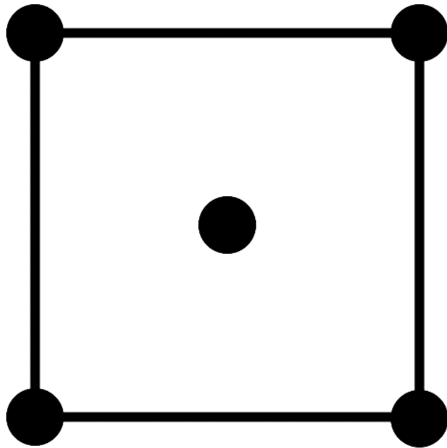

Figure 6: The location of the five points on a floor patch that serve as handles for manipulating the underlying triangle mesh.

2.2.2 Wall Patches

Manipulating the geometry of floor patches can result in discontinuities in the Terrain that appear as holes when viewed from the side. These discontinuities occur when a floor patch is manipulated in such a way that one of its edges does not match up with the edge of an adjacent floor patch. A second type of patch exists to cover up these holes called wall patches. Wall patches cannot be directly manipulated by the user or by agents. They also do not possess the data stores that floor patches have. Instead, wall patches are manipulated indirectly. Wall patches automatically update to cover holes created by discontinuities and are hidden when discontinuities are not present [Figure 7]. Because the corners of each floor patch are fixed in the x and y dimensions, wall patches, as well as the discontinuities that they cover, do not have a visible cross section when looking down the z-axis. Therefore, any patch that does not have a visible cross section when looking down the z-axis is a wall patch. There is no interaction between agents and wall patches. Wall patches exist only for the purpose

of visualization so that the Terrain appears to be one solid entity without gaps.

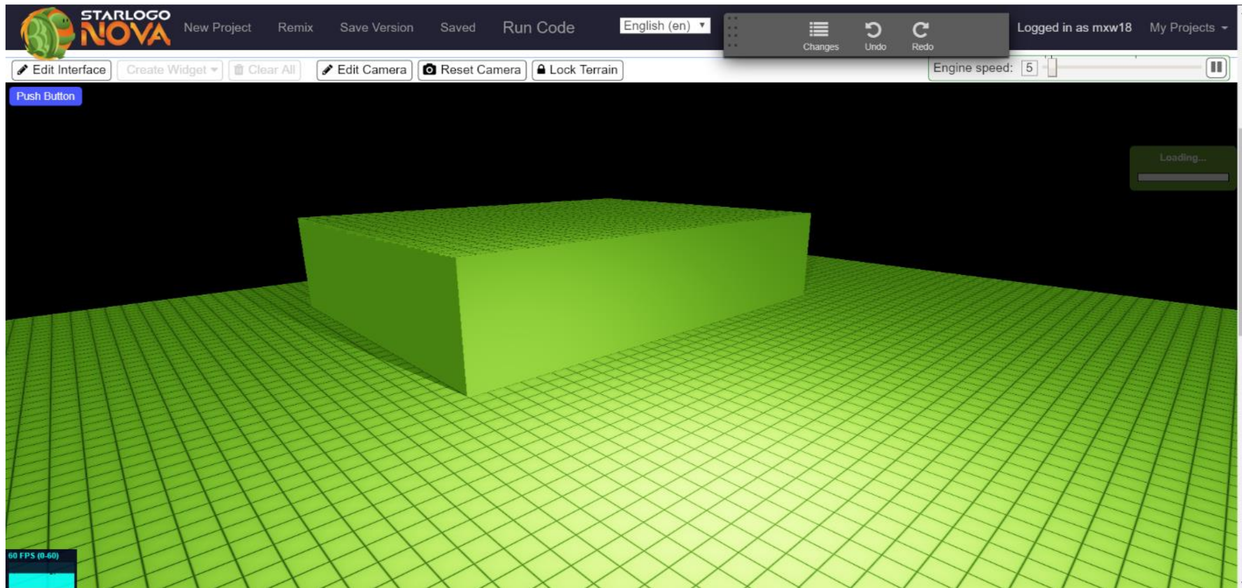

Figure 7: The vertical sections are wall patches that adjust in order to cover up discontinuities that arise when the edges of neighboring floor patches do not agree.

2.3 Commands

Patches are manipulated by agents and by the user via commands. The geometric properties of patches (the underlying triangles) are not directly exposed to users or to agents. Instead, a set of commands exist that manipulate the geometry of one or more patches according to a procedure that acts on the corner and center points of floor patches. Individually, these commands make simple changes to the geometry of the Terrain, but when used together can result in complex results and can be a very expressive tool. Section 3 discusses the commands that are available to the user and Section 4 discusses the commands available to agents.

2.3.1 Undo/Redo

Most commands are recorded into a log of commands. By referencing this log, commands that manipulate the Terrain can be undone and redone via two special commands called undo and redo.

These function similarly to the undo and redo as found in many other software applications that provide an editor interface.

3 Terrain Editor

The Terrain Editor is a new system within *StarLogo Nova* that exposes a set of commands to the user that affect the geometry of the Terrain. The Terrain Editor is available to the user both during and outside of simulation, so that the user can make changes to the Terrain ahead of time or on the fly. In this section we discuss the user interface of the Terrain Editor and describe the commands that it exposes.

3.1 User Interface

Here we only explain the key features of the user interface. Additional minor features can be found in the Appendix. The Terrain Editor can be activated in *StarLogo Nova* by pressing the "Edit Terrain" button located above Spaceland. Clicking "Lock Terrain" deactivates the Terrain Editor. When the Terrain Editor is active, black gridlines appear over the Terrain. These gridlines show the boundaries between individual floor patches. The Terrain Editor makes use of *StarLogo Nova's* current camera control system in order to move around the user's perspective and view the Terrain from different angles, which can be helpful during editing. The camera controls can be found in the Appendix.

3.1.1 Selection

Commands can only be performed by the user on floor patches that are currently selected. Selecting floor patches is done using the left mouse button. By clicking with the left mouse button on an individual floor patch and then dragging, the user can select a rectangular region of floor patches. Releasing the left mouse button completes the selection. Selected patches appear highlighted in purple. The next command that is performed by the user will operate on the floor patches highlighted purple

within the selection. The selected region can extend past the edges of the Terrain, in which case virtual patches within the selection appear in cyan [Figure 8]. How selecting virtual patches affects the command done by the user depends on which command was used.

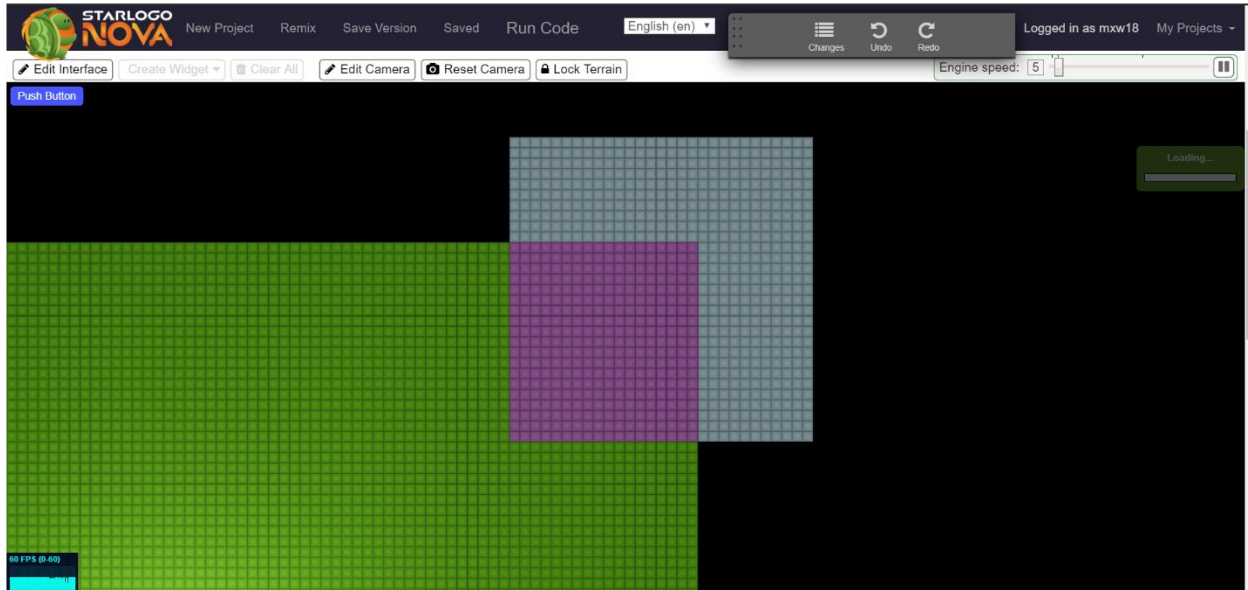

Figure 8: A selection performed by the user in the terrain editor. Cyan patches are "virtual patches" outside of the terrain bounds.

3.2 Tools and Terrain Modification

Tools in the Terrain Editor allow the user to execute commands on the floor patches that they have selected. To distinguish between a tool and a command, a command is a procedure that operates on floor patches. A tool invokes a command with the proper arguments based on the user's input. When a command takes an argument for elevating the height of the Terrain, tools provide those arguments in World units. In the following subsections, we give a qualitative description of the effect of each command available to the user and give a high-level description of the algorithm used to provide that effect. We then explain the user interface controls of the corresponding tool that is used to execute the command. Tools make use of the mouse wheel and right mouse button, leaving the left mouse button available to use for selection.

3.2.1 Raise

Raise is a command that elevates every floor patch within a selection. Features of the Terrain within the selection are preserved, and a discontinuity is created around the edges of the selection that results in the formation of wall patches [Figure 9].

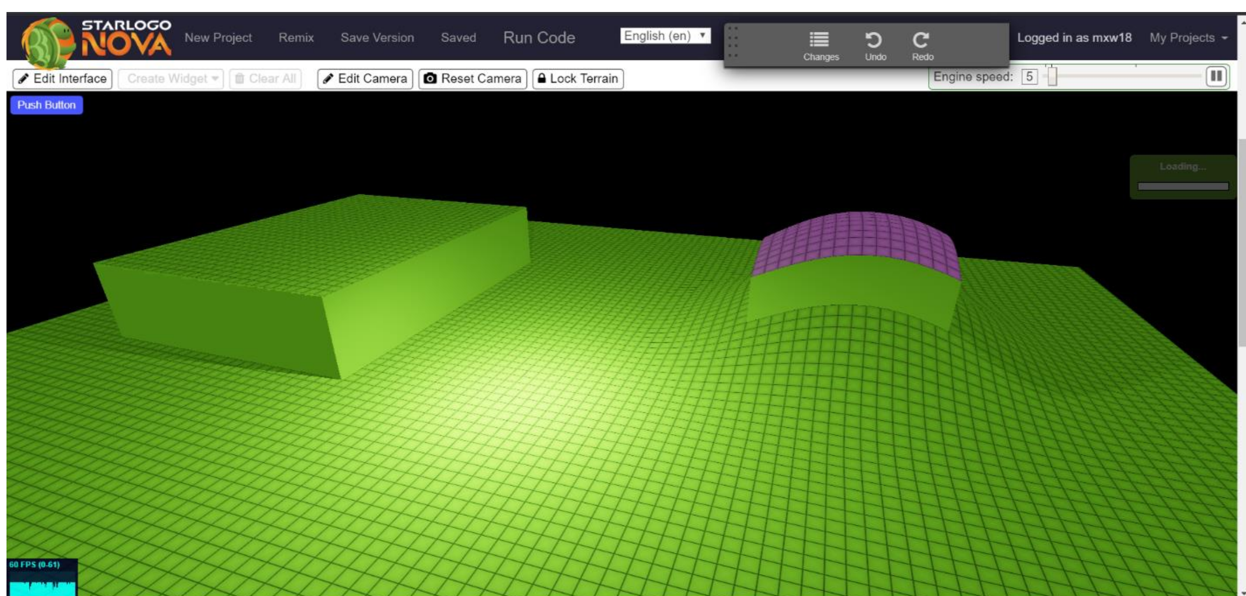

Figure 9: The raise command performed on both a flat and a curved region.

The algorithm for raise is simple. Raise iterates over each floor patch and increments the height of each point in the patches by a constant value, where the height value is given as an argument. Raise is additive, meaning that performing raise multiple times on the same selection is equivalent to performing raise once with the sum of the height arguments.

The tool for raise makes use of the mouse wheel. Scrolling up repeatedly invokes raise, causing the selection to move up by one world unit with each scroll. Scrolling down also invokes raise, but with a negative height argument, causing the selection to move down one world unit with each scroll. If part of

the selection includes virtual patches, the virtual patches are ignored. The raise tool is activated by pressing ‘1’ on the keyboard.

3.2.2 Mound

Mound is a command that elevates every floor patch within a selection, but does not create discontinuities and therefore does not result in the creation of wall patches. The floor patches are not elevated uniformly; instead each point in a floor patch is elevated based on its distance from the center of the selection. When applied to a flat region of the Terrain, this results in the formation of rounded geometry that resembles a hill. If the region is not flat, features of the Terrain within the selection are distorted by the curvature of the mound, and any wall patches that were previously visible will persist [Figure 10]. The curvature of the mound can be adjusted wider or skinnier.

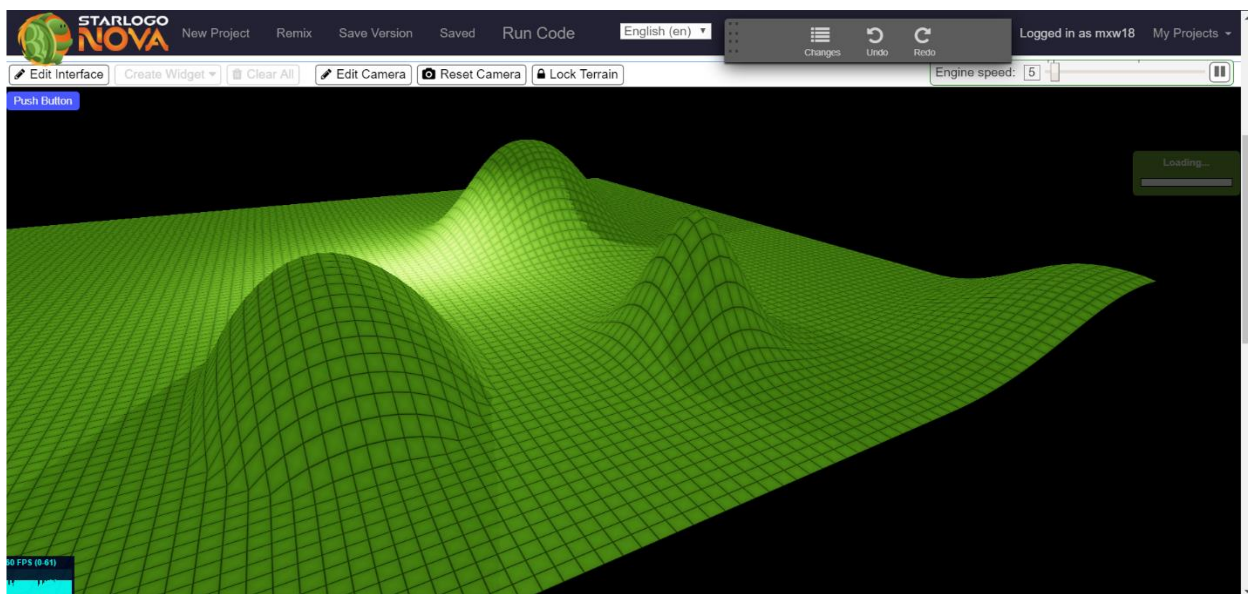

Figure 10: The mound command. The width and tapering of the mound can be adjusted. The rear mound features default width and tapering.

The algorithm for mound uses a special height function to compute the height that each point in each floor patch should be elevated.

$$\Delta z = A\cos\left(\frac{\pi(x-x_0)}{w}\right)\cos\left(\frac{\pi(y-y_0)}{h}\right)\left[max(1-\alpha,0)+\max(\alpha,0)\exp\left(-3\left(\frac{(x-x_0)^2}{w}+\frac{(y-y_0)^2}{h}\right)\right)\right]$$

$$\alpha = \left(2\sqrt{\frac{(x-x_0)^2}{w^2}+\frac{(y-y_0)^2}{h^2}}\right)^p$$

The variables x and y are the coordinates of a point within a floor patch. The variables x_0 and y_0 are the coordinates of the center of the selection. Variables w and h are the width and height of the selection. Variable A is the elevation that would occur at the center of the selection, the highest elevation output by the function. Variable p is an attenuation parameter that controls how quickly the elevation tapers off toward zero, effectively making the mound wider or skinnier.

The height function used by mound is an interpolation between a cosine term and a hybrid gaussian-cosine term. This can be seen by factoring in the cosine term and noting the α function to be the interpolation factor. The α function creates an elliptical interpolation between the two functions. The level sets of the α function are ellipses, where the center of the selection is the $\alpha=1$ level set and the $\alpha=0$ level set is inscribed in the boundaries of the selection. The max function ensures that the interpolation factor does not go below zero in the area between the $\alpha=0$ level set and the boundaries of the selection. This interpolation ensures a smooth transition between the cosine term near the center of the selection and the hybrid gaussian-cosine term near the boundaries.

There are two reasons for choosing this complicated height function over a simpler a cosine, gaussian, or hybrid gaussian-cosine function. Firstly, the shape of the cosine function creates a more well-rounded shape than either the gaussian function or the hybrid function but does not taper off at the edges of the selection. The gaussian function does taper off, and can be truncated to zero at the boundary without any noticeable visual artifacts. The hybrid function also successfully tapes off at the boundaries. An interpolation between the cosine and hybrid function produces a function that retains

the pleasing shape of the cosine function while also tapering off just in time near the selection boundaries with minimal visual artifacts. The second reason for the use of the interpolation factor is that is gives us a degree of control over the shape of the mound that is easy to use. By introducing the attenuation parameter p, we can control how quickly the function transitions from cosine to the hybrid, effectively controlling how wide or skinny the mound is. We can even reproduce the shape of either the cosine or the hybrid by using a very large p or a p close to zero respectively.

The function above is computed at and added to the z coordinate of each corner point within each floor patch. For the center points, they are set to the average height of the corner points. This is done to prevent visual artifacts and to ensure that the result of the mound command is a smooth surface. Mound is additive, meaning that performing mound multiple times on the same selection is equivalent to performing mound once with the sum of the height arguments.

The tool for mound makes use of the mouse wheel and the right mouse button. Scrolling up with the mouse wheel repeatedly invokes mound, causing the selection to move up by one world unit at the center of the selection with each scroll. Scrolling down also invokes mound, but with a negative height argument, causing the selection to move down one world unit at the center of the selection with each scroll. By holding down the right mouse button and dragging left or right, the user can adjust the attenuation parameter p of the height function and adjust the shape produced by mound to be wider or skinnier. If the selection includes virtual patches, the virtual patches are taken into consideration when determining the value of the parameters x_0, y_0, w, and h of the height function, allowing for the creation of partial mounds that do not taper off before reaching the edge of the Terrain. The mound tool is activated by pressing ‘2’ on the keyboard.

3.2.3 Level

Level is a command that sets the height of every point in every floor patch within the selection to the same value. It produces a flat plane within the selection and eliminates wall patches within the selection as well. It can also result in the creation of wall patches around the edges of the selection [Figure 11].

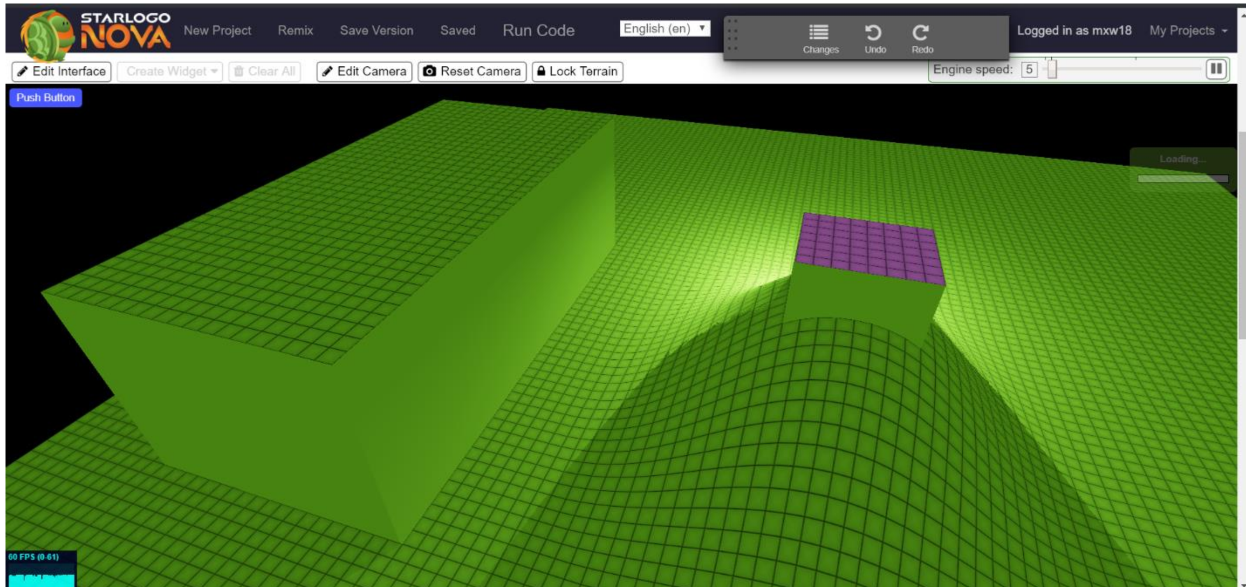

Figure 11: The level command. The top of the curved mound was made flat again and level with the raised structure on the left.

The algorithm for level is simple. Level iterates over each floor patch and sets the height of each point in the patches to the same constant value, where the height value is given as an argument. Level is not additive; performing level again on a selection completely overwrites the previous level command and produces a result equivalent to only the last level command executed.

The tool for level makes use of both the right click and the mouse wheel. Once a selection is made, right clicking on any floor patch will invoke level, passing the height of the minimum point within the right-clicked patch as the height argument. This results in every floor patch within the selection becoming level with the minimum point within the right-clicked floor patch. Scrolling the mouse wheel

up invokes level with one plus the height of the minimum point within the selection, allowing the user to adjust the height of a leveled region after the first invocation of level. Scrolling down similarly invokes level with the height of the minimum point minus one to lower the selection. If part of the selection includes virtual patches, the virtual patches are ignored. The level tool is activated by pressing '3' on the keyboard.

3.2.4 Close Seams

Close seams is a command that removes all wall patches visible within and around the boundaries of a selection. Only the points along discontinuities in the Terrain are adjusted so that the minimal possible work is done to remove the wall patches [Figure 12].

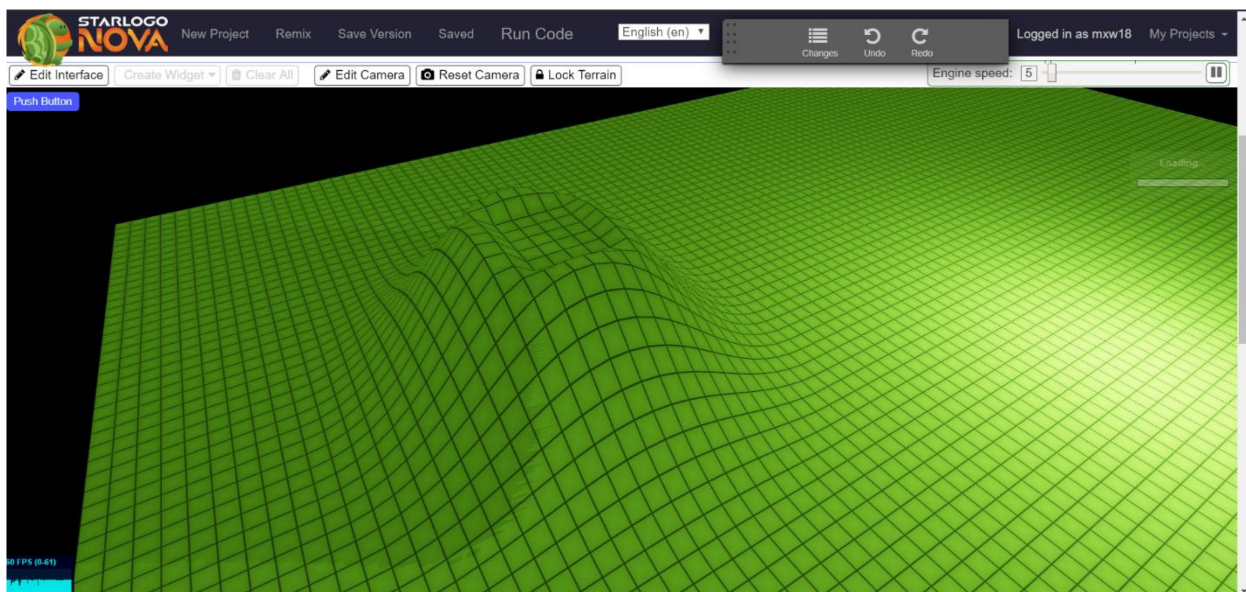

Figure 12: Close seams command. Discontinuities between the mound and the level region on top of it are removed, forming a curved lip.

The algorithm for close seams iterates over all of the corner points within the selection. Because floor patches are aligned on a regular grid, each corner point overlaps with three other corner points from neighboring floor patches. The average height of the corner point and its three neighboring corner points is computed and the corner point is set to the average. A discontinuity exists if and only if

overlapping corner points do not share the same height, so this algorithm only modifies the selection where wall patches are visible.

The tool for close seams is simple. Once a selection is made, the user can right click anywhere in order to execute the command. Close seams is not additive; executing close seams again on a selection that no longer has visible wall patches results in no change. If part of the selection includes virtual patches, the virtual patches are ignored. The close seams tool is activated by pressing '4' on the keyboard.

3.2.5 Ramp

Ramp is a command that creates a smooth, linear transition between the points along opposite edges of a selection. The transition is created in either the direction of the x-axis from left to right, or the direction of the y-axis from top to bottom [Figure 13]. Ramp is useful for joining regions of the Terrain at different heights.

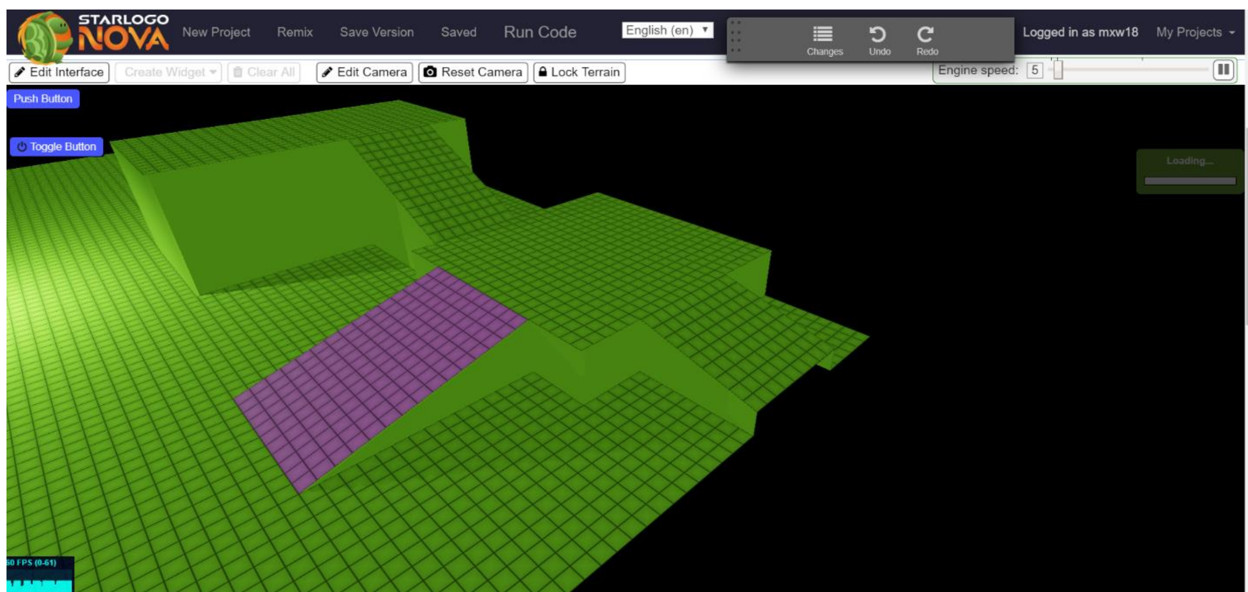

Figure 13: The ramp command used to make connections between raised structures and the ground. Rightmost ramp created with the help of virtual patches.

The algorithm for ramp takes an argument that indicates whether the linear transition should be made in the direction of the x or the y axis. If in the x-axis direction, the algorithm loops over rows of floor patches within the selection that are aligned in the x direction. If in the y-axis direction, the algorithm loops over columns of floor patches within the selection that are aligned in the y dimension. In the case of transition in the direction of the x-axis, for each row of floor patches, the corner points just outside of the selection on the left and right side are linearly interpolated from left to right. This is done twice, separately for both the corner points along the bottom of the row and along the top [Figure 14]. Center points are set to the average z position of the four corner points around them. The procedure is analogous for the y-axis direction.

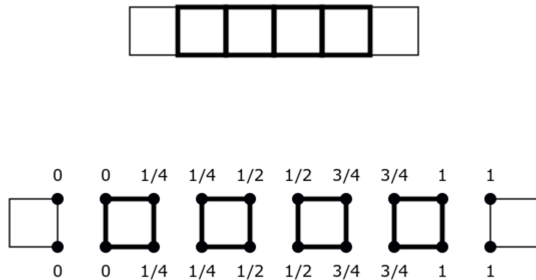

Figure 14: An illustration of a horizontal linear transition computed over corner points by ramp. The bold patches are the floor patches inside the selection. The light patches are neighboring floor patches just outside the selection

The tool for ramp makes use of the right mouse button. Once a selection has been made, the user can right click anywhere to the left or to the right of the selection to create a linear transition in the direction of the x-axis from left to right. Clicking above or below the selection creates a linear transition in the direction of the y-axis from top to bottom. If part of the selection includes virtual patches and the virtual patches extend in the direction of the linear transition, then the linear transition transitions to

zero at the outermost edge of the virtual patches. The ramp tool is activated by pressing ‘5’ on the keyboard.

3.2.6 Copy

Copy is a command that replicates the geometry of the Terrain inside the selection at another location. It functions similarly to copy-and-paste as found in many other software applications that provide an editor interface.

The algorithm behind copy is simple. For every floor patch in the selection, the height of each point is written into a temporary buffer. The contents of the temporary buffer are then written to floor patches at a location specified as an argument.

The tool for copy makes use of the right mouse button. After a selection is made, right clicking on any floor patch makes that floor patch a starting reference. Once the starting reference is chosen, the selection changes into an orange placement guide. Moving the mouse moves the placement guide; the placement guide follows the same displacement from the original selection as the displacement of the mouse from the starting reference. Right clicking again copies the contents of the selection to the current location of the placement guide. If part of the selection includes virtual patches, then the value of zero is copied from the virtual patches. The copy tool is activated by pressing ‘6’ on the keyboard.

3.2.7 Move

Move is a command that is similar to copy. Move replicates the geometry of the Terrain inside the selection at another location and resets the original geometry to zero. It functions similarly to cut-and-paste as found in many other software applications that provide an editor interface.

The algorithm behind move is similar to copy as well. For every floor patch in the selection, the height of each point is written into a temporary buffer. The z position of every point in the floor patches

are then set to zero. Finally, contents of the temporary buffer are then written to floor patches at a location specified as an argument.

The tool for move functions exactly the same as the tool for copy. After a selection is made, right clicking on any floor patch makes that floor patch a starting reference. Once the starting reference is chosen, the selection changes into an orange placement guide. Moving the mouse moves the placement guide; the placement guide follows the same displacement from the original selection as the displacement of the mouse from the starting reference. Right clicking again moves the contents of the selection to the current location of the placement guide. If part of the selection includes virtual patches, then the value of zero is copied from the virtual patches. The move tool is activated by pressing '7' on the keyboard.

3.2.8 Stretch

Stretch is a command that scales the size of all patches uniformly. The Terrain becomes larger as a result of the scaling. It should be noted that, although the size of a floor patch becomes larger, world units remain the same. By default, floor patches are 1 world unit by 1 world unit. Stretch is used to change floor patches to an arbitrary size. Stretch is used to increase the size of the simulation environment without increasing the number of patches in the Terrain. For example, if the size of a floor patch is currently 1x1 world units, and the stretch command is used to double the size of the Terrain, then the size of a floor patch will become 2x2 world units and an agent now has to move twice as far to cover the same number of floor patches. Stretch only affects the x and y dimensions of the patches, the z dimension is unaffected. This is because tools operate in the z dimension in terms of World units.

The algorithm for stretch takes a scale argument. The geometry of the entire Terrain is first written into a temporary buffer. Instead of multiplying the x and y positions of the points in every patch by a multiplicative factor, the position of each patch are computed from the scale argument in the same

manner as the initialization of patch position data. This is to prevent accumulation of numerical error and ensure that patches always have the exact same floating-point position when the same scale argument is passed. After scaling, the z positions of the geometry from the temporary buffer is written to the Terrain ensuring that the z dimension is not affected by stretch.

The tool for stretch makes use of the right mouse button. Because stretch targets all patches in the Terrain, the user cannot make a selection while the stretch tool is active. Holding down the right mouse button hides the Terrain and brings up an orange square guide indicating the overall scale of the Terrain. Dragging away from the center of the guide increases the scale of the Terrain, while dragging toward the center decreases the scale. Releasing the right mouse button invokes the stretch command to resize the Terrain to match the size of the guide. The stretch tool is activated by pressing ‘8’ on the keyboard.

3.2.9 Grow

Grow is a command that increases or decreases the number of patches in the Terrain. This results in the Terrain becoming larger or smaller. Grow adds to or takes away from the perimeter of the Terrain so that the Terrain maintains a square perimeter. By itself, grow can be used to make space for more Terrain content. Grow can also be used in combination with stretch to change the density of patches within the Terrain, allowing the user to create more detailed features or to increase the resolution of simulations where agents interact with the Terrain by increasing the density of the Terrain’s spatial data store and allowing agents to modify the geometry of a denser set of patches. When the Terrain is enlarged using grow, the Terrain’s current geometry is maintained at the center, and the new patches added to the perimeter are set uniformly to height zero. If the Terrain is made smaller, geometry at the perimeter of the Terrain is lost, while the rest of the geometry remains intact.

The algorithm for grow takes a width and height argument for new width and height of the Terrain in floor patches. Similar to stretch, the algorithm first copies the geometry of the Terrain into a temporary buffer, then frees up memory for the current Terrain patches and initializes the required amount for the new Terrain dimensions. The geometry in the temporary buffer is then written back to the center of the Terrain, with any geometry lying outside the new Terrain dimensions being lost. The width and height arguments determine how many patches are created and how they are arranged to form the new Terrain.

In practice, the same value is always used for both width and height to maintain a square Terrain. This limitation is enforced because it is not clear how certain other systems in *StarLogo Nova* should be updated to work with a non-square Terrain, such as the painting system that allows agents to draw on the Terrain. Design choices regarding such systems and their implementation are beyond the scope of this thesis, and non-square Terrain dimensions using grow is ready to be supported once those design choices have been addressed.

The tool for grow makes use of the right mouse button. Because grow affects the entire Terrain, the user cannot make a selection while the grow tool is active. Holding down the right mouse button hides the Terrain and brings up an orange grid indicating the size and patch resolution of the Terrain. Dragging away from the center of the grid increases the patch resolution of the Terrain, while dragging toward the center decreases the patch resolution. Releasing the right mouse button invokes the grow command to resize the Terrain to match the patch count of the grid. The grow tool is activated by pressing ‘9’ on the keyboard.

4 Agent-Terrain Interaction

Agents and the Terrain interact with one another in one of two ways. The first way is through agent movement. When an agent moves, its movement is influenced by the geometry of the Terrain.

The second way is through commands. There are commands available to agents that enable them to manipulate the geometry of the Terrain, which in turn influences their movement, and also commands that enable agents to exchange information with the Terrain. This section discusses these interactions in more detail.

4.1 Terrain Traversal

Agents in *StarLogo Nova* can move about The World using movement blocks. The movement blocks available to agents include the forward, backward, up, and down. The up and down blocks move the agent in the z dimension, while the forward and backward blocks move the agent in the x and y dimensions based on the agent's heading, a property of the agent that determines which way it is facing in the xy-plane. In previous versions of *StarLogo Nova*, agents can move freely through The World with no influence from the environment. Moving an agent forwards or backwards would leave the agent at the same height. We introduce a new movement system to *StarLogo Nova* where moving forward or backwards moves the agent not only in the x and y dimensions, but in the z dimension as well. Movement in the z dimension is determined by the change in the height of the Terrain as the agent moves over it.

To implement this new movement system, we introduce a new property of the agent called altitude. Altitude represents the agent's height above the surface of the Terrain. As an agent moves over the Terrain using the forward or backward block, its altitude remains the same, but its z position may change depending on the height of the Terrain beneath its current position [Figure 15]. A positive altitude means the agent is above the Terrain, while a negative altitude means the agent is below the Terrain. An altitude of zero means the agent is just on top of the Terrain. Moving with the up or down

blocks adjusts the altitude of the agent up or down.

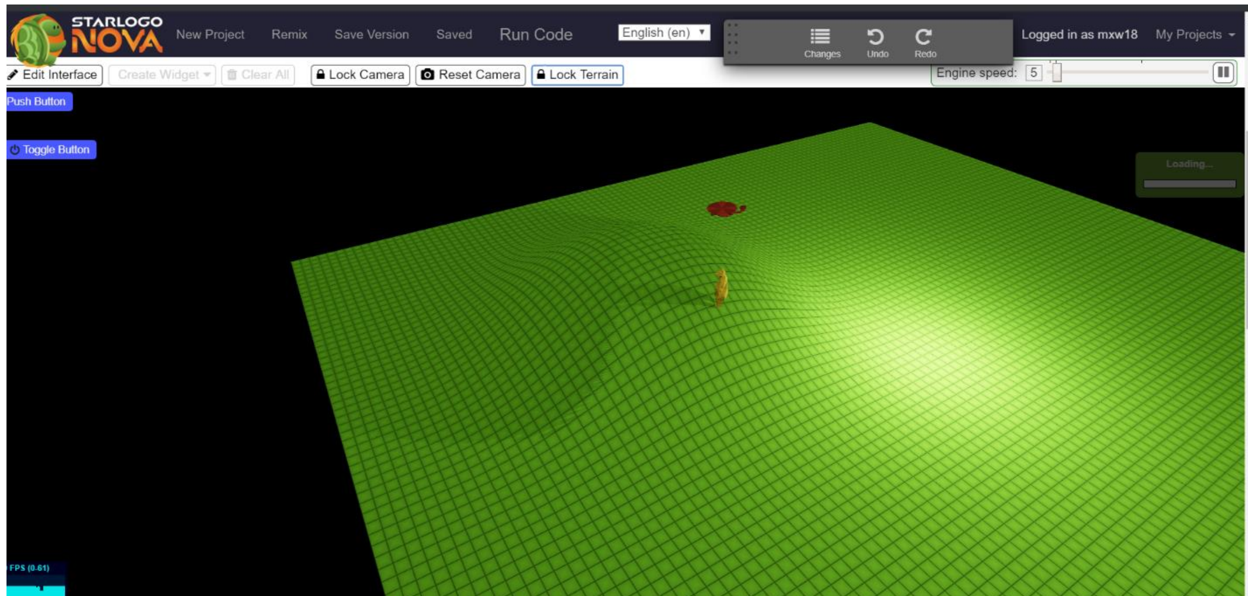

Figure 15: The bear and the helicopter are in the same xy-position, following the slope of a mound as they move forward. The bear is at altitude zero, and the helicopter at a positive altitude above the terrain.

In order to define the altitude of an agent, we must define the height of the Terrain for any xy-position. In previous sections, we have mentioned that floor patches possess handles for manipulating their geometry called points. Points have a z coordinate value that determines the height of the floor patch at that point. However, we have not defined the height of a floor patch continuously in the areas between points. The height of the Terrain at any arbitrary xy-position is determined by the triangle mesh of the Terrain. We define it formally in the following way. For a position (x, y), the height of the Terrain at (x, y) is the z coordinate of the intersection between the triangle mesh and a line that passes through the coordinate (x, y, 0) perpendicular to the xy-plane [Figure 16]. The height of the Terrain is a piecewise linear function due to this definition.

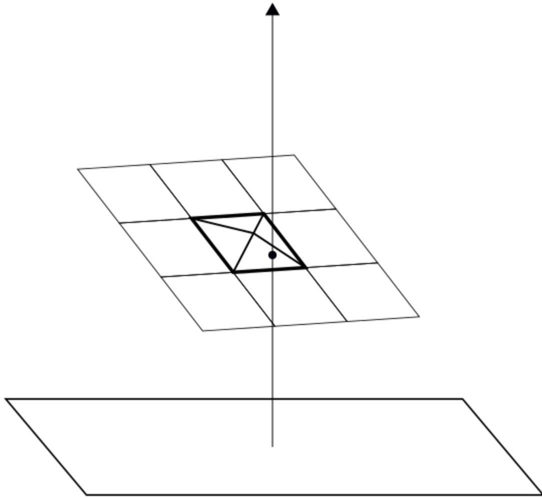

Figure 16: An illustration of a ray perpendicular to the z=0 plane intersecting a triangle within a floor patch. The z coordinate of the point of intersection is the height of the Terrain at that location.

4.2 Terrain Modification

Several commands are available to agents to manipulate the geometry of the Terrain. Unlike the commands available to the user through the Terrain Editor, these commands target patches within a small, local area based on the agent's xy-position. These commands are made available to agents through blocks.

In the following subsections we give a qualitative description of the effect of each command and give a high-level description of the algorithm used to provide that effect. In each of the following subsections, the phrase "floor patch beneath the agent" refers to the floor patch directly beneath the agent's position when looking down the negative z-axis; the agent's position falls between the four borders of the floor patch when projected to the xy-plane. The phrase "volume preserving" means that a command, when given the same arguments, always adds the same amount of signed volume to the Terrain. For a volume preserving command, the signed volume added to the Terrain is also directly proportional to one or more of the command's arguments. Neither the location of the agent, nor the

geometry of the Terrain affect the signed volume added by the command. The signed volume of the Terrain can be thought of as the volume between the triangle mesh of the Terrain and the z=0 plane, where the volume counts as positive where the Terrain's height is positive and negative where the Terrain's height is negative. This volume preserving property can be useful for simulations that require conservation of mass. Section 6 gives an example of such a simulation.

4.2.1 Build / Dig

Build is a command that takes a height argument and elevates the floor patch beneath the agent uniformly by that height. Build can create discontinuities in the Terrain that result in wall patches becoming visible. Build is also volume preserving. It is similar to raise in its behavior.

The algorithm for build is simple. Each point of the floor patch beneath the agent is elevated by the height argument.

Dig is a counterpart command to build. It is equivalent to invoking build with a height argument multiplied by -1.

4.2.2 Yank / Stomp Grid

Yank grid is a command that takes a height argument and elevates the floor patch beneath the agent as well as the neighboring floor patches in a three by three grid [Figure 17]. Yank grid elevates the floor patches in a way so as not to create wall patches. Yank grid is volume preserving, adding the most volume to the floor patch beneath the agent and adding volume to the surrounding floor patches

following a discrete gaussian distribution.

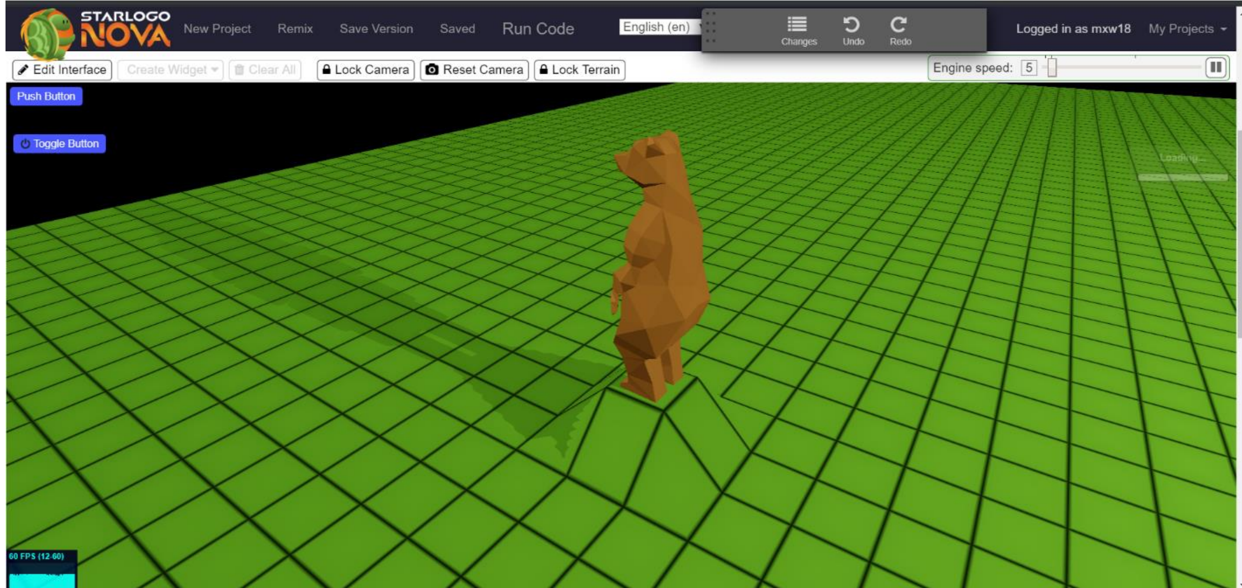

Figure 17: An agent executing the yank grid command.

The algorithm for yank grid uses precalculated values from a template in order to determine the height that should be added to each point within the three by three grid affected by yank grid [Figure 18]. These template values are scaled by a height argument. The choice of template values ensures that wall patches do not get created by assigning the same value to overlapping corner points. The template values also ensure that the total signed volume added by yank grid is distributed in a discrete gaussian distribution. One fourth of the volume is distributed to the center floor patch, one eighth to the directly

adjacent floor patches, and one sixteenth to the corner floor patches.

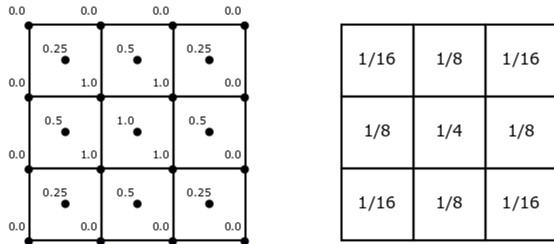

Figure 18: The left illustration shows the template used for elevating each point within the 3x3 grid of floor patches. The decimals near each point are the fraction of the height argument that should be added to that point. The right illustration shows the breakdown of the signed volume distributed to each floor patch, a normalized, discrete gaussian kernel.

Stomp grid is a counterpart command to yank grid. It is equivalent to invoking yank grid with a height argument multiplied by -1.

4.2.3 Yank / Stomp

Yank is a command that takes a height argument and elevates the floor patches in the vicinity of the agent. The size of the vicinity is determined by a second width argument. Unlike yank grid, the peak of the elevation is not aligned with the center of any floor patch. It is aligned with the xy-position of the agent instead. Yank is a continuous version of yank grid. Like yank grid, yank does not result in the creation of wall patches. Yank is not volume preserving, the signed volume added to the Terrain is dependent on where the xy-position of the agent lies within the floor patch beneath the agent. As a compromise, yank does preserve the total height added to points in the Terrain and distributes that height in a gaussian distribution over the vicinity determined by the width argument. Height

preservation can be thought of as a discrete analog of volume preservation, with the points within a floor patch acting as the discrete sampling positions.

The algorithm for yank first computes the boundaries of a square region around the xy-position of the agent. The width of the square region is determined by the width parameter. A gaussian kernel function then computes a height displacement at every point that falls within the square region. The gaussian kernel is centered on the xy-position of the agent, and the standard deviation of the gaussian kernel is one third the width argument. After the height displacements have been computed by the gaussian kernel, they are reweighted so that their sum equals the height argument. This enforces the height preservation discussed in the previous paragraph.

4.3 Patch Data

Several commands are available to agents for reading from and writing to the data stores at each floor patch of the Terrain. These commands are made available to agents through blocks. Because these commands do not alter the geometry of the Terrain, they are not recorded by the undo/redo system. The following subsections use the phrase "floor patch beneath the agent" as described previously in Section 4.2.

4.3.1 Get / Set Attribute

Set attribute is a command for writing a numerical value to the data store of the floor patch beneath the agent. Set attribute takes a string argument in addition to the numerical value. The data store behaves similarly to a dictionary, so that if the string is not present in the data store, the numerical value is associated with the string. If the string is already present in the data store, then the numerical value overwrites the value that was previously associated with that string.

Get attribute is a command for reading a numerical value from the data store of the floor patch beneath the agent. Get attribute takes a string argument. As mentioned previously, the data store

behaves like a dictionary. If the string is present in the data store, then the last value associated with that string by set attribute is returned. If the string is not present in the data store, then zero is returned.

4.3.2 Interpolate Attribute

It is a common task to have to interpolate values between elements in a spatial data structure. Interpolate attribute is a command that provides this functionality. Interpolate attribute performs bilinear interpolation between the data stores of the four floor patches closest to the xy-position of the agent. The data stores are located at the center point of their respective floor patch. Interpolation is performed based on where the xy-position of the agent falls between the xy-positions of the center points of the four nearest floor patches. For those unfamiliar with bilinear interpolation, we refer the reader to [Press 1992].

5 Software Design and Implementation Details

The previous sections have described the new features provided by the work of this thesis as they appear to a *StarLogo Nova* user. In this section we discuss the software representation of the Terrain and the interface that it exposes to the rest of the *StarLogo Nova* application that make the new features possible. We also discuss the data representation of the Terrain's state and the data structures used to organize and maintain that state. We also shed some light on how the Terrain's state goes from data to the visualization seen by users on screen, and how users and agents are able to interact with that data.

5.1 Terrain Class

Following an object-oriented design pattern, the Terrain is represented by a class implemented in Javascript. In this section, we describe the key methods of the Terrain class and how they are used. By creating an instance of the Terrain class and making calls to the following methods, *StarLogo Nova* is

able to provide the features that have been described in Section 3 and Section 4. We also explain in this section the implementation of features that are managed internally by the Terrain class, namely the undo/redo system.

Some methods of the Terrain class make use of a different coordinate system than The World that uses the width of a patch as the unit of measure. We refer to the coordinate system of the Terrain as patch coordinates and the coordinate system of The World as world coordinates from now on. Patch coordinates are a two-dimensional coordinates system only defined in the xy-plane, unlike the three-dimensional world coordinates that are defined anywhere in space. Integral patch coordinates align with the at the center points of every floor patch. The bottom, leftmost floor patch is located at (0, 0), with the x patch coordinate increasing from left to right and the y patch coordinate increasing from bottom upwards.

5.1.1 Methods and Interface

Public Interface:

toggleEditMode (enable)
    enable: True to enable, false to disable.
Enables or disables the Terrain Editor.

setMouseCommand (command)
    command: Bitwise OR of flags representing different commands.
Activates the tools indicated by the command argument and disables all other tools.

setWheelSpeed (sensitivity)
    sensitivity: New sensitivity of the mouse wheel.
Changes the sensitivity of the mouse wheel for tools in proportion to the sensitivity argument.

select (left, bottom, right, top)
    left: Left boundary of a selection, inclusive. Integer patch coordinate.
    bottom: Bottom boundary of a selection, inclusive. Integer patch coordinate.
    right: Right boundary of a selection, inclusive. Integer patch coordinate.
    top: Top boundary of a selection, inclusive. Integer patch coordinate.

Marks a rectangular region of floor patches the selection to be manipulated by the next command from the user in the Terrain Editor.

clearSelection ()
Sets a null selection in the Terrain Editor.

patchToWorld (patch_x, patch_y)

 patch_x: Horizontal coordinate in patch coordinate space. Floating point.
 patch_y: Vertical coordinate in patch coordinate space. Floating point.

Converts patch coordinates to world coordinates and returns them as a length two array.

worldToPatch (world_x, world_y)

 world_x: Horizontal coordinate in world coordinate space. Floating point.
 world_y: Vertical coordinate in world coordinate space. Floating point.

Converts world coordinates to patch coordinates and returns them as a length two array.

getHeightAt (x, y)

 x: Horizontal coordinate in patch coordinate space. Floating point.
 y: Vertical coordinate in patch coordinate space. Floating point.

Computes the height of the Terrain at an arbitrary point using barycentric interpolation.

raise (height, x, y, right, top, logHistory)

 height: Distance to elevate in world units.
 x: Left boundary of a selection, inclusive. Integer patch coordinate.
 y: Bottom boundary of a selection, inclusive. Integer patch coordinate.
 right: Right boundary of a selection, inclusive. Integer patch coordinate.
 top: Top boundary of a selection, inclusive. Integer patch coordinate.
 logHistory: True to log command with the undo/redo system. False to omit.

Performs the raise command on the region defined by x, y, right, and top. If right or top is null, perform the command on the patch specified by x, y. If x or y is null, perform the command on the current selection.

mound (amplitude, attenuation, x, y, right, top, logHistory)

 amplitude: Distance to elevate in world units.
 x: Left boundary of a selection, inclusive. Integer patch coordinate.
 y: Bottom boundary of a selection, inclusive. Integer patch coordinate.
 right: Right boundary of a selection, inclusive. Integer patch coordinate.
 top: Top boundary of a selection, inclusive. Integer patch coordinate.
 logHistory: True to log command with the undo/redo system. False to omit.

Performs the mound command on the region defined by x, y, right, and top. If right or top is null, perform the command on the patch specified by x, y. If x or y is null, perform the command on the current selection.

level (x, y, right, top, logHistory)

x: Left boundary of a selection, inclusive. Integer patch coordinate.

y: Bottom boundary of a selection, inclusive. Integer patch coordinate.

right: Right boundary of a selection, inclusive. Integer patch coordinate.

top: Top boundary of a selection, inclusive. Integer patch coordinate.

logHistory: True to log command with the undo/redo system. False to omit.

Performs the level command on the region defined by x, y, right, and top. If right or top is null, perform the command on the patch specified by x, y. If x or y is null, perform the command on the current selection.

closeSeams (method, x, y, right, top, logHistory)

method: A constant indicating what method should be used to remove wall patches, joining floor patches at the minimum, maximum, or average of their overlapping points.

x: Left boundary of a selection, inclusive. Integer patch coordinate.

y: Bottom boundary of a selection, inclusive. Integer patch coordinate.

right: Right boundary of a selection, inclusive. Integer patch coordinate.

top: Top boundary of a selection, inclusive. Integer patch coordinate.

logHistory: True to log command with the undo/redo system. False to omit.

Performs the close seams command on the region defined by x, y, right, and top using the method indicated by the method argument to seal seams. If right or top is null, perform the command on the patch specified by x, y. If x or y is null, perform the command on the current selection.

ramp (direction, x, y, right, top, logHistory)

direction: A constant indicating whether the direction of the ramp should be left to right or top to bottom.

x: Left boundary of a selection, inclusive. Integer patch coordinate.

y: Bottom boundary of a selection, inclusive. Integer patch coordinate.

right: Right boundary of a selection, inclusive. Integer patch coordinate.

top: Top boundary of a selection, inclusive. Integer patch coordinate.

logHistory: True to log command with the undo/redo system. False to omit.

Performs the ramp command on the region defined by x, y, right, and top in the direction indicated by the direction argument. If right or top is null, perform the command on the patch specified by x, y. If x or y is null, perform the command on the current selection.

copy (dst_x, dst_y, x, y, right, top, logHistory)

dst_x: Horizontal coordinate in patch coordinate space. Integer.

dst_y: Vertical coordinate in patch coordinate space. Integer.

x: Left boundary of a selection, inclusive. Integer patch coordinate.

y: Bottom boundary of a selection, inclusive. Integer patch coordinate.

right: Right boundary of a selection, inclusive. Integer patch coordinate.

top: Top boundary of a selection, inclusive. Integer patch coordinate.

logHistory: True to log command with the undo/redo system. False to omit.

Performs the copy command on the region defined by x, y, right, and top and copies to a region beginning at (dst_x, dst_y). If right or top is null, perform the command on the patch specified by x, y. If x or y is null, perform the command on the current selection.

move (dst_x, dst_y, x, y, right, top, logHistory)

dst_x: Horizontal coordinate in patch coordinate space. Integer.

dst_y: Vertical coordinate in patch coordinate space. Integer.

x: Left boundary of a selection, inclusive. Integer patch coordinate.

y: Bottom boundary of a selection, inclusive. Integer patch coordinate.

right: Right boundary of a selection, inclusive. Integer patch coordinate.

top: Top boundary of a selection, inclusive. Integer patch coordinate.

logHistory: True to log command with the undo/redo system. False to omit.

Performs the move command on the region defined by x, y, right, and top and moves to a region beginning at (dst_x, dst_y). If right or top is null, perform the command on the patch specified by x, y. If x or y is null, perform the command on the current selection.

build (height, x, y, logHistory)

height: Distance to elevate in world units.

x: Horizontal coordinate in patch coordinate space. Integer.

y: Vertical coordinate in patch coordinate space. Integer.

logHistory: True to log command with the undo/redo system. False to omit.

Performs the build command on the floor patch located at (x, y). Dig can be implemented by supplying a negative argument for height.

yankGrid (height, x, y, logHistory)

height: Distance to elevate center in world units.

x: Horizontal coordinate in patch coordinate space. Integer.

y: Vertical coordinate in patch coordinate space. Integer.

logHistory: True to log command with the undo/redo system. False to omit.

Performs the yank grid command centered on the floor patch located at (x, y). Stomp grid can be implemented by supplying a negative argument for height.

yank (mass, width, x, y, logHistory)

mass: Total height elevation to distribute.

width: Width in floor patches of the area to distribute height elevation over.

x: Horizontal coordinate in patch coordinate space. Floating point.

y: Vertical coordinate in patch coordinate space. Floating point.

logHistory: True to log command with the undo/redo system. False to omit.

Performs the yank command centered on the location (x, y). Stomp can be implemented by supplying a negative argument for mass.

setCustomAttribute (attributeName, value, x, y)

attributeName: Key used for reading and for writing to the value. String.

value: The value to be stored.
x: Horizontal coordinate in patch coordinate space. Integer.
y: Vertical coordinate in patch coordinate space. Integer.

Performs the set attribute command on the floor patch at location (x, y).

getCustomAttribute (attributeName, x, y)

attributeName: Lookup key for reading value. String.
x: Horizontal coordinate in patch coordinate space. Integer.
y: Vertical coordinate in patch coordinate space. Integer.

Performs the get attribute command on the floor patch at location (x, y).

interpolateCustomAttribute (attributeName, x, y)

attributeName: Lookup key for interpolating value. String.
x: Horizontal coordinate in patch coordinate space. Floating point.
y: Vertical coordinate in patch coordinate space. Floating point.

Performs the interpolate attribute command at location (x, y).

getScale ()

Return the current side length of a floor patch in world units.

setScale (scale)

scale: The new scale of floor patches in the Terrain, provided in world units.

Performs the stretch command on the Terrain. Note that stretch is not currently supported by the undo/redo system.

getPatchWidth ()

Returns the current width of the Terrain in floor patches.

getPatchHeight ()

Returns the current height of the Terrain in floor patches.

setPatchDimensions (width, height, logHistory)

width: The new width of the Terrain in floor patches.
height: The new height of the Terrain in floor patches.
logHistory: True to log command with the undo/redo system. False to omit.

Performs the grow command on the Terrain.

undoCommand ()

Performs the undo command.

redoCommand ()

Performs the redo command.

Private Methods:

getPatchLocation (x, y)

x: Horizontal coordinate in screen space. Integer pixel position.

y: Vertical coordinate in screen space. Integer pixel position.

Utilizes the picking system discussed later in Section 5.4 in order to return the ID of the patch visible at pixel screen coordinate (x, y), if any.

getZeroPlaneIntersect (screenX, screenY, continuous)

screenX: Horizontal coordinate in screen space. Integer pixel position.

screenY: Horizontal coordinate in screen space. Integer pixel position.

continuous: True to return floating point patch coordinates, false to return nearest integer patch coordinates.

Returns the patch coordinates as a length two array of the intersection between a ray emanating from the camera through the pixel located at screen coordinates (screenX, screenY) and the z=0 plane.

readSubData (left, bottom, right, top)

left: Left boundary of a selection, inclusive. Integer patch coordinate.

bottom: Bottom boundary of a selection, inclusive. Integer patch coordinate.

right: Right boundary of a selection, inclusive. Integer patch coordinate.

top: Top boundary of a selection, inclusive. Integer patch coordinate.

Returns a buffer with a copy of the geometry of the patches within the boundaries determined by left, bottom, right, and top. Specifically, it holds a copy of the position and normal attribute data explained later in Section 5.2. This buffer can be written to another region of the Terrain using writeSubData().

writeSubData (x, y, subData)

x: Horizontal coordinate in patch coordinate space. Integer.

y: Vertical coordinate in patch coordinate space. Integer.

subData: A buffer formatted like those returned by a call to readSubData.

Writes the subData buffer to the geometry of the Terrain starting at the floor patch located at (x, y) as the bottom, leftmost floor patch to write to.

initPatches ()

Initializes or reinitializes the data structures discussed later in Section 5.3.2.

colorSelection (r, g, b, left, bottom, right, top)

r: Red RGB color channel value between 0 and 1.

g: Green RGB color channel value between 0 and 1.

b: Blue RGB color channel value between 0 and 1.

left: Left boundary of a selection, inclusive. Integer patch coordinate.

bottom: Bottom boundary of a selection, inclusive. Integer patch coordinate.
right: Right boundary of a selection, inclusive. Integer patch coordinate.
top: Top boundary of a selection, inclusive. Integer patch coordinate.

Applies a colored highlight to the floor patches within the boundaries given by left, bottom, right, and top. Used by the Terrain Editor for UI purposes. Supplying zero for r, g, and b removes any highlights.

pushCommandToHistory (command, argList, restoreData)

command: A reference to a method implementing a command.
argList: An array containing the arguments that were used with command in order.
restoreData: A buffer like the one returned by readSubData().

Logs command with the undo/redo system. The restoreData parameter provides the previous state of the geometry of the target of command before it was executed.

5.1.2 Commands and the Undo/Redo System

Each command presented in Section 3 and Section 4 has a corresponding method in the public interface of the Terrain class. We refer to these methods as command methods. We also refer to the floor patches that are altered by a call to a command method the target of the command method. Every command method must follow a design pattern in order to work with the undo/redo system. A command method first uses the readSubData() private method to obtain a buffer containing a copy of the target's geometry. The command method then proceeds to alter the geometry of the Terrain. After altering the Terrain, a command method must call the pushCommandToHistory() private method in order to log the command it executed with the undo/redo system. The arguments that the command method received and the buffer returned by readSubData() are also logged with the command.

The undo/redo system maintains an internal history. This history is a list of commands that have been executed thus far, along with the arguments of the command and the geometry of the target before the command was executed. When a user uses the undo command, a call to undoCommand() is made, and for the redo command a call to redoCommand() is made. The methods undoCommand() and redoCommand() maintain an index into the history called the history index that points to the last command executed to achieve the current geometry of the Terrain. Calling undoCommand() uses the

buffer stored with the command at the current history index to reverse the effect of the command with a call to writeSubData(), then decrements the history index. Calling redoCommand() simply increments the history index and executes the next command in history with the arguments that were stored with it. When a new command is logged with the undo/redo system, any commands past the current history index are forgotten and replaced with the newly logged command. This implements the behavior of a traditional undo/redo system as commonly found in other software.

Commands that are considered additive are treated differently than other commands by the undo/redo system. When an additive command is executed in succession with the same target, the additive property is exploited by the undo/redo system, and the history is revised so that the net effect of the additive commands is logged as a single equivalent command. This special treatment of additive commands is intended to make the Terrain Editor easier to use. Tools that make use of the mouse wheel would normally log a several commands from a single scroll of the wheel, meaning the user would have to use the undo command several times to undo the result of a single scroll action. Combining additive commands allows a single scroll action to be undone by a single undo command.

5.2 3D Visualization

Spaceland is visualized in *StarLogo Nova* by a 3D rendering engine implemented in WebGL. WebGL is an API that allows Javascript web applications to communicate with the GPU. *StarLogo Nova* uses WebGL to upload a data representation of Spaceland to the GPU and sends instructions to the GPU on how to render the data representation to an image onscreen inside the viewport. For more information on 3D rendering and computer graphics using WebGL, we refer the reader to [Parisi 2012]. *StarLogo Nova* must provide the GPU with a data representation of the Terrain that can be rendered to the screen. This section discusses the data representation of the Terrain that is used by both the GPU for

visualization and internally as the geometry that agents traverse when moving and manipulate through commands.

StarLogo Nova's 3D rendering engine requires geometry to be specified as triangles defined by a set of 3 vertices. This is the reason we chose the triangle mesh as the geometric representation of the Terrain as explained in Section 2.1. The triangle mesh is suitable both for 3D rendering with the existing rendering engine and interaction with agents. Interaction with agents only requires triangles to specify a 3D position for each triangle vertex. However, *StarLogo Nova's* 3D rendering engine requires additional information to be specified at each vertex. We summarize the information that needs to be specified at each triangle vertex, called attributes, and how they are relevant to the visualization of the Terrain.

- Position: A coordinate specifying the location of the vertex in three-dimensional space.
- Normal: A normal is a three-dimensional vector that points perpendicular to the tangent plane of a surface. Normals are used to produce a shading effect on objects when illuminated by a light source. The normal is interpolated across the surface of the triangle during the rendering process by the GPU. Although triangles have a constant normal direction across the entire surface of the triangle, by specifying a different normal at each of the triangle's three vertices, the triangle can be made to appear curved rather than flat by the rendering process. This is critical to producing the visualization of geometric features created by mound and ramp.
- UV Coordinate: UV coordinates specify a mapping from a triangles surface to a two-dimensional image. A painting system allows agents in *StarLogo Nova* to color the Terrain to create images on the surface of the Terrain. The painting system requires uv coordinates to be supplied to the GPU in order to work.

- Color: An RGB color value. The 3D rendering engine special-cases the Terrain so that color produces a colored highlight over the triangle. This is how the purple and orange highlighting of floor patches is achieved by the Terrain Editor.
- ID: An integer value used to identify triangles. Each vertex should have the same ID. The purpose of IDs are explained later in Section 5.4.

Attributes are stored in arrays, with a separate array for each attribute. Attribute values located at the same index within their respective arrays belong to the same vertex. A separate array called an index array determines the triangle that an attribute belongs to. Every three elements in the index array specify a triangle, with the elements being offsets into the attribute arrays that associate the attribute data with a particular triangle vertex. The index array is required by the *StarLogo Nova* rendering engine for rendering.

The grid overlay that appears on the Terrain when the Terrain Editor is active is a procedurally generated texture, meaning that the placement and position of the gridlines are computed on the GPU based on the geometry of the Terrain. In particular, the position attribute is used to procedurally generate the texture. The code for procedurally generating the grid overlay is written in GLSL and was inserted into the shader program used by the 3D rendering engine to render the Terrain.

5.3 Patch Class

Instances of the Patch class represent the individual patches within the Terrain. The methods within the Terrain class, namely the command methods, operate on instances of the Patch class in order to manipulate the geometry of the Terrain. The Patch class provides a layer of abstraction between the Terrain class and the underlying attributes of the triangle mesh used to represent the Terrain.

The Patch class is used to represent both floor patches and wall patches, though some methods are only meant to be called on floor patches. An instance of the Patch class stores pointers to the attribute arrays used to represent the Terrain, as well as an offset into the arrays. The elements of the attribute arrays are ordered in such a way that, given the offset to the first vertex included in a patch, the offset to the data for other vertices in the same patch can easily be computed. As stated in Section 2.2, patches possess handles called points used to manipulate the underlying triangles of the patch. There is a correspondence between the points within a patch and the vertices of the triangles within a patch [Figure 19]. The position of a point is the same as the position of the vertices associated with it, so manipulating points directly manipulates the vertices. The Patch class provides several methods for reading and writing to the attribute data at each point within a patch, an abstraction that simplifies the implementation of most commands when more than one vertex is associated with a point.

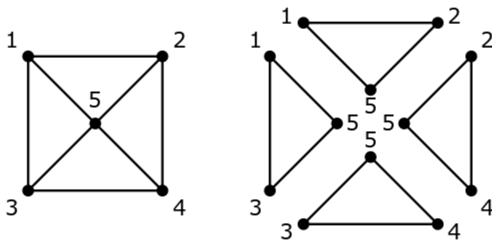

Figure 19: The floor patch shown on the left has 4 underlying triangles shown on the right. The numberings show the correspondence between patch points and triangle vertices.

Instances of the Patch class can also be used as nodes in a data structure similar to a linked-list by setting the neighbors of a patch to be another patch. We show later in Section 5.3.2 how instances of

the Patch class can be arranged into data structures that further simplify the implementation of many commands.

5.3.1 Methods and Interface

Public Interface:

getNeighbor (neighborNumber, distance)

neighborNumber: A constant indicating either the left, right, bottom, or top neighbor.

distance: Number of steps away to retrieve the neighbor. A value of 1 retrieves the patch's neighbor, a value of 2 returns the neighbor of the patch's neighbor, and so forth. Defaults to 1.

Returns a neighboring patch instance or a patch instance some number of steps away as determined by the arguments.

setNeighbor (patch, neighborNumber)

patch: A patch instance.

neighborNumber: A constant indicating either the left, right, bottom, or top neighbor.

Sets patch to be the direct neighbor of this patch instance in the direction determined by neighborNumber.

getPosition (pointNumber)

pointNumber: A constant specifying one of the 5 points in a floor patch (or 4 points in a wall patch).

Returns the position of a point in the patch as a length 3 array.

getNormal (pointNumber)

pointNumber: A constant specifying one of the 5 points in a floor patch (or 4 points in a wall patch).

Returns the normal at a point in the patch as a length 3 array.

getColor (pointNumber)

pointNumber: A constant specifying one of the 5 points in a floor patch (or 4 points in a wall patch).

Returns the color at a point in the patch as a length 3 array.

getUv (pointNumber)

pointNumber: A constant specifying one of the 5 points in a floor patch (or 4 points in a wall patch).

Returns the uv coordinate at a point in the patch as a length 2 array.

getId ()

Returns the ID of the patch instance.

getNumVertices ()
Returns the number of unique vertices in the patch. This is equivalent to the number of points in the patch.

getNumVerticesRepeated ()
Returns the number of vertices in the underlying triangles of the patch. This is equivalent to the sum of the number of vertices associated with each point in the patch.

setPosition (x, y, z, pointNumber)

x: Floating point value.

y: Floating point value.

z: Floating point value.

pointNumber: A constant specifying one of the 5 points in a floor patch (or 4 points in a wall patch).

Sets the components of the position of a point in the patch. If the patch is a floor patch, then the points of neighboring wall patches are updated automatically so that holes do not appear in the Terrain.

setNormal (x, y, z, pointNumber)

x: Floating point value.

y: Floating point value.

z: Floating point value.

pointNumber: A constant specifying one of the 5 points in a floor patch (or 4 points in a wall patch).

Sets the components of the normal at a point in the patch.

setUv (u, v, pointNumber)

u: Floating point value.

v: Floating point value.

pointNumber: A constant specifying one of the 5 points in a floor patch (or 4 points in a wall patch).

Sets the components of the uv coordinate at a point in the patch.

setColor (r, g, b, pointNumber)

r: Red RGB color channel value between 0 and 1.

g: Green RGB color channel value between 0 and 1.

b: Blue RGB color channel value between 0 and 1.

pointNumber: A constant specifying one of the 5 points in a floor patch (or 4 points in a wall patch).

Sets the RGB components of the color at a point in the patch.

setPatchSmoothNormals(smoothOverWalls, exclude)

smoothOverWalls: True to ignore triangle faces from floor patch neighbors when the wall patch between this floor patch and the neighboring floor patch is visible. False to include them.

exclude: A lookup table specifying which faces from this floor patch and neighboring floor patches to ignore.

Alters the normal within the floor patch to give it a smooth appearance when visualized by the 3D rendering engine. Calculates the new normals at each point by averaging the face normals of the triangles with a vertex that overlaps the point. This includes faces from neighboring floor patches. Arguments are used to exclude certain triangles from the calculation to avoid smoothing the appearance of the floor patch with certain neighbors. Can only be called on a floor patch.

interpolateZ (x, y)

x: A value between 0 and 1, where 0.5 corresponds to the horizontal center of the floor patch.

y: A value between 0 and 1, where 0.5 corresponds to the vertical center of the floor patch.

Performs barycentric interpolation within one of the triangles of the floor patch in order to find the height of the floor patch at an arbitrary location within the floor patch. Can only be called on a floor patch.

getMinPosition (pointNumber)

pointNumber: A constant specifying one of the 5 points in a floor patch (or 4 points in a wall patch).

Returns the minimum position of a point in the floor patch and the points of neighboring floor patches that overlap the specified point as a length 3 array.

getMaxPosition (pointNumber)

pointNumber: A constant specifying one of the 5 points in a floor patch (or 4 points in a wall patch).

Returns the maximum position of a point in the floor patch and the points of neighboring floor patches that overlap the specified point as a length 3 array.

getAveragePosition (pointNumber)

pointNumber: A constant specifying one of the 5 points in a floor patch (or 4 points in a wall patch).

Returns the average position of a point in the floor patch and the points of neighboring floor patches that overlap the specified point as a length 3 array.

updateWallNormal (neighborNumber)

neighborNumber: A constant indicating either the left, right, bottom, or top neighbor.

Calculates and sets appropriate normals at each point of a wall patch based on the positions of the points. Can only be called on a wall patch.

5.3.2 Patch Data Structure

There are two data structures that instances of the Patch class are organized into. The first one is a simple spatial lookup data structure. The Terrain class maintains a two-dimensional array with the same width and height as the width and height of the Terrain in floor patches. The integer patch coordinates of a floor patch are used as indices to insert the floor patch instances into this spatial data structure. This data structure makes it easy for command methods to iterate over all the floor patches within their target.

The second data structure incorporates both floor patches and wall patches. It is similar to a linked-list data structure, where instances of the Patch class are the nodes, but the data structure is two-dimensional. For each floor patch, its neighboring nodes are set to be the wall patches directly adjacent to it; in other words, the wall patch that lies between the floor patch and the next floor patch over [Figure 20]. This data structure provides patches with an awareness of their neighbors and other patches in their local vicinity. This data structure helps with the implementation of methods where patches interact with or must update their neighbors. An example of this is how floor patches automatically update the vertices of their neighboring wall patches whenever their own vertices change so that holes do not appear in the Terrain.

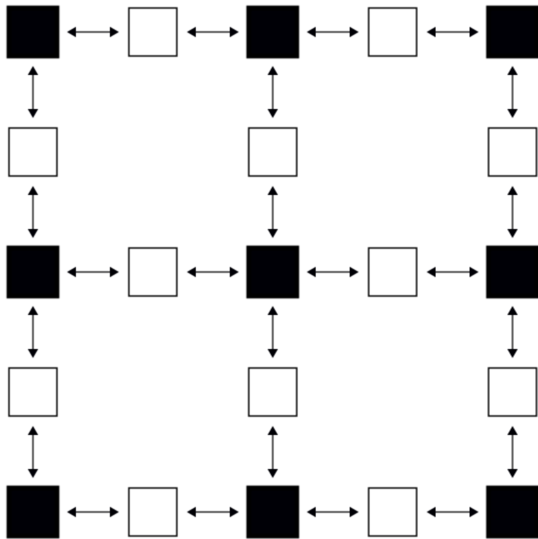

Figure 20: An illustration of the linked patch data structure. Black squares represent floor patches and white squares represent the wall patches that lie between them. Both floor and wall patches are treated as nodes in the data structure, and the arrows show the linked-list style connections between nodes.

5.4 Picking System

As part of the work of this thesis, a picking system was added to *StarLogo Nova's* 3D rendering engine. When a user's mouse is over the viewport, the picking system identifies the nearest object that has been rendered at the location of the mouse, if any. The picking system what the Terrain class uses to implement selection in the Terrain Editor. The precise floor patch beneath the mouse can be identified by the picking system.

The picking system works by performing 3D rendering using *StarLogo Nova's* rendering engine, but instead of rendering colors to pixels in an image, integer IDs are rendered to the pixels instead. There is a one to one correspondence between the image containing the IDs and the image in the viewport. The IDs are provided by the ID attribute discussed in Section 5.2. By assigning unique IDs to every patch, with the ID being the same for every triangle vertex within a patch, the picking system can be used to distinguish patches. The Terrain class' getPatchLocation() private method makes use of the picking system.

For selecting virtual patches, the picking system is not utilized. As an optimization, an intersection is calculated between a ray emanating from the camera in the direction of the mouse position and the z=0 plane. The location of this intersection is then used to compute which virtual patches are within the selection made by the user. The getZeroPlaneIntersect() private method handles the intersection calculation.

6 An Erosion Model Using Our New Features

As a concrete example of a model that could not be implemented in *StarLogo Nova* before, but can now be implemented, we revisit the erosion due to runoff example from Section 1.3.1. The Terrain can be used to model the mountain, the visualization of the Terrain is already suitable. The shape of the mountain determines the starting conditions of the erosion simulation, and the Terrain Editor makes it easy for the user to configure the starting conditions. Agents can be used to represent rain that falls and then becomes runoff once their altitude reaches zero. Runoff moving over the surface of the Terrain will automatically follow the shape of the Terrain with the changes made to the movement system. In a discrete erosion model, the volume underneath the Terrain can represent the remaining mass of the mountain. The stomp grid command can be used by runoff agents to erode the mountain and the yank grid command can be used by runoff to deposit sediment on the mountain elsewhere. The volume preservation property of yank grid and stomp grid automatically enforce conservation of mass. The data stores of each patch can store a firmness value determining how easily the patch is eroded, which can inform runoff agents how much to erode the Terrain using the get attribute command (to initialize the values, a common design practice of *StarLogo TNG* users is to do procedural Terrain editing by creating temporary agents; a similar method can be used to initialize the data stores using the set attribute command). In a continuous model, the Terrain height at each patch point can represent the remaining mass of the mountain, allowing us to use yank and stomp at locations between patch centers and still

achieve conservation of mass. The firmness of the Terrain can be determined at locations between patch centers using the interpolate attribute command.

7 Conclusion

StarLogo Nova is primarily designed to be a tool for computational modeling used by secondary school students in a classroom setting. However, despite the flexibility of *StarLogo Nova's* agent-based design, there are computational models that cannot be implemented or are difficult to implement in *StarLogo Nova*. Computational models that model an environment are particularly difficult to implement using only agents in *StarLogo Nova*. The previous representation of the environment in *StarLogo Nova* is simply a visualization of ground level called the Terrain that is mostly unaffected by simulations created by the user, and has little capability to interact with agents. We have expanded the set of computational models that can be implemented in *StarLogo Nova* by adding to the representation of the Terrain and creating new features. These features facilitate many kinds of interactions between agents and their environment that can be defined by the user programmatically. The state of the Terrain can also be configured by the user through the Terrain Editor, allowing the user to have control over the starting conditions of a simulation that makes use of interaction between agents and the Terrain. With the combination of new programmable interactions between agents and the Terrain and the addition of the Terrain Editor, the set of computational models possible within *StarLogo Nova* is expanded, and it is much easier to create computational models that model an environment.

8 Future Work

In this section we discuss ways in which the work of this thesis can either be extended or improved.

8.1 User Interface

The focus of this thesis was to add new functionalities to *StarLogo Nova* rather than to come up with successful user interface for utilizing those features. However, the creation of a minimal user interface was necessary in order for some of the new functionalities usable, namely the Terrain Editor. A good user interface design could constitute the work of another thesis in and of itself. This section serves as a starting point for future work on the user interface of the Terrain Editor, indicating what work is most critical for the Terrain Editor to be usable in a release version of *StarLogo Nova*.

Currently, the Terrain Editor only supports a user interface via mouse and keyboard. However, the current release version of *StarLogo Nova* is supported on mobile devices. Schools that use tablet devices for classroom learning are one target audience that requires support for mobile devices. A separate Terrain Editor user interface must be designed from the ground up with tools that are friendly to mobile devices in order for the new Terrain Editor to be available to all of *StarLogo Nova's* target audience.

The current Terrain Editor was designed to require minimal changes to the visual layout of *StarLogo Nova's* user interface and to be quick to implement. It was also designed to be used for debugging purposes when implementing commands. For these reasons, there is a lack of user-friendly, on-screen buttons for activating tools. There is also no minimal communication with the user about the state of the Terrain Editor, such as what tools are active. Instead, the user interface consists mostly of keyboard shortcuts and mouse operations that are opaque to the user. Users cannot learn how to use the Terrain Editor on their own by experimentation and must be taught the specific controls and memorize them. Further work can be done so that *StarLogo Nova* users can easily pick up on how to use the Terrain Editor through experimentation on their own.

Another drawback of the Terrain Editor is that it was designed solely using a conventional computer mouse and keyboard. As a result, user testing has indicated that the user interface can be hard to use on a laptop with only a trackpad available. Perhaps modifications to the tools can make them easier to use with a trackpad, or a separate control scheme can be devised for trackpad users.

8.2 Agent Movement

Considering that agents in *StarLogo Nova* have previously always been free to move in three dimensions, the interaction between agent movement and Terrain presented in this thesis is perhaps the simplest next step. Currently, agents only interact with floor patches. There is no interaction between wall patches and agents. The original intention of wall patches was that they could act as walls that could block or inhibit an agent's movement. The precise behavior of an agent whose movement is inhibited (by a wall patch or other obstacle) has yet to be decided. Perhaps the behavior could be programmable, as is the case with collisions between pairs of agents, or a predefined behavior could be implemented.

A consequence of the new movement system for agents is that, once the Terrain is uneven, the forward block can no longer move an agent parallel to the xy-plane or in a straight line as it used to behave. Sometimes this kind of movement can be desirable, and the agent should ignore the geometry of the Terrain when moving. A new property of agents could be implemented that determines whether the agent's movement is affected by the geometry of the Terrain or not. New movement blocks that ignore the Terrain could also be implemented.

8.3 Additional Features

Currently, the Terrain is completely reset every time a project is opened or close, or the user refreshes the page. This means that any work done by the user in the Terrain Editor is lost between user sessions, as well as any changes to the Terrain made by agents during a simulation. A method of saving

the state of the Terrain is required in order to keep the state of the Terrain persistent between user sessions. Persistent Terrain state will most likely be a requirement for a release version of *StarLogo Nova* featuring the Terrain Editor.

The simulation environment of *StarLogo Nova* is bounded, and the current behavior of agents that reach a boundary is for their movement to be redirected so that they do not escape the boundary. However, some kinds of simulations cannot be conducted in an environment with boundaries, or are influenced by the presence of boundaries, which can lead to inaccuracies. A popular example of a simulation affected by the presence of boundaries is Conway's Game of Life [Adamatzky 2010]. An environment without boundaries can be achieved by changing the topology of the three-dimensional space. If the environment is a bounded cube, as is the case with *StarLogo Nova*, then the topology of that cube can be changed so that faces opposite one another are connected. The two-dimensional equivalent of this is to take a square and connect the opposite sides so that it forms a torus. This change in topology would create an "asteroids effect", where an agent passing the boundary at one side of the cube would emerge from the oppose side. From the agent's perspective it can travel infinitely in any direction, emulating an infinitely large environment in a limited space. Implementing this would require changes to many systems within *StarLogo*. In particular, agent movement and other behaviors near a boundary would have to take into account the topological change. The Terrain Editor would also have to take into account the topological change when the selection reaches past the boundaries of the Terrain.

StarLogo Nova has an existing painting system where agents can draw on the Terrain in different colors. The work of this thesis preserves the painting system by allowing agents to draw on floor patches beneath them. Wall patches present a portion of the Terrain's surface that cannot be drawn on by agents when, previously, agents could draw on any portion of the Terrain. A system for drawing on wall patches is another possible extension of the work of this thesis.

9 Acknowledgements

I would like to thank my thesis supervisor, Eric Klopfer, and my direct supervisor, Daniel Wendel, for the opportunity to work on this thesis with them and for all their guidance, feedback, and support. I would also like to thank all the members of the MIT Scheller Teacher Education Program, who have always made me feel like a welcomed member of the lab. It is an honor and a privilege to have a small part in the *StarLogo* legacy at MIT. Finally, I thank my family for supporting me through my studies at MIT during the completion of this thesis.

10 References

Adamatzky, Andrew. Game of life cellular automata. Vol. 1. London: Springer, 2010.

Bau, David, et al. "Learnable programming: blocks and beyond." *arXiv preprint arXiv:1705.09413* (2017).

Foley, James D., et al. Introduction to computer graphics. Vol. 55. Reading: Addison-Wesley, 1994.

Lee, Irene, Fred Martin, and Katie Apone. "Integrating computational thinking across the K--8 curriculum." Acm Inroads 5.4 (2014): 64-71.

Melnik, Roderick. Mathematical and Computational Modeling. New Jersey: Wiley, 2015.

Parisi, Tony. WebGL: up and running. " O'Reilly Media, Inc.", 2012.

Press, W. H., et al. "Numerical Recipes." Cambridge University Press, New York (1992).

Appendix: Terrain Editor Keyboard and Mouse Inputs

General

- The Terrain Editor is activated by pressing the “Edit Terrain” button above Spaceland. Pressing “Lock Terrain” deactivates the Terrain Editor.
- While the Terrain Editor is active, a grid appears over the Terrain indicating the Terrain’s patches. Patches are the elements of the Terrain which you can edit. Left click and drag to select

a rectangular region of patches to edit. Then either scroll with the mouse or use a right click depending on the current tool to make edits to the selected region.

Tools

- *Raise*: Press 1 on the keyboard to make Raise the active tool. While Raise is active, scroll up to uniformly increase the height of the selected patches and scroll down to uniformly lower their height. Walls are created around the edge of the selection. The user may continue to use the left mouse button to make selections while Raise is active.
- *Mound*: Press 2 on the keyboard to make Mound the active tool. While Mound is active, scroll up to increase the height of the selected patches in the shape of a curved mound. Scroll down to decrease the height of the selected patches in the shape of a pit. Hold down on the right mouse button while moving the mouse left or right to change the curve of the mound/pit to be wider or skinnier. The user may continue to use the left mouse button to make selections while Mound is active.
- *Level*: Press 3 on the keyboard to make Level the active tool. While Level is active, right click on any terrain patch to change the height of the selected patches to equal the height of the patch that was right-clicked. Scroll the mouse wheel to flatten all selected patches, then continue scrolling to Raise them. The user may continue to use the left mouse button to make selections while Level is active.
- *Close Seams*: Press 4 on the keyboard to make Close Seams the active tool. While Close Seams is active, right-click anywhere to remove any walls within or around the selected patches by connecting the patch edges along walls. The user may continue to use the left mouse button to make selections while Close Seams is active.
- *Ramp*: Press 5 on the keyboard to make Ramp the active tool. While Ramp is active, right-click to the left, right, above, or below the selected patches to create a ramp in that direction. The height of the selected patches are automatically adjusted to create a smooth transition from the height on one side to the height on the other in the direction of the ramp. The user may continue to use the left mouse button to make selections while Ramp is active.
- *Copy*: Press 6 on the keyboard to make Copy the active tool. While Copy is active, right-click a reference patch within the selected patches. Then right-click again on any terrain patch to copy the selected patches at that location, matching the reference patch to where you clicked. The user may continue to use the left mouse button to make selections while Copy is active.

- *Move*: Press 7 on the keyboard to make Move the active tool. Move is identical to copy, except that the selected patches are set to height zero before copying.
- *Stretch*: Press 8 on the keyboard to make Stretch the active tool. While Stretch is active, hold down the right mouse button and drag to make the terrain patches bigger or smaller in size. The user cannot use the left mouse button to make selections while Stretch is active.
- *Grow*: Press 9 on the keyboard to make Grow the active tool. While Grow is active, hold down the right mouse button and drag to increase or decrease the number of patches in the terrain. The user cannot use the left mouse button to make selections while Grow is active.

Special

- Clear Selection and Tool: Press esc on the keyboard to unselect all patches and deactivate all tools.
- Reset: Press ctrl-r to reset the selected patches to height zero. If the Stretch tool is active, all patches become size 1. If the Grow tool is active, the terrain resizes to 101 by 101 patches.
- Anchor: Hold down the 'a' button on the keyboard to keep the same starting patch while making selections with the left mouse button.
- Trim: Press the 't' button to trim the selection to the boundaries of the terrain (remove out of bounds area from selection).
- Ground Level: Hold down the 'g' button to show a visual guide indicating where height zero is.
- Undo: Press ctrl-z to undo the last edit to the terrain.
- Redo: Press ctrl-y to redo the last edit to the terrain.
- Camera Control: Hold down the shift key on the keyboard to control the camera with the mouse instead of editing.

Camera

- Scrolling up zooms the camera in, scrolling down zooms out.
- While holding down the left mouse button, moving the mouse left, right, up, or down pans the camera in the same direction.
- While holding down the right mouse button, moving the mouse left, right, up, or down rotates the camera around its focus in the same direction.