StarLogo Nova Say

An Interactive Qualifying Project Proposal
Submitted to the Faculty of WORCESTER POLYTECHNIC INSTITUTE
In partial fulfillment of the requirements for the
Degree of Bachelor of Science
By Andrew Walter

April 26, 2018
Abstract

StarLogo Nova is an agent-based modelling tool that has been used to great effect in educational curricula. However, it is missing features of its predecessor. One of the missing features, the Say block debugging tool, was implemented. A study was performed to determine whether the new Say block was better than existing debugging tools in StarLogo Nova. The study showed that students are averse to using any debugging tool without training. To this end, several tutorials were written.
Acknowledgements

This project would not have been possible without the help of many people. I would especially like to thank the following:

My advisor, Professor Ryder, for putting many hours into this project, giving me access to her BB3010 course, and providing enthusiasm and positivity even when progress was slow. This project truly could not have been done without her.

Daniel Wendel, the StarLogo Nova Project Manager / main developer, for providing a great deal of technical help while I was developing the Say block implementation.

The students who participated in the Say study. When the study did not go as well as I had planned, the students pressed on and helped me salvage as much value as I could from the study.

The students in Professor Ryder’s D term BB3010 Simulation in Biology course, for welcoming me into their classroom and listening to my short debugging tool primer.

WPI’s Academic and Research Computing team, who set up a server for me to host my modified version of StarLogo Nova on campus.
Table of Contents

Table of Contents 4
List of Figures 6
List of Tables 7
1. Introduction 8
2. Background 10
   2.1. Modelling 10
   2.2. Logo to StarLogo Nova 11
   2.3. Debugging 13
3. Methodology and Results 16
   3.1. The Say Block 16
      3.1.1. Prior Work 16
      3.1.2. Design 18
      3.1.3. Text Rendering 19
      3.1.4. Architecture / Integration with StarLogo Nova 22
      3.1.5. Performance 24
   3.2. Say Block User Study 25
      3.2.1. Original Study Design 25
      3.2.2. Implemented Study Design 26
      3.2.3. Results 28
      3.2.4. Study Conclusions 31
   3.3. Tutorials 31
      3.3.1. Procedures 31
      3.3.2. Membrane 32
      3.3.3. What to Model? 33
      3.3.4. Debugging 34
3. Conclusions and Future Directions 35
Appendix A: Study Design Documentation 37
Appendix B: Tutorials 41
   Procedures Tutorial 41
   Membrane Tutorial 52
List of Figures

Figure 1: Scratch (left) vs StarLogo Nova (right) .................................................. 12
Figure 2: The Whyline in action ........................................................................... 14
Figure 3: Example of Say in StarLogo TNG .............................................................. 16
Figure 4: Billboard of text bubbles in StarLogo TNG. (note that the camera’s orientation is different than in the previous figure, but the labels are still facing the camera) ............ 17
Figure 5: Multi-line text bubbles, and combinations of text and values. .................... 17
Figure 6: Print to JS Console Example: ................................................................. 18
Figure 7: The letter ‘e’: ......................................................................................... 20
Figure 8: The letter ‘a’, in the Arial font: ............................................................... 20
Figure 9: From left to right, alpha-blended upscaling, alpha-tested upscaling, and SDF rendering ........................................................................................................ 21
Figure 10: Single channel distance field (left) versus multi-channel distance field (right) for the glyph ‘M’ in the Arial font. Texture size is 32x32 pixels. Generated using msdfgen (Chlumský 2017). .................................................. 22
Figure 11: Letters rendered with SDF (top) vs MSDF (bottom). ............................... 22
Figure 12: Example of the Say block .................................................................. 30
Figure 13: Average number of bugs fixed for each group and model ..................... 30
Figure 14: Graphs of proportion of bugs fixed for each of the study models ......... 30
Figure 15: Graphs of the performance of the two study groups ......................... 30
Figure 16: Agent trajectories after colliding with a wall: .................................... 33
List of Tables

Table 1: Constraints on the Say implementation, as well as critical requirements and optional features ......................................................................................................................... 18
Table 2: Planned study groups ........................................................................................................................................................................................................... 26
Table 3: Implemented study groups ................................................................................................................................................................................................... 27
1. Introduction

Agent-based modeling (ABM) is a paradigm of modeling that focuses on simulating autonomous entities inside of a system, as opposed to defining the behavior of the entire system. By giving each agent a simple set of rules, a model can be created that exhibits complex behavior. One of the major attractions of ABM is the relative ease with which users are able to model systems that would require more complex mathematics if they were simulated on the population level versus on the individual level. This opens up the possibility of teaching younger students – those without knowledge of differential equations, which would otherwise be required in many cases – how to create agent-based models.

Agent-based modelling software has been used to great effect in educational curricula from elementary school through college. StarLogo Nova is a relatively new agent-based modelling tool that runs in a web browser, enabling it to be used on tablets and inexpensive computers like Chromebooks, in addition to desktop computers. However, StarLogo Nova is missing several features that its predecessor, StarLogo The Next Generation (TNG), had.

Among these missing features is the Say block - a feature that enables the user to have agents display some information next to them. This can be used to show a particular part of the internal state of each agent, which is particularly useful when trying to understand or debug a model. For example, if a model contains a lion agent that behaves differently depending on how hungry it is, the Say block could be used to print out each lion’s current hunger level.

The major goal of this IQP was to implement the Say block in StarLogo Nova, and assess whether it helps novice StarLogo Nova programmers find and fix bugs in models. The data collected in the user study undertaken during this project will help the developers of StarLogo Nova focus their resources on areas of that tool that will most benefit users, as well as help Professor Ryder assess the debugging portion of her BB3010 curriculum. The implementation of the Say block produced during the course of this project may be merged into the official version of StarLogo Nova, giving all users of the tool access to the feature.

In addition to implementing the Say block in StarLogo Nova, several StarLogo Nova tutorials were produced during this IQP. Even college-level StarLogo Nova learners tend to be averse to using any feature of StarLogo Nova without having run through at least a single tutorial teaching it.

To assess the effect of the Say block on debugging, a user study was performed on some of Professor Ryder’s BB3010 students. The study participants were asked to debug two example buggy models, one after the other, and were asked about their approach. Additionally, screen recordings of each participant’s computer were captured, and the website used during the test was modified such that it would log information about the participant’s actions.
We found that users require a more substantial introduction to how to use Say, or even the existing Print to JS Console block, before they are willing to use it, even if it is already set up inside a model. This implies that more debugging tutorials and teaching are needed to ensure that students are able to debug their models.
2. Background

As computers have become more and more integrated into the everyday lives - and jobs - of nearly every American, there have been an increasing number of calls for the recognition of computer literacy as a core competency that should be taught in schools. While computer science courses have slowly been making their way into high schools, a complementary approach is to integrate computation into science courses. Wilensky, Brady et al. (2014) advocate for this approach, saying that it brings many distinct advantages over traditional computer science courses:

... integrating computation into science learning... increases access to computing for all students in all schools; ... enhances students' motivation for and depth of understanding of scientific principles by using computing in powerful ways; ... brings science education in line with authentic scientific practice and the needs of 21st-century science; and ... provides students with experiences of computers beyond searching and sorting, demonstrating the power of computation to help them make sense of their world.

That is, science classes provide an entry point for introducing computer literacy education into schools that may not have the resources to run standalone computer science courses, and computation helps students both be more prepared for real-world scientific techniques, and improves student understanding of scientific concepts.

2.1. Modelling

Essentially, all models are wrong, but some are useful.

– George E. P. Box

Models have been used in science for a long time – from the Bohr model of an atom to models of the Earth’s atmosphere to the periodic table, they have been crucial in enabling scientific progress. Indeed, there is “broad agreement that in fact much of science is model-based” (Kuorikoski and Marchionni 2015). However, as Kuorikoski and Marchionni go on to say, “there is still little agreement on pretty much anything else”. There is disagreement on what the definition of a model is, and to complicate matters, the ideas associated with the term “model” have changed over time (Hon and Goldstein 2012).

Across two studies in which U.S. and U.K. scientists were asked about the definition of a model (Bailer-Jones 2002, Schwartz and Lederman 2005), some general consensus can be found. A model is a representation or explanation of the real world that can be empirically tested. A model doesn’t have to consist solely of mathematical equations. A model is a simplification of a complex system in the real world, distilling the most important elements of a system. This is for practical reasons - modelling the full complexity of the system would take too much time, require too much data, or require a greater understanding of the system than is currently available. Furthermore, the simplified system provided by the model is often easier for humans to
understand than the original system, enabling deeper understanding of a concept. However, the simplifications and omissions that go into a model often limit the situations in which the model can function.

For our purposes, we will use the definition provided by Galán et al.: “Modelling is the process of building an abstraction of a system for a specific purpose.”

Agent-based modeling (ABM) is a particular paradigm of modeling that focuses on simulating autonomous entities inside of a system, as opposed to defining the behavior of the entire system (or large sections of the system). Classes of agents are each given a set of rules, as well as interactions with other agents. Even when giving each agent a relatively simple set of rules and interactions, often a system is created that exhibits complex behavior. (this complex behavior derived from simple rules is called emergent behavior) An example of emergent behavior is flocking - using only three rules, a group of agents can be made to move together like a flock of birds, or a school of fish.

ABM’s focus on agents makes it a natural fit for many of the systems studied in sociology and ecology, since many people find it intuitive to think of the individual agents making choices in such systems. This is opposed to aggregate-based modelling, where the behavior of groups of people - which tends to be less intuitive - must be modelled. Wilensky indeed found that, in several case studies with students, StarLogo’s agent-based design was easier for students to understand than models of overall population:

“Thinking in terms of individual creatures seems far more intuitive [than thinking in terms of overall population], particularly for the mathematically uninitiated. Students can imagine themselves as individual turtles/creatures and think about what they might do.” (Wilensky and Resnick 1999)

ABM is particularly useful in science education because models are able to concretely show the connection between the actions of a part of a system - for example, an atom, or an organism - and the behavior that emerges in the system as a whole. As Wilensky, Brady et al. (2014) say, these connections between “micro and macro aspects of scientific phenomena” are integral to many of the scientific concepts that students find the most challenging. As Klopfer, Scheintaub et al. (2009) have found, as opposed to aggregate-based modelling, ABM is “readily adopted by novice modelers”, making it more appropriate for primary and secondary-school students.

2.2. Logo to StarLogo Nova

*If I have seen further it is by standing on the shoulders of giants.*

-- Isaac Newton

StarLogo Nova (SLNova) is an agent-based modeling tool developed by MIT’s Scheller Teacher Education Program. It provides a block-based visual programming environment, similar to that
provided by Scratch (see Figure 1). SLNova can trace its lineage back through a number of educational tools, going back to Logo.

![Figure 1: Scratch (left) vs StarLogo Nova (right)](http://www.slnova.org/s-ebnerz/projects/266440)

Logo was developed in 1967 as a tool for teaching mathematical thinking. It is a text-based language designed without the “difficult technical features ... found in traditional programming languages (e.g., loops, counters, array declarations, multiple mode arithmetic, etc.).” (Feurzeig 1969) It is best known for Turtle graphics, an addition to the language which enables users to control the movement of a “turtle” robot, and use its built-in pen to draw patterns on the screen. The Turtle graphics feature was added by Seymour Papert in the late 1960s to support the control of a physical turtle robot (Berry 2015).

StarLogo was originally developed as an extension to Logo by Michael Resnick at the MIT Media Lab in 1994. Its major innovation over Logo was a focus on modeling decentralized systems, in which agents ran in parallel (Resnick 1996). It also gave turtles better sensing abilities for detecting other turtles, enabled larger numbers of turtles, and allowed the grid squares on which the turtles move to store additional information.

Through much research, including case-studies of using StarLogo in K-12 classrooms, the StarLogo developers found that having to teach a text-based programming language was a major
barrier for getting StarLogo into classrooms. Particularly, “text-based syntax errors seemed to be a significant barrier to Logo implementation by teachers” (Wang, McCaffrey et al. 2006). Based on this research, StarLogo TNG (The Next Generation) was developed by MIT’s Scheller Teacher Education Program (STEP) and released in 2006 as an upgrade to StarLogo. It offers many of the same features as StarLogo, but provides a visual programming language (StarLogoBlocks) as well as 3D graphics. Wang et al. [2006] found that StarLogo TNG had a lower barrier to entry than StarLogo, with students getting started more quickly and more easily recalling previously learned blocks.

StarLogo Nova is a web-based rewrite of StarLogo TNG, offering a comparable feature set, along with an online gallery of models. Users access StarLogo Nova through their web browser, and all of their models are stored online.

2.3. Debugging

*Only half of programming is coding. The other 90% is debugging.*

– Unknown

Debugging is a critical aspect of programming - a 2013 study conducted by Britton, Jeng et al. at the University of Cambridge Judge Business School found that programmers spent 49.9% of their programming time debugging. Thus, having high-quality debugging tools available significantly improves a programmer’s quality of life - and performance - when using a programming language. Text-based programming languages tend to have high-quality debugging tools, thanks to many decades of use and refinement. Visual programming languages don’t have the same amount of history to draw from, and as such the debugging tools for visual languages tend to be less well developed than those for text-based languages.

Ko and Myers (2004) discuss a new paradigm of debugging that they call interrogative debugging (ID). ID has some properties that make it useful for ABM. ID is based around helping programmers hypothesize about the reason behind a program failure, by enabling programmers to ask questions about why their program did (or didn’t do) something. Ko and Myers found that 50% of errors created by the Alice programmers they studied were caused by “false assumptions in the hypotheses they formed while debugging existing errors”. Thus, Ko and Myers designed their debugging tool (the Whyline) to assist users in creating and checking hypotheses. Taking an example from Ko and Myers’ paper, imagine that a user is developing a Pac-Man style game where Pac shrinks when the ghosts who are chasing Pac touch Pac. While testing, the user causes Pac to collide with a ghost, but notices that Pac does not shrink. This is due to another condition in the user’s code that they forgot about. The user can go to the Whyline and, from a menu, ask it to explain why Pac didn’t resize (see figure x). The Whyline will then explain that the condition in the if statement surrounding the resize block was not satisfied, and will further show exactly which part of the condition failed.
Agent-based models (ABMs) have some inherent properties that make them harder to debug than many other kinds of programs. Ropella, Railsback et al. (2002) discuss many differences between agent-based models and other kinds of programs, but the following three are the most relevant to debugging:

- Code flow in an ABM is usually nonlinear and parallel – that is, the code of many agents is often run (pseudo-)simultaneously, and one agent’s code can trigger that of other agents. Traditional debuggers are usually designed to debug non-concurrent, procedural code – that is, code that executes linearly from start to finish, except for procedure calls.

- The code for one kind of agent in an ABM is often spatially separate, and quite different, from the code for other kinds of agents. This, coupled with the ability of the code for one agent kind to trigger or affect that of another agent kind, makes for a complicated and inter-dependent system that is difficult to understand.

- ABMs often produce dynamic emergent behavior that is difficult to predict. While emergent behavior may be one of the attractive features of agent-based modeling, the fact that a system’s behavior is unpredictable makes it difficult to determine whether or not the system is functioning correctly.
Grimm (2002) expands on the last property above by noting that the process of debugging ABMs is necessarily more involved than the traditional debugging process, since traditional debugging only focuses on a small portion of the program, and assumes that the correct program output is known. When debugging an ABM, it is likely that the model’s “correct” behavior is not known, so an ABM debugging tool should help the user to understand the full model.

Grimm describes a process for debugging he calls “visual debugging”. Visual debugging does not only help a user locate and fix specific bugs in a model, but also helps the user test and analyze a model’s behavior. One of the elements of a visual debugging tool that he describes is the graphical representation of state variables of a simulation. This includes enabling the visualization of agent variables (i.e. agent age) and system variables (i.e. age distribution) but also includes visualizing intermediate state variables. These intermediate state variables aggregate agent variables in such a way that the emergence of behaviors on the system scale becomes more easily visible.

The Say block is a realization of a small part of Grimm’s visual debugging toolset. It allows users to visualize agent variables, and derivations thereof. By using the Say block, a user can gain a greater understanding of the behavior of the model – and thus be better able to detect and repair bugs in the model. For example, imagine that a user is working on a predator-prey simulation, and they find that the prey keep dying out. The user adds the Say block to their models to display agent energy, and finds that the prey have less energy than they thought, never gaining enough energy to reproduce. The user now has a better understanding of their model and has a better idea of where to look next: inside of the code that modifies an agent’s energy.

While debugging visual programming languages is hard, having Grimm’s visual debugging tools could help make debugging visual programming languages even more intuitive and powerful than debugging textual programming languages. Being able to see agent state right next to the agent itself is immensely useful for the user, and allows them to see patterns that might be less obvious when viewing the agent state separately from the agents themselves.
3. Methodology and Results

This IQP set out with three major goals: to implement a prototype of the Say block in StarLogo Nova, perform an experiment to see if it helps students find bugs in their models, and write tutorials to supplement those that are used in BB3010.

3.1. The Say Block

*For the things we have to learn before we can do them, we learn by doing them.*

--- Aristotle

3.1.1. Prior Work

The Say block was first made available in StarLogo TNG. This block allows an agent to show a small text bubble next to itself with some user-defined data (see Figure 3). This is useful as a debugging tool, since it lets the user see some of the agent’s state.

![Example of Say in StarLogo TNG](image)

*Figure 3: Example of Say in StarLogo TNG*

Note that the size of an agent’s text bubble varies based on the distance from that agent to the camera. This helps alleviate the problem of many agent text bubbles cluttering the screen, since labels far from the camera’s focus are de-emphasized. Text bubbles will always face the camera, a property which is called “billboarding” in computer graphics (see Figure 4).
StarLogo TNG also has support for string concatenation, enabling printing arbitrary combinations of text and values. Multi-line text bubbles are also supported, making it possible to display many values at once (see Figure 5).

The only existing debugging tool in StarLogo Nova is the “Print to JS Console” block. As its name suggests, this block allows users to print a value or string to the web browser’s JavaScript console (see Figure 6). This gives users the ability to print out values, similarly to Say, except that it is much less obvious which values are associated with each agent, even when the agent ID is printed out. Thus, “Print to JS Console” is more often used to determine if some portion of the code in a model has executed, rather than to print out values.
StarLogo Nova also has tools for graphing. While the graphing tools are not necessarily intended to be used for debugging, they can be still be quite useful for that task, especially when dealing with issues closer to model verification. For example, one might use a graph of average energy over time to determine that the average agent energy never increases - implying that perhaps the code that increments an agent’s energy when it eats something does not work. However, it is important to note that the graphing tools available in StarLogo Nova lend themselves more to monitoring aggregate quantities than individual quantities for each agent.

3.1.2. Design

Throughout the course of the IQP, it was determined that our Say block implementation should satisfy some constraints (see Table 1).

<table>
<thead>
<tr>
<th>Constraints on Say implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performant on low-end computers, and a variety of platforms</td>
</tr>
<tr>
<td>Simple to use</td>
</tr>
<tr>
<td>Has user-friendly defaults - e.g., automatically rounds numbers with many decimal places</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Critical requirements derived from constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Labels should be able to display plain text or the value of a block</td>
</tr>
</tbody>
</table>
Labels should follow their associated agent
Label rendering should be fast enough to keep up with the agent update rate
A reasonable number of labels should not significantly lower overall frame rate

Optional features
System should support non-English languages and glyphs
Labels should always be facing the camera
Labels should have a background and outline, to improve readability

The major challenge in implementing the Say block in StarLogo Nova was developing a method for rendering text in WebGL. WebGL is an implementation of the OpenGL graphics programming language for web browsers. Traditionally, applications that need to render text in OpenGL will use a library like GLUT (OpenGL Utility Toolkit) to handle most of the low-level text rendering details; however, no such library is available for WebGL at the moment.

At a high level, two different approaches were considered for the implementation of the Say block: rendering the labels for each agent in 2D using the web browser’s built in text rendering capabilities, or rendering the labels in the same WebGL view that the rest of the StarLogo Nova world is rendered in. Using the browser’s existing rendering capabilities would have been simpler than doing most of the rendering in WebGL, but also less performant.

StarLogo Nova currently uses the three.js library for interfacing with WebGL. Three.js provides a high-level interface for WebGL, making it easier to create, manipulate, and display highly custom 2D and 3D graphics in the web browser. However, the rendering component of three.js is not very flexible.

3.1.3. Text Rendering

Text rendering is a complex process, and many techniques have been developed for it. A glyph (a single character) inside a font file is generally stored as a series of lines and curves. To render the glyph to a computer screen, the glyph first must be rasterized - that is, it must be transformed into an array of pixels, also known as a bitmap (see figure x). This process is computationally expensive, so traditionally font renderers have rasterized all of the glyphs in a font into a “font atlas” (a “texture”, or group of bitmaps) once when the font is loaded, and then simply reused the bitmaps from the font atlas as needed.
The downside to the font atlas approach is that the texture produced is optimized for a certain font size and orientation. If the font atlas was generated for a font size of 12pt, rendering the font larger than 12pt will result in either blurry or pixelated characters (see figure 8). This is because a small glyph bitmap, stored in the font atlas, does not contain all of the information that a large glyph bitmap stores. Thus, when scaling a small glyph bitmap up to a larger size, (in particular, when the scale factor is not a whole number), the renderer has to guess the values for some of the pixels in the larger bitmap, a process known as interpolation. There are many interpolation algorithms, but two of the most common algorithms are nearest-neighbor and bilinear. Nearest-neighbor simply sets the color of a pixel in the new (scaled-up) image to the color of the pixel that it is closest to in the old image. This introduces a large amount of pixelation and jagged edges. Bilinear interpolation sets the color of a pixel in the new image to a combination of the colors of the pixels adjacent to it. This reduces the amount of pixelation, but blurs the image. Applying a threshold filter to the result of bilinear interpolation removes the blur, but causes the jagged edges to become bumpy.
In 2007, Valve’s Chris Green published a paper describing a method for improving the quality of scaled font atlas bitmaps, while being well-suited for running on a GPU. When using this method, the font atlas stores a signed distance field (SDF) for every glyph, as opposed to a normal rasterized bitmap for each glyph. That is to say, each pixel in the bitmap for a particular glyph in the font atlas has a “brightness” that is proportional to its distance from any part of the glyph.

The GPU renders a glyph by scaling the glyph’s distance field bitmap appropriately and then applying a threshold filter to it, turning pixels brighter than a certain threshold color white, and turning all other pixels black. The method is such that even low-end devices are capable of running it at high frame rates. This approach performs better than scaling traditional font atlas bitmaps, retaining smooth edges and curves without the usual aliasing artifacts. However, these improvements come at the expense of rounding off sharp corners (this is visible in Figure 11).

Valve’s original SDF font rendering method was extended by Viktor Chlumsky in his 2015 master’s thesis. Instead of using a single color channel to store the distance field (i.e. white to black), Chlumsky developed a method that uses multiple color channels to store several different distance fields for each glyph, which he calls MSDF. The extra data stored in the multiple color channels is used to reduce the corner rounding artifacts introduced by the SDF method.
Figure 10: Single channel distance field (left) versus multi-channel distance field (right) for the glyph ‘M’ in the Arial font. Texture size is 32x32 pixels. Generated using msdfgen (Chlumský 2017).

Figure 11: Letters rendered with SDF (top) vs MSDF (bottom). From Chlumský, Sloup et al. (2018)

There are other, more modern methods for high-performance font rendering, including some that rasterize glyphs on the GPU (as opposed to generating glyph bitmaps on the CPU and saving them) (Pranckevičius 2017). However, many of these methods are not amenable to use on old/low end devices, or are not trivial to port to WebGL.

The MSDF font rendering method is ideal for the Say block, since it combines high performance, high quality when scaled, and relative simplicity compared to other methods. Additionally, there are several libraries available for generating and rendering MSDF fonts, which reduced the amount of implementation work required.

3.1.4. Architecture / Integration with StarLogo Nova

StarLogo Nova is split into four major components: the world renderer (weblogo_threejs), the main server/website (wlsite), the compiler/runtime (weblogo), and the block specifications
Most of the work involving the Say block was done in weblogo_threejs, but some changes were also required in the other components as well to add the Say block.

The final implementation of label rendering for the Say block centers mainly around the files LabelManager, MSDFFont, and MSDFFontUtils. LabelManager is the high-level interface to the label rendering code. MSDFFont handles loading in MSDF textures and data files. MSDFFontUtil is responsible for doing layout for a string, determining the positions for each glyph inside a label.

The Viewport (which represents the view containing the StarLogo Nova world) is the main entrypoint into the label rendering code. When a user first loads a StarLogo Nova project in their browser, the Viewport creates a new LabelManager object. When the LabelManager object is created, it creates a MSDFFont object and asks it to load a default font from the server.

When a model is running, there are two kinds of events that the LabelManager cares about: state updates, and render calls. State updates occur every “tick” of the model; they indicate that the model has progressed one “tick” in time, and that some model data may have changed. Render calls occur every time the Viewport wants to update the world view. Render calls occur roughly
60 times every second (giving 60fps), while state updates occur much more infrequently (StarLogo Nova’s default is 5 times per second).

Every time a state update occurs, Viewport calls LabelManager, giving it the new world state. This world state includes information about each agent, including what they are saying (if anything), and their position. LabelManager calls MSDFFontUtil for each agent in the world state, telling it to layout the agent’s say string. After this has been done for every agent, LabelManager groups the layout for each agent’s say string together by glyph (i.e. all of the ‘a’ glyphs are grouped together into one list, as well as all of the ‘b’ glyphs in their own list...). LabelManager then stores all the information needed to draw each of the groups into a data structure that will be used later.

Every time a render call occurs, Viewport calls LabelManager. LabelManager iterates through each of the groups generated during the state update code, copying the information needed to draw each instance of each glyph into a buffer on the GPU. To reduce memory use, the LabelManager will reuse old buffers if they are big enough.

3.1.5. Performance

The current iteration of the Say block renderer is significantly faster than the previous iteration. In my initial implementation, as opposed to grouping together all of the glyphs of a given kind together into batches, the renderer rendered all of the labels together at once. This was slow because the process for merging all the labels into a single object was slow. Additionally, it meant that caching was difficult, because changing a single character inside an agent’s label invalidated all the cached data for that label. A lot of data was needlessly duplicated as well, since each instance of a glyph needed a copy of the glyph’s data (which is the same across all instances of that glyph).

The reason that batching is needed at all is that performing a draw call on the GPU has a lot of overhead, especially in WebGL. This means that it is desirable to perform as few draw calls as possible, to reduce the amount of time wasted getting a draw call ready. If each label had its own draw call, performance would be poor when large numbers of labels are visible. By batching by glyph kind, the number of draw calls needed to draw all the labels is proportional to the number of unique glyphs used in all the labels. Finally, batching by glyph kind enables the use of “instanced rendering”, which allows each instance of a glyph to share a single copy of the data that is the same between each instance. This translates into a significant reduction in memory usage.

Performance using the current implementation of the Say block is satisfactory. The renderer is able to maintain 60fps on a 2012-era Macbook Pro with 1000 agents each Saying something different.
Performance data shows that the main bottleneck of the label rendering is the generation of label glyph layouts. For a 1000-agent model, laying out the glyphs for text labels accounts for 56.1% of the total processing time. Presently, glyph layout is done using the layout-bmfont-text library (Jam3 2017). By writing a custom glyph layout library for Say, higher performance should be achievable. Additionally, the glyph layout cache parameters could be optimized to increase performance.

3.2. Say Block User Study

*You’re almost always wrong about your users.*

--- @ManiKrathee

To test whether the Say block is helpful for novice StarLogo Nova users in detecting bugs, a study was performed using students from Professor Ryder’s D-2018 BB3010 Simulation in Biology class.

The original study design was not exactly followed when the study was carried out, due to technical issues and a lower-than-expected number of participants. The original study design will be described below, followed by the slightly modified study design representing the study that was actually carried out. Additional documentation regarding the original study design and procedure is available in Appendix A: Study Design Documentation.

### 3.2.1. Original Study Design

The main goal of the study was to have participants debug two models: one with the ability to use Say, and one without.

The study was intended to take place on a Wednesday during BB3010’s 50-minute lab block. The students would receive instruction in basic debugging skills during the two days before the study. The day before the study, at the end of BB3010’s normal class period, the study would be described to the students, and students interested in participating in the study would be read the informed consent document, and asked to sign it. This would help increase the amount of time available during the study itself. Participants would also be shown how to use the screen-recording software that would be in use during the study.

The night before the study would take place, two AWS servers would be set up with the Say StarLogo Nova code. One of the servers would have the Say block enabled, while the other would have the Say block disabled. On the two study servers, anonymous StarLogo Nova accounts would be created for the students who indicated interest in participating in the study. A mapping would be created by the student investigator from real name to anonymous username. All data collected during the course of the study would be associated with anonymous usernames only.
Participants would be split into 4 groups based on which model they debugged first, and whether they had the Say block available during their first or second task (see Table 2). The groups would be assigned using block randomization, attempting to ensure that each group had a similar gender ratio. Varying whether participants started with access to the Say block was intended so that the effects of having previously debugged a model could be measured and eliminated. (i.e. participants might generally do better debugging the second model, just because they’ve gotten in the debugging mindset). Varying which model participants had the Say block available for was intended to enable detection of a difference in difficulty between debugging models A and B.

Table 2: Planned study groups

<table>
<thead>
<tr>
<th>Group</th>
<th>Task 1 Model</th>
<th>Say available during Task 1?</th>
<th>Task 2 Model</th>
<th>Say available during Task 2?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 1</td>
<td>Model A</td>
<td>No</td>
<td>Model B</td>
<td>Yes</td>
</tr>
<tr>
<td>Group 2</td>
<td>Model B</td>
<td>No</td>
<td>Model A</td>
<td>Yes</td>
</tr>
<tr>
<td>Group 3</td>
<td>Model A</td>
<td>Yes</td>
<td>Model B</td>
<td>No</td>
</tr>
<tr>
<td>Group 4</td>
<td>Model B</td>
<td>Yes</td>
<td>Model A</td>
<td>No</td>
</tr>
</tbody>
</table>

At the beginning of class on the day of the study, each participant would receive their anonymous username and password, as well as two post-task survey forms. The study investigators would check to ensure that all of the participants could log in using these credentials, and that they were able to run the screen-recording software used in the study. Participants would then be given 15 minutes to work on their first task. Participants’ computer screens would be recorded during that time. After the 15 minutes had elapsed, participants would be asked to stop working on the first task and spend 5 minutes filling out one of the post-task surveys. After the 5 minutes were up, the participants would be given another 15 and 5 minutes to work on their second task and fill out their second post-task survey.

Three forms of data would be collected during the course of the study: survey responses, screen recordings, and website logs.

3.2.2. Implemented Study Design

The day before the study, students watched a quick run-through of the use of the Say and Print to JS Console blocks. Students also were walked through the installation and use of the RecordRTC Chrome screen-recording extension (Khan 2018), as well as the consent agreement.

11 students participated in the study. During the study, participants were asked to debug two StarLogo Nova models. The participants did not have access to the Say block when they debugged the first model. For the second model, the participants debugged a model that either had Say blocks inserted in it, or Print to JS Console blocks. After debugging each model, the participants
were asked to fill out a survey that asked questions hoping to gather data about what techniques the participant used when debugging the model, as well as how well the student understood the model. Participants had their screens recorded during the study, and the recordings were collected at the conclusion of the study.

It was decided that adding the Print to JS Console block would be valuable because of the known issues with the Say block at the time the study was performed. It was hoped that the Print to JS Console block would provide a useful benchmark to compare against, and to see whether or not a known working debugging tool would improve student performance.

Due to technical issues, it was decided to omit the collection of website logs, since the information gathered in such logs would already be reflected in the screen recordings.

Due to the lower-than-expected turnout, to keep group sizes sufficiently large, participants were split into 2 groups (see Table 3).

**Table 3: Implemented study groups**

<table>
<thead>
<tr>
<th></th>
<th>Task 1 Model</th>
<th>Say or JS available during Task 1?</th>
<th>Task 2 Model</th>
<th>Say or JS available during Task 2?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 1</td>
<td>Model A ('Eco')</td>
<td>No</td>
<td>Model B ('Epi')</td>
<td>Print to JS Console available</td>
</tr>
<tr>
<td>Group 2</td>
<td>Model B ('Epi')</td>
<td>No</td>
<td>Model A ('Eco')</td>
<td>Say available</td>
</tr>
</tbody>
</table>

Instead of having some of the participants start with Say or Print to JS Console, all of the participants started without access to a debugging tool, and were given access to either Say or Print to JS Console during their second task. The version of Model A that was used in Task 2 contained already placed Say blocks, and the version of Model B that was used in Task 1 contained already placed Print to JS Console blocks. This was done due to concerns that the participants would not use either Say or Print to JS Console even if they had access to them, thus reducing the effectiveness of the study at assessing Say’s ability to assist users in debugging.

Both Model A and Model B were based on Project GUTS Debugging Challenge models (2016). Project GUTS provides these models alongside a set of questions to help teach students learning StarLogo Nova tools for debugging their models. The models were modified so that they would take a shorter amount of time to complete, and so that they tested users’ abilities to debug specific kinds of bugs that Say might be useful for. The models have 3 or 4 questions, with the first few being more straightforward bugs, and the last being slightly deeper, moving closer to model validation rather than bug fixing.
Model A (‘Eco’) was based on the Project GUTS Ecosystems Debugging Challenge, which involves a simulation of a simple ecosystem that contains grass, rabbits, and mountain lions. Both rabbits and mountain lions have an internal energy trait that is decremented by a small quantity every tick. When their energy drops to zero, the agent dies. The first bug involves an incorrect collision predicate for rabbits and grass: instead of rabbits increasing their energy and destroying the grass when they collide with grass, the rabbits increase their energy and destroy themselves. The second bug involves an incorrect nesting of create statements, which causes 2 mountain lions to be created every time a grass is created during setup, as opposed to 2 mountain lions being created once. The third bug involves an incorrect comparison inside of the mountain lion code that causes mountain lions to die if their energy was >= 0, as opposed to <= 0. The final bug involves rabbits incorrectly reproducing right after being born. This is because the range of energy levels that baby rabbits are randomly assigned at birth has an upper bound that is higher than the threshold for reproduction, meaning that some rabbits immediately have enough energy to reproduce after birth.

Model B (‘Epi’) was based on the Project GUTS Epidemic Debugging Challenge, which involves a simulation of a disease epidemic in a group of people. People can either be healthy, sick, or recovered. A healthy person has a chance of being infected when they collide with a sick person. A sick person has a chance of recovering every tick. Recovered people cannot be made sick when they collide with sick people. After 100 ticks of being recovered, a recovered person has a chance to lose their immunity and become a healthy person again (at which point they can become reinfected). The first bug involves an incorrect range set on the transmission rate parameter slider, causing the transmission rate to max out at a 10% chance. The second bug involves a procedure existing but never being called, resulting in sick people never recovering. The final bug involves an incorrect comparison being done inside the code that causes people to lose their immunity, resulting in recovered people that never return to the healthy state.

Due to issues encountered during the study, participants had 12 minutes to work on their first model but only 8 minutes to work on their second model.

3.2.3. Results

As stated above, there were 11 participants in the study. 10 of the participants were female, and one was male. However, one female participant did not complete the survey forms or produce screen recordings, meaning that there is only data for 10 participants. 4 of the participants were in Group 1, and 6 of the participants were in Group 2 (see figure x above).

Overall, the survey responses indicated that users generally did not find either the Say block or the Print to JS Console block useful for debugging the models. Most of the participants reported that they debugged the models using inspection (“looking at the model”) or experimentation (“playing around with the model”). Many participants specifically mentioned graphs as being
helpful when debugging. A few participants commented that the fact that the models heavily used procedures for organization was helpful for their understanding of the model.

Due to technical issues, the version of Say running on the study server did not have automatic rounding for numbers enabled, and had an issue where labels of dead agents would not disappear, causing the modified Model A (‘Eco’) to quickly become cluttered with labels. The two participants who referenced the Say block in their survey results both had a negative opinion of it - one of them thought it was a bug, and the other noted that it was difficult to see what was going on with Say enabled. This underscores the need for Say to have user-friendly defaults - if new users are faced with a version of Say that prints out huge, non-rounded numbers and does surprising things, they will be even less inclined to use it.

Reaction to the Print to JS Console block was mixed. One participant noted that having the console output available was helpful for them, while another said that it was not intuitive at all to use.

By analyzing the screen recordings, it was determined which bugs each study participant was able to fix. If the debugging tools were helpful, then groups using them should have been able to fix more bugs while working on the second task. If Say was more helpful than Print to JS Console, then the participants in Group 2 (with Say on Eco) should have performed better relative to their performance on the first task than the participants in Group 1 (with Print to JS Console). If one of the models was more difficult than the other, participants should have fixed fewer bugs in that model than the other model.

The participants of Group 1 (Epi for task 1) and Group 2 (Eco for task 1) performed similarly on their first task. Though appears that bugs 2 and 4 of Eco were more difficult than any of the other bugs in Eco or Epi, the average number of bugs that each group fixed for each model were roughly identical (see Figure 13). This suggests that the Epi and Eco models are similar in difficulty.

On average, the participants in Group 2 (with Say on Eco) seemed to do roughly as well (if not a little better) than the participants in Group 1 on Model B (‘Epi’) (see Figure 14). On the other hand, the participants in Group 1 (with Print to JS Console on Epi) seemed to differ quite a bit from the participants in Group 2 in their performance on the Ecosystem model. One Group 1 (with Print to JS Console) participant debugging the wrong model during Part 2, repeating work on Model B (‘Epi’). Thus, only 3 participants from Group 1 debugged the Ecosystem model during Part 2, possibly contributing to the variability of the data. These data indicate that Say and Print to JS Console were similar in helpfulness to participants.

Looking at the performance of each group between their first and second model (see Figure 15), Group 2 did slightly better on their second model (Epi) than their first model (Eco). Group 1 did roughly the same on their second model (Eco) as their first model (Epi), though none of them
were able to fix Bug 3 in the Ecosystem model. This indicates that neither Print to JS Console or Say were helpful to participants when debugging.

**Figure 13:** Average number of bugs fixed for each group and model

**Figure 14:** Graphs of proportion of bugs fixed for each of the study models
*Note that the Ecosystem model contained 4 bugs and the Epidemic model contained 3.*

**Figure 15:** Graphs of the performance of the two study groups
*Note that the Ecosystem model contained 4 bugs and the Epidemic model contained 3.*
3.2.4. Study Conclusions

The widespread use of inspection to debug models indicates that the study models were too simple, and that their bugs were too obvious. This simplicity allowed the students to debug the study models in a “brute force” manner – by reading the model’s code and looking for places it could cause bugs, rather than using a debugging tool to help them find the causes of bugs.

Including graphs may also have contributed to the ability of the students to debug the models without the use of the Say block. The graphs were carried over from the Project GUTS Debugging Challenge models on which the study models were based. It seems that the graphs were quite helpful to the participants, reflecting their utility. Removing the graphs might have forced the participants to experiment with other debugging tools, including perhaps the Say block. At the same time, it would be quite unrealistic to test the ability of users to debug without the use of the graphing tools in StarLogo Nova.

The biggest takeaway from the study is that the students are not inclined to use the debugging features of StarLogo Nova, even after basic instruction. Having both tutorials for Say and Print to JS Console, as well as an in-depth introduction to the process of debugging, would likely help increase the use of these debugging tools. The lack of improvement between debugging without and with a debugging tool, combined with the fact that several users either did not use, or disabled, the debugging blocks built into the Part 2 models, helps underscore the fact that students will avoid using a tool that they are not familiar with, even if it is already set up and included in a model.

3.3. Tutorials

*Recipes tell you nothing. Learning techniques is the key.*

-- Tom Colicchio

In addition to implementing the Say block, several tutorials were written over the course of this IQP. Though the StarLogo Nova developers, and Project GUTS, have produced an excellent set of tutorials and assignments for teachers of elementary through high-school students, more is needed to account for the higher level of complexity that is expected of models used to simulate biological systems at the college level. To this end, tutorials geared towards these more complex models were necessary, as well as tutorials for more robust debugging methods.

The full text of these tutorials is available in Appendix B: Tutorials.

3.3.1. Procedures

This tutorial is intended to teach users how to write procedures, and when creating procedures might be appropriate. This is even more important at the college level, since the models that the students are expected to produce are quite complex.
The learning goals for this tutorial are: Students will...

- Learn the syntax for creating procedures with and without parameters and return values
- Learn how to use procedures to organize model code
- Create a few example procedures to reinforce this material

The tutorial first introduces procedures as an organizational tool. Two versions of an example model are then shown: one written without procedures, and one written with procedures. The difference in readability between the two models is used as a motivator for the use of procedures for organizing code. The tutorial then points out a procedure in the model that takes a parameter, and uses it to explain how procedure parameters work in StarLogo Nova. Similarly, the tutorial next points out a procedure that returns a value, and uses it to explain how returning a value works in StarLogo Nova.

The tutorial then moves on to walk the reader through writing several procedures: a distance procedure that takes in parameters and returns a value, a purely organizational procedure that extracts code that is repeated in several places in a model, and a procedure that makes use of the “return early” block.

### 3.3.2. Membrane

One of the most common things students using the simulation for molecular and cellular biology need to do is have molecules interact appropriately with cell membranes. This tutorial is intended to teach users how to write a model that includes a simulated cell membrane, which is treated like a “wall” that agents bounce off of. The membrane is represented using terrain stamping, rather than agents, to eliminate the need for a workaround for an issue that can occur when an agent collides with more than one “wall” agent in a tick.

The learning goals for this tutorial are: Students will...

- Learn how to create simple geometric drawing using the terrain grid coloring system
- Learn how to create a simple membrane, which agents bounce off of
- Learn about potential pain points when creating more complex membrane systems

The tutorial first describes how to draw a simple box using terrain stamping, by having an agent follow a path and stamp the terrain under it each tick. This approach works well for basic polygons, but there isn’t a convenient way to terrain stamp an arbitrary pattern in StarLogo Nova anyways (unlike Netlogo, which allows users to import images onto the terrain). The tutorial then describes a simple method for having an agent deflect off a surface that it collided with, noting that the approach does not behave realistically (see figure x). Plans are in place to add an optional section at the end of the tutorial that describes the code for simulating deflection of agents after colliding with a wall.
3.3.3. What to Model?

This tutorial is intended to show users the thought process that goes into deciding what components and behaviors of a real-world system should be included in a model. It uses a model of the influence of DDT use on bird population as a case study.

The learning goals for this tutorial are: Students will...
- Learn to think about tradeoffs that occur in model design
- Learn some criteria for choosing certain behaviors to model or not to model

The tutorial starts with a light introduction to the effects of DDT on birds, and proceeds to create a model that does not include agent death or reproduction. This produces a model that is not useful for the purpose it was designed for, since it does not represent a critically important portion of bird population dynamics: that birds die if they get too old, or if they don’t eat enough. The tutorial then walks the user through the additions that are made to the model, to produce a model that more accurately simulates bird population versus DDT usage.

This tutorial was also intended to include a section about models with too much complexity, in which the case study model would have several extraneous agents added to it that do not significantly contribute to bird population. This section proved to be difficult to write, since how to answer the question of “is a model more complicated than it needs to be?” for some particular model is highly nuanced, and does not have a great deal of consensus. The desired complexity of a model depends on a variety of factors, including the complexity of the particular system being
modeled, the availability of computing resources, and the purpose for which the model is built. Generally, increasing the complexity of the model will increase its predictive ability. However, it does so at the cost of making the model slower to run, and making it harder for someone reading or modifying the model to understand it. Additionally, I found it difficult to come up with an example of an addition to the model that would make it needlessly complex that is not obviously unnecessary.

3.3.4. Debugging

The final tutorial that was planned for this IQP was a model on debugging. This tutorial has not been written yet. The design of this tutorial will be informed by the data gathered in the Say user study. Its purpose will be to teach users not only about debugging tools, but also about strategies and mindsets that can be helpful when debugging a model.

The learning goals for this tutorial are: Students will...

- Learn how to use the Print to JS Console and Say blocks
- Learn strategies for finding the source of a bug
- Learn strategies for model verification
4. Conclusions and Future Directions

This IQP produced a functional prototype of a Say block in StarLogo Nova. Additional work needs to be done before it can be integrated into the official StarLogo Nova server, but it is most of the way there.

The Say implementation developed during this IQP satisfies all of the critical requirements described in the Design section of this paper (Table 1). The Say block supports both displaying plain text and values. Labels follow the agent they are associated with. Label rendering is sufficiently fast to keep up with the agent update rate. Finally, rendering a reasonable number of labels does not cause a frame rate drop on the computers tested.

While none of the optional features described were completed, several of them have had at least initial design work and research completed:

- **System should support non-English languages and glyphs** - Support already exists for all languages. When generating the MSDF font file, the glyphs that should be included in it are manually specified. Thus, glyphs for any language may be included in the font file (assuming they exist in the font file from which the MSDF font file is generated). This system is not ideal for languages that have many glyphs (i.e. in Japanese and Chinese, 2000-3000 characters are considered a threshold for literacy (Tamaoka, Makioka et al. 2017)). To support such languages, it might be better to do the MSDF font file generation on the fly, either on the client or server. Additionally, it might be worth investigating newer font rendering techniques that can be done entirely on the GPU, without the need for distance field texture generation.

- **Labels should always be facing the camera** - A significant amount of time was spent trying to implement this, however my lack of computer graphics experience may have hindered me. This should be possible, by modifying the matrix used to transform vertices from world space to view space.

- **Labels should scale based on their distance from the camera, like in StarLogo TNG** - Labels should have a background and outline, to improve readability

The “What to Model?” tutorial was challenging to produce, and still needs some work. Future work on this tutorial should make more clear the thought process behind the choices of behaviors and agents to model, and assumptions to make.

Due to time constraints, the debugging tutorial did not get completed. Future work on this tutorial should address the data from the Say user study. In particular, the tutorial should include a detailed walkthrough of how to use both the Say block and the Print to JS Console block to discover information about a model, as well as how to interpret the output of both blocks.
Despite the disappointing results found during the study, I believe that the Say block can be an effective tool, both for understanding models and for debugging them. With additional instruction, students should become more comfortable with both Say and Print to JS Console, better preparing them to debug their own models, as well as models of others.
Appendix A: Study Design Documentation

The following was submitted to the WPI IRB as part of the IRB study approval process. It describes the procedures that were expected to be followed during the study.

StarLogo Nova Say Debugging Efficacy

Lead Student Investigator: Andrew Walter
Faculty Advisor: Elizabeth Ryder

Purpose

The purpose of this study is to determine whether or not the addition of the Say block to the StarLogo Nova visual agent-based modelling programming environment improves the ability of novice StarLogo Nova programmers to find and fix software bugs in models.

Study Protocol

The study will take place during a Wednesday class session of Professor Ryder’s BB3010 Simulation in Biology course, over a 50 minute time period. Students will have received instruction in some basic debugging skills during the previous two classes, including a short demonstration of debugging using the Say block. The student investigator will assist in this instruction. Students will receive information about the study on the Tuesday class session the day before the study occurs, and those who agree to participate in the study will fill out written consent forms then.

During the study, participants will be asked to debug two buggy StarLogo Nova models. Half of the participants will have access to the Say block when they debug the first model, and not when they are debugging the second model, and vice versa for the other half of the participants. After debugging each model, the participants will be asked to fill out a survey that gathers information about what techniques the participant used when debugging the model, as well as information about the student’s understanding of the model.

Procedure:

1. During class, the day before the study (a Tuesday):
   a. Introduce students to the goals and structure of the study. Inform students that should they wish to participate, the computer screens of the lab computers that they will be using will be recorded, and that the website will record their actions, for the duration of the experiment on Wednesday. Inform students that if they decide to participate, they are not required to answer every survey question if they are not comfortable doing so, and that they can end their participation in the study at any time.
b. Make it clear to the students that the purpose of the study is to determine the ways in which they go about debugging a model, and that it is not expected that they find every bug in the given models.

c. Inform the students that should they decide to participate in the study, they will receive a $10 Dunkin' Donuts gift card. Inform the students that they will still receive the gift card if they decide to cease participation during the course of the study.

d. Inform the students that their participation or non-participation in the study, as well as their performance during the study, will have no effect on their course grade.

e. Answer any questions that the students may have about the study.

f. Obtain written consent from any students that wish to participate in the study.

2. After class, the day before the study

a. Using the written consent forms to establish how many participants are expected to be involved in the study, create an account (random username and password) for each participant on the two different test StarLogo Nova servers. Create a mapping from participant names to random usernames, to be destroyed after the study is completed.

b. Use block randomization to split the expected participants into 4 approximately-equally sized groups, attempting to ensure the groups have similar compositions with respect to gender.

<table>
<thead>
<tr>
<th></th>
<th>Task 1 Model</th>
<th>Say available during Task 1?</th>
<th>Task 2 Model</th>
<th>Say available during Task 2?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 1</td>
<td>Model A</td>
<td>No</td>
<td>Model B</td>
<td>Yes</td>
</tr>
<tr>
<td>Group 2</td>
<td>Model B</td>
<td>No</td>
<td>Model A</td>
<td>Yes</td>
</tr>
<tr>
<td>Group 3</td>
<td>Model A</td>
<td>Yes</td>
<td>Model B</td>
<td>No</td>
</tr>
<tr>
<td>Group 4</td>
<td>Model B</td>
<td>Yes</td>
<td>Model A</td>
<td>No</td>
</tr>
</tbody>
</table>

3. During class, the day of the study (a Wednesday)

a. Provide each student with a piece of paper containing a unique username and password for the test StarLogo Nova server. Ensure that all participants can use these credentials to log into the test server, and that the screen recording software is correctly functioning on all computers. Reiterate that participants can end their participation in the study at any time, and that they do not need to answer any survey question if they are uncomfortable doing so. Reiterate that the participants are not expected to find every bug in the given models, and that the focus of the study is to explore their approach to finding and fixing bugs. (approx. 5 minutes)

b. Provide each participant with:
i. A task sheet, containing a description of Task 1, and a link to the appropriate model.

ii. A task sheet, containing a description of Task 2, and a link to the appropriate model.

iii. Two copies of the survey, pre-filled with the participant’s anonymous username.

*(approx. 3 minutes)*

c. Ask participants to start Task 1. Participants will be given 15 minutes to complete Task 1. If a participant completes Task 1 before the allotted time has elapsed, they are instructed to begin filling out the Task 1 survey. *(approx. 15 minutes)*

d. Ask participants to stop working on Task 1, and fill out the Task 1 survey if they have not already done so. *(approx. 5 minutes)*

e. Ask participants to start Task 2. Participants will be given 15 minutes to complete Task 2. If a participant completes Task 2 before the allotted time has elapsed, they are instructed to begin filling out the Task 2 survey. *(approx. 15 minutes)*

f. Ask participants to stop working on Task 2, and fill out the Task 2 survey if they have not already done so. *(approx. 5 minutes)*

g. Thank the participants for their cooperation and provide them with contact information if they have any questions about the study in the future. Distribute Dunkin’ Donuts gift cards. *(approx. 2 minutes)*

**Subject Information**

Participants will be recruited from Professor Ryder’s BB3010 *Simulation in Biology* course in D-term 2018. One Wednesday class session, and a portion of the class session the previous day, will be dedicated to performing the study. The course population is currently expected to consist of 25 undergraduate students of both genders, ranging in age from approximately 18 - 23. The students will be notified that participation in the study is optional and will not affect their grade in the course.

Participants will receive a $10 Dunkin’ Donuts gift card at the end of the study. They will receive this regardless of whether or not they decide to cease participation during the course of the study.

**Confidentiality and Anonymity**

Three forms of data will be collected during this study: screen recordings, website logs, and survey responses. Participants will identify themselves on the survey responses using the username given to them at the beginning of the study. Screen recordings, website logs, and survey responses will be linked together using the username.
Two Amazon EC2 servers will be rented for the duration of the study. The data on the servers will only identify the participants by their randomly generated usernames, which can only be mapped back to a particular participant using the mapping in possession of the student investigator. After the class ends on Wednesday, this data will be downloaded from the Amazon EC2 servers, and the Amazon EC2 servers will be shut down, and their storage volumes wiped.

A mapping from student name to username will be kept by the student investigator for the duration of the study.

Screen recordings will be acquired by the student investigator as soon as possible after the end of the study, and will be edited to remove any personally identifiable information. The original recordings will then be destroyed, and only the edited recordings will be kept.

Any participant data quoted in the project report will be identified anonymously or using a pseudonym.
Appendix B: Tutorials

Procedures Tutorial

Introduction

Procedures are an organizational tool provided in many programming languages to help separate large sections of code into smaller chunks of related code. Additionally, they allow the programmer to reuse code in multiple places without having to copy and paste it.

A procedure is defined in a procedure definition block. The code put inside this block is the code that will run when the procedure is called (used).

To use a procedure, you use a procedure call block. You can think of this block as essentially a shorthand for all the code inside the procedure definition block. When your program is run, the computer will replace the procedure call block with the code from the procedure definition block, and then run that code normally (this is a pretty big simplification).

In essence, procedures let you break down complicated code into simpler and smaller pieces.
For example, here’s a portion of code from [turtle without procs model].

There’s a lot going on in this code, and it’s hard to see which blocks go together to do something. In addition, there are so many blocks that they cannot all fit onto the screen at once, making it even harder for someone to mentally go through the code.
Here's the same code, rewritten to use procedures. The code inside the forever loop is much more concise, and it is now possible for someone to understand what the turtle does in the forever loop without understanding exactly how it does it. The reader doesn't have to get bogged down in the details of how `reprod_daily` works to understand that calling that procedure will cause the turtle to reproduce daily, possibly limited by some conditions.

Here are the procedures called by the main forever loop code. The reader can examine each one separately if he/she wants to know the details of what it does.
Now, let’s take a closer look at the speed procedure.

This procedure, and the twist procedure, both look a little different from the other procedures: they have extra sockets on their call blocks, and they have extra lines in their procedure definition blocks. This is because these procedures take parameters - that is, when you call them, you have to give them some extra information. Intuitively, this makes some sense - the speed procedure just tells the turtle to go forward by some amount, but how much should it go forward? Maybe in one place the programmer wants the turtle to move forward by 1 unit, and in another place by 2 units. Parameters let us tell the procedure how much to move forward each time it is called, without our having to write separate procedures to go forward by 1 unit and go forward by 2 units.

Writing procedures with parameters has a big advantage: you are much less likely to make mistakes when you have a single copy of code that does something, rather than multiple copies, each of which has some slight variation and is named something slightly different.

Another useful feature of procedures is that in addition to taking in parameters, they can return a value. Here’s an example of a procedure which returns a value:
This procedure returns the larger one of the two values that are given to it.
To use this procedure, we use the call block with the little connector on the left, instead of the connector on the top.

In the next exercise, you’ll build several working procedures.
Exercise 1: Distance

Let’s say we are simulating the behavior of a particular species of bird. Once this bird builds a nest, it will only travel a certain distance away from the nest while hunting. Once it reaches this distance from its nest, it returns to its nest.

To simulate this behavior, we need to be able to determine how far away a bird is from its nest. Each bird agent will store the x and y position of its nest, and can get its own x and y position. Thus, if we write a procedure that can calculate the distance between two (x, y) positions, we will be able to figure out how far the bird is from its nest.

We can use the distance formula to calculate the distance between two points:

\[ d = \sqrt{(x_{bird} - x_{nest})^2 + (y_{bird} - y_{nest})^2} \]

First, we have to create the procedure definition block. Drag a “procedure” block from the block palette into the “Everyone” tab.

Next, we have to name the procedure. It is important to choose names that describe the function’s purpose, to make it easier for others to read your models. In this example, I’ve named the procedure “distance”, but feel free to choose your own name.

Next, we have to add parameters. This is how the procedure will be able to take in data from other parts of a model. Parameters are added by clicking on the + icon to the left of “add parameter” in the procedure block.
By default StarLogo Nova will create parameters for data values, which is what we want in most cases (including this case). Parameters can also be set to take in only lists, or only agents, by clicking on “data” and selecting the appropriate option.

We need 4 parameters here: x1 and y1 for the first point, and x2 and y2 for the second point. Again, the exact names of the parameters can be anything, but the x1 notation is familiar to people using the distance formula. Once you’re done naming the first parameter, add the other 3 parameters and name them.

We also want our procedure to return a value - the distance. Click on “nothing” to the right of return, and select “data” from the dropdown menu that appears.

Now, we have to actually write the procedure. We could write out the procedure on a single line, but that line is very long and hard to read. It’s better to do parts of the calculation in steps, saving them into variables.
First, we need to calculate the difference between the x1 and x2 values, and the y1 and y2 values. In this example, I call these two values dx and dy. To do this, use the “var” block to create a new variable with the value of x1 - x2, and do the same thing for the y values.

Next, we have to calculate the value of $dx^2$ and $dy^2$. This can be calculated either by doing $dx \times dx$ and $dy \times dy$, or by using the power block. I used the power block, since it’s a little bit shorter.
Finally, we have to sum $dx^2$ and $dy^2$, and then take the square root of that value. This is short enough that we could probably put it directly into the return connector, but I created another variable for it.

Finally, now that we have the distance, we should return it.
procedure: distance

- x1 data
- y1 data
- x2 data
- y2 data

+ add parameter

var: dx data is parameter x1 - parameter x2

var: dy data is parameter y1 - parameter y2

var: dx^2 data is value of dx power 2

var: dy^2 data is value of dy power 2

var: distance data is sqrt value of dx^2 + value of dy^2

return value of distance data
Membrane Tutorial

Membranes

There are a few ways to simulate a membrane in StarLogo. One is to use agents to simulate the membrane, and use StarLogo’s built in collision blocks to detect when something collides with the membrane. Another is to use StarLogo’s terrain stamping system, and have the agents check whether they are standing on top of a position with a certain color.

In this tutorial, we will go with the terrain stamping approach. The agent-based approach has an annoying issue that causes agents to pass through the membrane if they hit two of the membrane agents at once. It’s possible to work around this, but it’s an annoying detail that complicates the code.

Drawing the membrane

The most annoying part about simulating any sort of terrain in StarLogo is creating it. This is an issue both with the agent-based approach and the terrain stamping approach. I’ve found that the easiest way to draw something using terrain stamping is to create an agent that moves around and stamps the terrain in the desired pattern.

The first step is to create a breed for drawing the membrane. You can use the default “Turtle” breed if you like, or create a new breed.

We are going to draw a box for the membrane. To do this, we create an agent at one of the corners of the box, and have it move forward by some amount, and then turn left by 90 degrees. It repeats this 4 times to generate the 4 sides of the box. After it is done drawing, it deletes itself. I recommend sticking this code in a procedure, since it will clutter up your setup block otherwise.
Additionally, you need to add a “clear terrain” block to the beginning of your setup block, either before or after the “delete everyone” block. This clears any stamps on the terrain.

Now, you should run your code and click the setup button. If everything worked right, you should see something like this:
NOTE: If you don’t see the blue box drawn on the terrain, it might help to click on the “Edit Interface” button at the top left corner of the screen and then click and drag on Spaceland (the green square). There is currently a bug in StarLogo that causes the stamps not to show up until the terrain is panned in this way.

Now that we’ve drawn the walls, we have to create a breed of agents which will interact with them.

Creating a new Breed

To add a breed, click on the “Add Breed” button to the right of the “World”, “Everyone”, and “Turtle” tabs, then type in the name you want for the new breed in the text box which appears. When you’re done, click somewhere else on the screen to finish the process.

I called my second breed “Molecule”, but you can call yours anything you want.
Creating a Molecule Setup procedure

After creating a new breed, the first thing I like to do is create a “breed setup” procedure. This procedure sets whatever properties that all agents from that breed share (for example, size and shape are often shared among all agents in a breed).

![Procedure example]

**Note:** Using this procedure to setup Molecule agents wherever they are created makes it much easier to change one of these properties in the future. In more complicated models, there can be several places in the code where an agent is created, meaning that there are several places where code needs to be changed if you decide you want the agent to be a different shape or color, or if you add a trait and need to set a default value for each agent when they are created.

Spawning Molecule agents

Now, we have to spawn the Molecule agents. I like to create another procedure, in the “The World” tab, for each breed, which creates all of the initial agents of that breed.
NOTE: Note that I set each of the X and Y of each Molecule agent to be a random number between -39 and 39. This is so that each Molecule agent is created inside the walls (which are at -40 and 40 in both X and Y). If you don’t set the x and y position of the Molecule agents here, they all will be created at the center of Spaceland (0,0).

Finally, put a call block for this function into the end of the setup block so that this procedure is called.

Now, if you Run Code and click on the setup button, you should see something like this:
Note: It can be hard to see the Molecule agents because they're so small. Additionally, if you don’t see the walls, try the trick described in the note at the end of “Drawing the Membrane”.

Molecule Wall Collisions

Now that we’ve drawn the wall, and created our Molecule agents, we can write the code to detect collisions between the Molecule agents and the wall. Since we’re using the terrain stamping approach, we don’t use the collide block. Instead, we can use the “terrain color” block to find out what color the ground is underneath each agent, and react appropriately if it’s a wall. As usual, I like to put this code in a new procedure:

```
procedure: wall collision
add parameter

if terrain color = color: blue
left by 180 degs

return nothing
```

NOTE: This doesn’t really do collisions in a way that looks (or is) correct. When a Molecule agent hits a wall, it is sent back the way it came. The Molecule agent should really be sent back at an angle to the surface it hits. Unfortunately, doing this is a little bit complicated, since you need to determine the angle of the Molecule agent with respect to the surface, which depends on whether the surface being hit is vertical or horizontal.
Finally, we have to run this code every tick, and make the Molecule agents move forward as well. (if you forget to make them move forward, your agents will just flip directions forever when they hit a wall).

Now, if you run the simulation, the Molecule agents should bounce off the walls!
What to Model Tutorial

What to model?

Unfortunately for us, the world is a complicated place. To perfectly model even a seemingly simple phenomenon - say, the water cycle - one would have to build a model with a massive number of variables and interactions. Such a model would run excessively slowly, and would be nearly impossible for a single person to understand.

To make a useful simulation, we have to take the phenomenon that we are simulating and distill it down to a model that is simple enough to be understandable and fast, but that is also complex enough to show whatever particular behavior we might be trying to show.

We'll show what we mean by this by building up a model of a system, showing which behaviors and agents we model and why.

What are we modelling?

The first step towards creating a model is certainly to determine what exactly you're building a model for. For this tutorial, we'll be building a model of DDT poisoning in birds. We first identify which agents should be involved in the model:

- Birds
- Bird eggs
- Earthworms
- DDT-sprayed areas

Earthworms in DDT-sprayed areas should gradually have DDT accumulate in them, and birds should have DDT accumulate in them when they eat earthworms containing DDT. DDT causes the shells of bird eggs to become thin, causing them to sometimes break under their own weight.

Let's start by creating these breeds, and creating some of each of these agents when the model is setup.
Now, let’s think about what data we need to store. Certainly Earthworms, Birds, and Bird Eggs must all store their DDT level, and DDT-sprayed areas should have their DDT concentration fall over time. We can do this by adding a “DDT level” trait to Everyone.

Now let's add the DDT concentration drop behavior to DDT-sprayed areas.

Now let’s add a behavior where Earthworms have their DDT concentration increase when they are near a DDT-sprayed area, as well as a basic “wiggle” motion.
Finally, let’s add Earthworm eating behavior to Birds, as well as a basic seeking movement.

Let’s also add a line graph for Bird and Earthworm population to the sim.

If we run the simulation for a while, we get a graph like this:
Clearly, this does not accurately simulate the real world - the Birds never die (even if they don’t eat) and neither Earthworms nor Birds reproduce. Given that the main effect of DDT on birds is shell thinning, our model should at least include Bird reproduction so that we can see this effect.

Let’s implement a simple reproduction system for both Birds and Earthworms.

For Earthworms, we’ll implement a population cap to ensure that the population doesn’t explode. It’s OK to take this “artificial” approach because we’re not simulating the lifecycle of Earthworms - we only care that there are enough for the Birds to live on.

For the Birds, we’ll implement a more realistic, but also more complicated system, that tracks the energy that each Bird has. When the energy rises above a certain threshold, the Bird will have a chance to produce an egg. The Bird’s energy will fall each tick, and the Bird will die if its energy
falls below 0. Finally, we’ll add an age trait to the Bird, so that it has a chance to die when it has been alive for more than 500 ticks.

We need to initialize Birds’ energy to a random value when they are created.
We also need to add logic to the Bird Egg so that it will hatch into a bird after a period of time, and so that it has a chance to die depending on its DDT level. We'll write a helper procedure here, lerp_clamp, that performs linear interpolation (turning a value in one range to a value in another range, clamping values outside the first range to the minimum and maximum value of the second range).

We'll use lerp_clamp to turn the Bird Egg's DDT level into a percent chance that it will die, checked once every tick.

Finally, we should test the model, to see if we're getting "reasonable" results given what we know about DDT.

The first thing to check is that the population dynamics seem reasonable when there are no DDT-sprayed areas.
This is what I got on my first attempt. The birds had a population increase that caused the worm population to fall too drastically, causing the rest of the birds to die of starvation. The reason this happens in our model is that there is a delay between birds reproducing, and the egg hatching. This means that even a bird reproduces when there are many worms in the world, when the egg hatches there is no guarantee that there are still worms left for it to eat. To moderate this effect, we can reduce the likelihood that a bird will reproduce, and increase the rate with which Earthworms spawn.

After these changes, I got something like the following graph:
This is better - at least the birds don’t all die out immediately - but could still be improved. We will leave the model the way it is for now, but feel free to come back and try to reduce the magnitude of the population oscillations.

Now let’s try with a large number of DDT-sprayed areas.

This seems pretty reasonable - the birds reproduce fairly normally at first, but as they begin to bioaccumulate DDT from worms, their eggs become less and less viable. Eventually, all of the birds begin producing eggs that are almost certain to die before they hatch, and the bird population dies out.
References


