

Performance Engineering of the StarLogo Nova Execution Engine

by

Jin Pan

Submitted to the Department of Electrical Engineering and Computer
Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2016

© Massachusetts Institute of Technology 2016. All rights reserved.

Author
Department of Electrical Engineering and Computer Science
August 16, 2016

Certified by
Eric Klopfer
Director of MIT Scheller Teacher Education Program
Thesis Supervisor

Accepted by
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Performance Engineering of the StarLogo Nova Execution Engine

by

Jin Pan

Submitted to the Department of Electrical Engineering and Computer Science
on August 16, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

StarLogo Nova is an online blocks-based programming environment designed for pre-college students to explore the collective behavior of decentralized agents. Users can drag and drop blocks to construct graphical scripts that control how individual agents respond to stimuli. These scripts are run by the Execution Engine and rendered in real-time. By investigating hypotheses about how small tweaks to individual behavior impact the entire system, students learn to think beyond the centralized mindset where all actions are dictated by a singular leader.

This thesis migrates the Execution Engine from the aging Adobe ActionScript 3 language to TypeScript, a weakly typed language that transpiles to JavaScript. To promote code health, this thesis introduces code formatters, linters, test cases, and a build process. Finally, this thesis optimizes the StarLogo Nova Execution Engine for performance, consistently beating the previous engine and bringing the execution time per cycle for key benchmarks under 10 milliseconds.

Thesis Supervisor: Eric Klopfer

Title: Director of MIT Scheller Teacher Education Program

Acknowledgments

To my parents, for all the difficult sacrifices they have made for my education. I hope that future StarLogo Nova users are able to indirectly benefit from your hard work.

I would like to thank all the teachers, mentors, and friends that I have met and learned immensely from, both inside the classroom and outside.

From that list of teachers, mentors, and friends, I would like to specifically Daniel Wendel and Eric Klopfer for our rich discussions this year: they not only helped me understand the internal StarLogo codebase, but also its role in the educational technologies world and students' lives.

Last but not least, I would like to thank my girlfriend Linda Wang for always being there and reassuring me in seemingly hopeless situations that there always will be a way. Thank you for all your inspiration and believing in me.

Contents

1	StarLogo Nova Background	15
1.1	LISP	16
1.2	Logo	16
1.3	StarLogo	17
1.4	StarLogo: The Next Generation	19
1.4.1	StarLogoT and NetLogo	22
1.5	StarLogo Nova	22
1.6	StarLogo Nova Technical Overview	25
1.7	StarLogo Nova Block and Functionality Overview	29
1.7.1	Agent blocks (figure 1-10)	29
1.7.2	Detection blocks (figure 1-11)	31
1.7.3	Environment blocks (figure 1-12)	32
1.7.4	Interface blocks (figure 1-13)	33
1.7.5	Keyboard blocks (figure 1-14)	34
1.7.6	List blocks (figure 1-15)	35
1.7.7	Logic blocks (figure 1-16)	35
1.7.8	Math blocks (figure 1-17)	36
1.7.9	Movement blocks (figure 1-18)	37
1.7.10	Procedure blocks (figure 1-19)	37
1.7.11	Sound blocks (figure 1-20)	38
1.7.12	Trait blocks (figure 1-21)	38
1.7.13	Variable blocks (figure 1-22)	39

1.7.14	Debugger blocks (figure 1-23)	40
2	StarLogo Nova Limitations and Rationale for Contributions	41
2.1	ActionScript3 Migration	41
2.2	New Tooling	43
2.3	Performance Engineering the Engine	44
3	Flash Migration	45
3.1	Initial Migration Steps	45
3.2	Migrating the Engine Core	47
3.2.1	A Difficulty in Migrating the Engine Core	47
3.2.2	A Simpler Interpreter	50
3.3	Miscellaneous Migration Details	55
3.3.1	JavaScript Maps vs Objects	55
3.3.2	Singletons and Static Classes	56
3.3.3	BlockTable	56
3.3.4	Key Manager	56
3.3.5	Dead Code	57
3.3.6	General Statistics	57
4	New Tooling	59
4.1	Migration Tools	59
4.1.1	<code>make</code> and <code>tsc</code>	59
4.1.2	<code>tslint</code>	61
4.1.3	<code>tsfmt</code>	63
4.2	Automatic Testing	65
4.3	Version Control Workflow	65
5	Performance Engineering the Engine	67
5.1	Overview of JavaScript Engines	67
5.1.1	Chrome V8	68
5.1.2	Firefox SpiderMonkey	69

5.1.3	Safari JavaScriptCore	70
5.1.4	Microsoft Edge Chakra	71
5.1.5	Summary of JavaScript Engines	71
5.2	Setting up a Testbed	72
5.3	Interpreter Optimization Attempt	74
5.4	Transpilation to JavaScript	76
5.4.1	Outline of how JavaScript is generated	77
5.4.2	Nuances to generating the JavaScript	77
5.5	Optimizing the Transpiled Code	81
5.5.1	Benchmark Suite Description	81
5.5.2	Universal Optimizations	83
5.5.3	Renderer Optimizations	93
5.5.4	Collision Optimizations	94
5.6	Negative Optimization Results	97
5.7	Overall Benchmark Results	101
6	Future contributions	103
6.1	Better Type Handling	103
6.1.1	Current Type System	103
6.1.2	Stronger Typing Challenges and Solutions	104
6.2	Live code editing	105
6.2.1	Live code editing groundwork	105
6.2.2	Live code editing semantics	106
6.2.3	Live code editing user interface	107
6.3	Debugging	108
6.3.1	Logging	109
6.3.2	Stepping	110
6.3.3	Alternative Stepping Design	110
6.4	User Scripting	111
6.4.1	Security	111

6.5 Tooling	112
7 Conclusion	115

List of Figures

1-1	Graphical representation of the Dragon Curve	16
1-2	Logo code to generate the Dragon Curve	17
1-3	Logo code to generate the Dragon Curve with errors	19
1-4	Blocks code to generate the Dragon Curve	21
1-5	Screenshot of the StarLogo Nova interface	24
1-6	Block Specification of <code>if</code>	26
1-7	Compatible blocks for the <code>if</code> block	27
1-8	Incompatible blocks for the <code>if</code> block	27
1-9	StarLogo Nova Execution Engine Pseudocode	29
1-10	Agent Blocks	30
1-11	Detection Blocks	31
1-12	Environment Blocks	32
1-13	Interface Blocks	33
1-14	Keyboard Blocks	34
1-15	List Blocks	35
1-16	Logic Blocks	36
1-17	Math Blocks	37
1-18	Movement Blocks	38
1-19	Procedure Blocks	39
1-20	Sound Blocks	39
1-21	Trait Blocks	40
1-22	Variable Blocks	40
1-23	Debugger Blocks	40

3-1	AS3 Interpreter code to compute <code>if</code>	48
3-2	AS3 Interpreter code to compute <code>CalcSum</code>	49
3-3	Simpler Interpreter code to compute <code>if</code>	52
3-4	Simpler Interpreter code to compute <code>CalcSum</code>	53
4-1	Makefile <code>all</code> Target	62
4-2	Overriding <code>tslint</code>	63
4-3	<code>tsfmt</code> invocation	64
5-1	Benchmark Script	74
5-2	Rotate Benchmark	74
5-3	JavaScript Code Generation for <code>If</code>	78
5-4	JavaScript Code Generation for <code>ProcCall</code>	79
5-5	Blocks for a Transpilation Example	80
5-6	JavaScript for a Transpilation Example	80
5-7	Profiler with no optimizations	83
5-8	Profiler with 1 round of optimizations	85
5-9	Profiler with 2 rounds of optimizations	86
5-10	Optimized JavaScript for a Transpilation Example	87
5-11	Profiler with 3 rounds of optimizations	87
5-12	Profiler with 4 rounds of optimizations	88
5-13	Profiler with 5 rounds of optimizations	89
5-14	Profiler with 6 rounds of optimizations	91
5-15	Cumulative Impact of General Optimizations	92
5-16	Unoptimized Renderer Profile	93
5-17	Optimized Renderer Profile	94
5-18	Unoptimized Collision Profile	95
5-19	Optimized Collision Profile	96
5-20	Strange Profiler Results	99
5-21	Duff's Device	100
5-22	Overall Benchmark Results Graph	102

List of Tables

5.1	Interpreter Optimization Results	76
5.2	Overall Benchmark Results Table	101

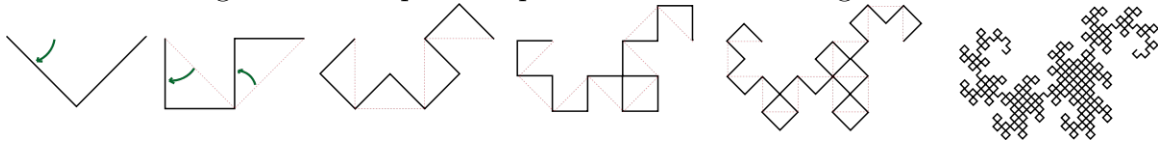
Chapter 1

StarLogo Nova Background

StarLogo Nova allows precollege students to create an enormous variety of games and simulations by dragging and dropping logical blocks together to construct scripts that provide the power of a regular programming language without the steep learning curve that comes with many text-based languages. These scripts govern how agents individually react to stimuli, such as collisions with neighbors or the push of a button. When a simulation is run, the blocks are run by a time-sharing execution engine that simulates the agents locally in the browser. The agents in the simulation are snapshotted between engine cycles, rendered in three dimensions, and shown to the user in real-time.

StarLogo Nova has a rich ancestry as the latest successor in the StarLogo TNG, StarLogo, Logo, and LISP lineage. This chapter presents an overview of these previous environments, the motivations behind the transitions between generations, and an overview of how StarLogo Nova operates. Chapter 2 discusses StarLogo Nova limitations and the motivation for the contributions of this thesis. Chapters 3-5 describe the contributions of this thesis: chapter 3 describes the migration of the StarLogo Nova engine from ActionScript3 to TypeScript; chapter 4 describes new tooling for the StarLogo Nova codebase; chapter 5 describes performance engineering of the Execution Engine. Chapter 6 presents an overview of possible future contributions and chapter 7 concludes this thesis.

Figure 1-1: Graphical representation of the Dragon Curve



The six images shown are the Dragon curve expanded to various iterations.

1.1 LISP

John McCarthy designed LISP in 1958 as a high-level functional language. It is the second oldest programming language¹ and introduced many groundbreaking computing ideas that we take for granted today, such as conditionals, a first-class function data type, recursion, dynamic typing, and garbage collection [4].

1.2 Logo

Papert, Feurzeig, and Solomon created Logo in 1967 as an educational programming language. The Logo language centers around a turtle in a two-dimensional space that can be controlled with primitive instructions such as **forward**, **left**, and **right**. As the turtle moves in the space, it traces the path in which it has traversed, allowing users to visualize the outcome of their program. The language was designed to help students reason about programming by imagining themselves as the turtle and manually tracing through the execution of a Logo script.

While such primitives can draw arbitrary patterns such as the dragon fractal shown in figure 1-1, specifying complicated patterns with just primitives is verbose and difficult to understand. Like LISP, Logo supports recursive procedures, allowing users to compose complicated patterns as calls to simple recursive procedures. In the example shown in figure 1-2, `dcr` and `dcl` are recursive procedures that can be used to draw the dragon fractal. Their compact definitions reveal an insightful representation of the nature of the fractal.

¹Fortran predates LISP by one year

Figure 1-2: Logo code to generate the Dragon Curve

```
to dcr :step :length
  make "step :step - 1
  make "length :length / 1.41421
  if :step > 0 [rt 45 dcr :step :length lt 90 dcl :step :length rt 45]
  if :step = 0 [rt 45 fd :length lt 90 fd :length rt 45]
end

to dcl :step :length
  make "step :step - 1
  make "length :length / 1.41421
  if :step > 0 [lt 45 dcr :step :length rt 90 dcl :step :length lt 45]
  if :step = 0 [lt 45 fd :length rt 90 fd :length lt 45]
end

dcr 9 300
```

`dcr` and `dcl` are recursively defined procedures. The call to `dcr 9 300` draws the 9th iteration of the dragon curve.

1.3 StarLogo

While Logo can help students think about executing a script and decomposing complicated computational steps into smaller procedures, one of its principal limitations is that it promotes a very centralized mindset since there is one turtle² and all the actions are centered on that turtle. While this simple mindset can be useful for tracing a design and thinking about a single algorithm, it can be difficult to use this mindset to reason about how several agents may cooperate or compete in complex environments.

In recognition of this limitation, Mitchel Resnick devised StarLogo as a new computational system in 1989 for modeling the world. Instead of controlling a single turtle tracing patterns, StarLogo users could control an arbitrary number of turtles representing “almost any type of object in the world: an ant in a colony, a car in a traffic jam, an antibody in the immune system, or a molecule in a gas.”[7] To reduce

²Some Logo variants can actually support a few turtles, but it is difficult to control them in a scalable manner.

confusion, I will henceforth use the term agent to describe these simulated turtles.

StarLogo diverges from Logo in three main areas: (1) StarLogo supports up to thousands of agents, (2) StarLogo agents can sense other nearby agents, and (3) the environment can spawn agents and store scents in addition to storing the drawn pattern.

With the ability to simulate many turtles at scale, StarLogo users can investigate how macro-level behaviors emerge from micro-level interactions. Emergent behaviors from a collective are often determined by the size of the collective and in order to accurately model many real-life phenomenon, StarLogo must simultaneously simulate thousands of agents³.

By equipping agents with the ability to sense nearby agents and read markings in their environment, agents can start interacting together. This allows for users to create interactive simulations, such as Conway's Game of Life or even models of animal colonies. By simulating interactions between agents, StarLogo allows students to investigate hypotheses for various phenomena and systematically reason like a researcher.

In contrast to Logo, StarLogo stores more than just pen traces in the environment. It also divides the scene into a large number of patches that can be individually marked with scents and chemicals. Additionally, each patch can also run scripts independently of agents.

By introducing the ability to simulate large quantities of agents, StarLogo allows users to model complex real-world behavior and understand how the actions of a few agents can impact the overall behavior of a group. For example, early StarLogo users modeled how traffic lights impact traffic throughput and found that strategically placed stops actually increased the traffic throughput, relative to a control group with no stops. Their conclusions matched both complicated queuing theory equations and empirical findings from the New York City Port Authority[7].

³StarLogo was initially implemented on Connection Machine, a passively parallel supercomputer with thousands of cores, with each core simulating a single agent in parallel. Shortly thereafter, Resnick reimplemented StarLogo on traditional sequential machines by time sharing the processor to simulate parallelism, allowing users without supercomputer access to use StarLogo.

Figure 1-3: Logo code to generate the Dragon Curve with errors

```
01 | to dcr :step :length
02 |     make "step :step - 1
03 |     make "length:length / 1.41421
04 |     if :step > 0 [rt 45 dcr :step :len lt 90 dcl :step :len rt 45]
05 |     if :step == 0 [rt 45 fd len lt 90 fd :len rt 45]
06 | end
07 |
08 | to dcl :step :length
09 |     make ''step :step - 1
10 |     make ''length :length / 1.41421
11 |     if :step > 0 [lt 45 dcr ;step :length rt 90 dcll :step :length lt 45]
12 |     if :step == 0 [lt 45 fd :length rt 90 fd :length lt 45
13 | end
14 |
15 | dcr 9 300
```

There are 13 minor errors that a novice may make while trying to implement the dragon curve. These errors are: 1) line 3: there is a missing space between before the colon in `length:length` 2-5) line 4-5: `len` should be `length` 6) line 5: `==` should be `=` 7) line 5: missing colon before the first length 8-9) line 9-10: double quotation marks should be used instead of two single quotation marks 10) line 11: the semicolon should be a colon 11) line 11: `dcll` should be `dcl` 12) line 12: `==` should be `=` 13) line 12: missing end bracket.

1.4 StarLogo: The Next Generation

StarLogo appears to promise the user unlimited power and complete understanding of the universe through simulations. However, StarLogo simulations are designed through careful text-based scripting of how agents interact together. As such, the simulation environment requires an exact understanding of programming syntax and minor mistakes can prevent the program from running at all. Figure 1-3 shows the source from Figure 1-2 with 13 minor errors that prevent the interpreter from running. These errors can be especially difficult to identify and fix for inexperienced users who have never seen code before.

Although motivated students picked up the StarLogo syntax and gained from the simulations, many students were intimidated by the dense juxtaposition of characters. When they tried to edit the code, they would sometimes be greeted by a sequence of

confusing error messages.

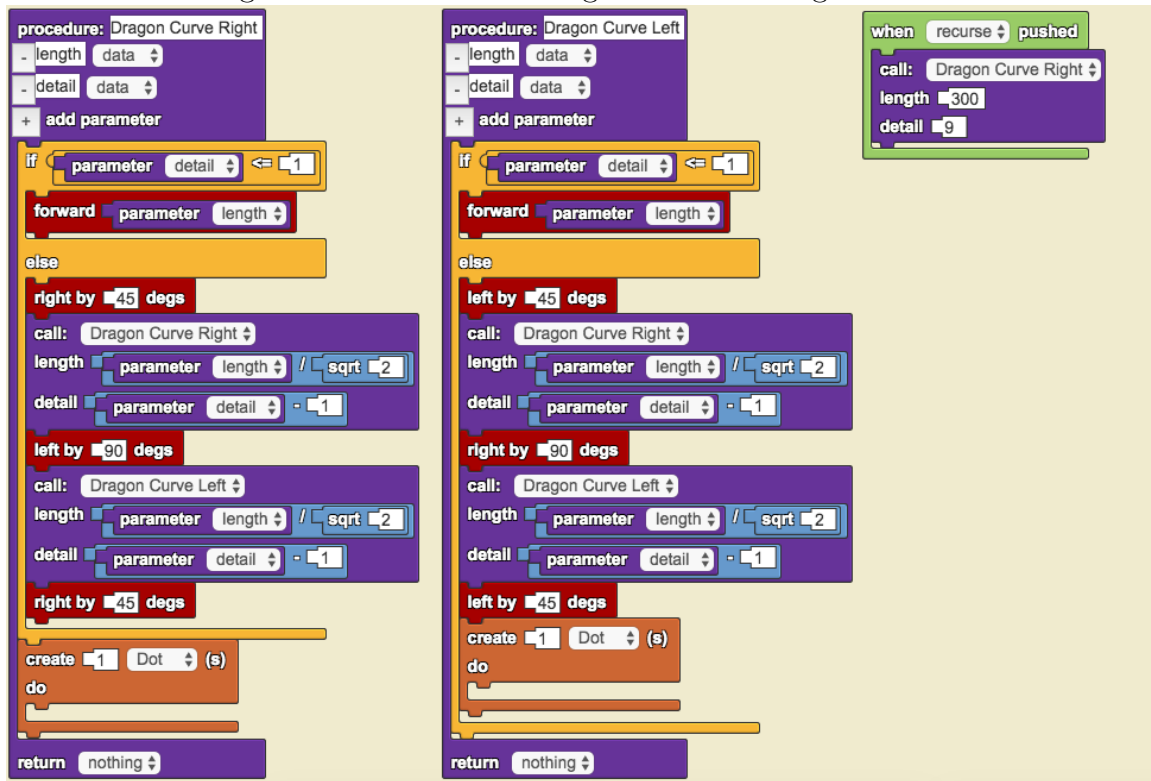
In an effort to bring the power of StarLogo to the masses, Eric Klopfer et al created StarLogo: The Next Generation (TNG) to enable secondary school students and teachers to construct their own simulations through visual blocks[8]. These blocks impose limitations on the structure of commands, ensuring that only valid programs can be assembled. In contrast to the text code for drawing a dragon curve in figure 1-2, the blocks code for drawing the same curve in figure 1-4 provide many hints about how they fit together and how they may be used. The StarLogo TNG interface also includes a drawer of blocks so users can quickly review what blocks are at their disposal instead of having to memorize all the keywords in the programming language.

This shift to blocks allows users with minimal programming experience to focus on the logic and parameters of their scripts instead of syntax. Consequently, many students and teachers expressed much greater comfort using the blocks editor in StarLogo: TNG relative to the textual code editor in previous versions of StarLogo.[8]

Another big advance that StarLogo: TNG brings is the addition of a three-dimensional renderer. Made possible by huge advances in computational power and graphics technology, this rich environment motivates users to build realistic models of systems in a familiar videogame-like setting. Additionally, this renderer makes it possible for users to create their own interactive 3D video games, bringing fun into the modeling process.

The StarLogo TNG team found the notion of patches running scripts to be confusing to students since many of them had a difficult time understanding that some blocks could only be run by patches and vice versa. Patches also contribute significantly to the simulation time: a rough division of the scene into a 100 by 100 grid results in the simulation of 10,000 patches each iteration of the execution engine. Consequently, the StarLogo TNG team deliberately left patches out to reduce student confusion and increase runtime performance. If a user really wants patch-like behavior, they could emulate patches by manually constructing and scattering invisible immobile agents.

Figure 1-4: Blocks code to generate the Dragon Curve



This is a program to generate the Dragon Curve in a blocks-based programming language. This script uses slightly different logic than the code in figure 1-2.

1.4.1 StarLogoT and NetLogo

Uri Wilensky created StarLogoT and NetLogo to extend StarLogo by introducing more built-in functions and datatypes to model more complicated behaviors[9]. NetLogo also comes with a rich library of existing models and is extensible, allowing users to interface with external components such as the filesystem and network. NetLogo is designed for high performance and many user scripts are compiled down to Java bytecode and executed by the Java Virtual Machine (which may in turn compile user scripts down to native code for the CPU to execute directly).

NetLogo is currently in the process of being ported to the web, and one of their challenges is achieving high performance⁴.

Unlike StarLogo TNG, StarLogoT and NetLogo retain the scripting interface and are not blocks based. Although NetLogo is a more powerful platform in terms of raw language features, it also has a higher learning curve than StarLogo TNG due to its scripting interface.

1.5 StarLogo Nova

One shortcoming of StarLogo: TNG is that it was built as a desktop Java application. While it can run on the widely ported Java Virtual Machine, it still encounters some limitations: (1) students may not be able to access their school StarLogo: TNG projects from home if they save their files on their school computer (2) school computers are heavily locked down and it is difficult to install native applications (3) not all operating systems versions are supported because it is expensive to ensure compatibility across a myriad of system interfaces and hardware drivers.

StarLogo Nova is a descendant of StarLogo TNG that runs on the web, addressing many of these limitations. (1) By storing projects remotely in a database, users can create, edit, and run their simulations anywhere from any device. (2) By nature of being a web application, users will not have to install StarLogo Nova on their

⁴The NetLogo team describes their challenges with high performance at <http://netlogoweb.org/info#performance>. As of mid 2016, they are focusing more on feature parity than performance.

computer directly, allowing teachers to use StarLogo Nova in their classrooms without having to configure any devices. This significantly increases the number of students StarLogo Nova can reach. (3) Furthermore, modern browsers offer a consistent API across a diverse set of platforms⁵, making it relatively simple to target a wide range of hardware, including tablets. As an added bonus, with many people and organizations looking towards the web as the future of applications, it is significantly easier to find and recruit StarLogo Nova contributors with web experience.

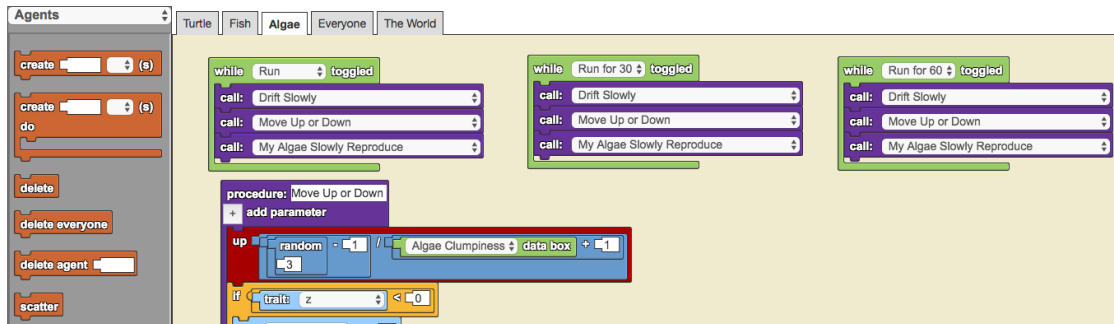
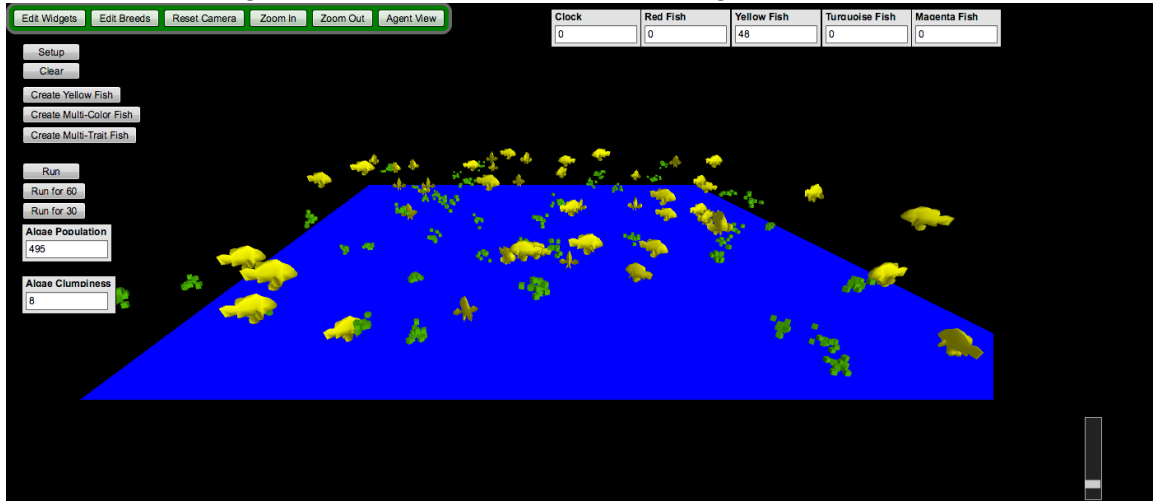
One of StarLogo Nova’s primary goals is to dramatically improve upon the usability of StarLogo TNG and further reduce the learning curve, based on 10 years of StarLogo TNG user studies. StarLogo Nova allows the user to customize the interface through a flexible widgets layer and carefully reduces the number of blocks by over 50% to simplify the blocks language. There are also many user interface innovations upon StarLogo TNG, such the introduction of an omnipresent static block drawer (figure 1-5).

To summarize the cumulative progress that Logo descendants have made in pursuit of a better educational tool, StarLogo Nova is a web-based game and simulation creation environment that combines a blocks-based programming language with a pseudo-parallel agent-based simulation engine and a 3D rendering engine. Users can drag and drop blocks to construct scripts that control agents and how they interact with each other to run biological simulations, first person games, and many other models of the world. Figure 1-5 shows a screenshot of the StarLogo Nova interface.

The contributions in this thesis revolve around the inner workings of the StarLogo Nova platform. The next section gives a technical overview of how the platform operated, prior to the contributions of this thesis.

⁵Although browsers can somewhat inconsistent between each other, versions, and operating systems, their differences are vanishingly small compared to native desktop and native mobile environments.

Figure 1-5: Screenshot of the StarLogo Nova interface



The top half of the screen is used by the renderer to display the simulation and the widgets interface to allow the user to interact with the simulation. The bottom half of the screen is the block editing interface. The gray left box is the block drawer that users can scroll through to design the behavior of their agents. The tan right box are where users combine blocks together to form scripts. The editor is currently on the "Algae" tab and every algae individually executes each script on this tab. Scripts on other tabs control other sets of agents.

1.6 StarLogo Nova Technical Overview

All block editing in StarLogo Nova is handled by ScriptBlocks, a JavaScript library written with Closure⁶. Blocks in StarLogo Nova are defined by ScriptBlocks specifications, which govern how blocks can connect to each other. For instance, an `if` block has two sockets: a boolean data socket and an instruction list socket. It also accepts command instructions to connect before and after it.

ScriptBlocks provides some degree of type safety: only a boolean block can be used to fill the first socket of the `if` block and only command blocks can be used to fill the second socket. These rules prevent a user from constructing many sorts of invalid programs. Figure 1-6 shows the specification for the `if` block and figures 1-7 and 1-8 show examples of how the `if` block can be used with other blocks.

There are 5 data types that ScriptBlocks differentiates between: dropdowns, booleans, lists, commands, and a generic data type.

Dropdowns restrict the block input to the output of a generic callback, which allows ScriptBlocks to enforce valid selections for arguments such as widget names, procedures, and traits.

Booleans (half circles) represent `true` and `false` values and are used extensively in logical comparisons.

Lists (triangles) store lists of either booleans, lists, or generic data types.

Commands (flat) are instruction blocks that do not return a value.

Generic data types (squares) are used as a catch-all to represent numbers, strings, colors, sounds, shapes, agents, and nulls.

All type safety is enforced by ScriptBlocks and the engine generally does not know about type information. When a block requires arguments to be of a certain type, the engine will cast arguments appropriately.

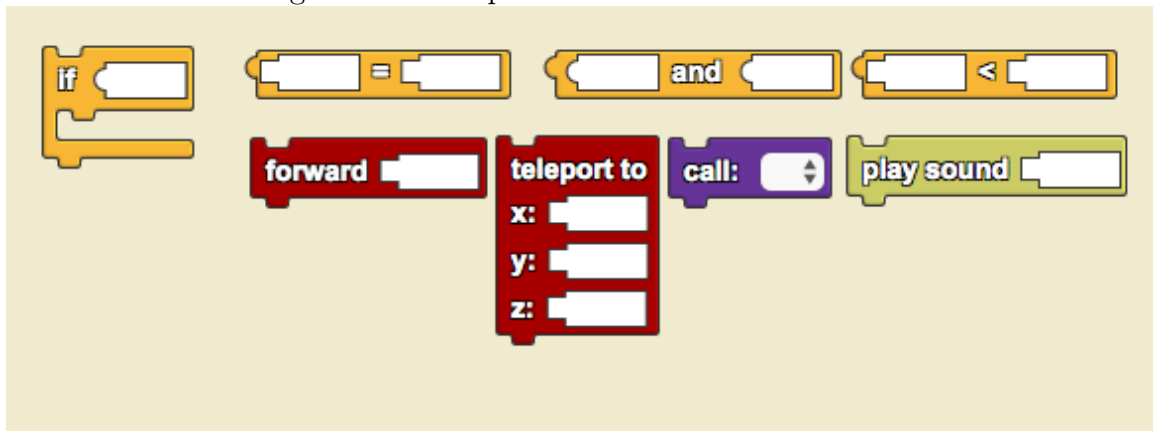
⁶Closure is a platform created by Google to help developers write optimized JavaScript. It features an extensive modular library, an optimizing compiler, and code health tools.

Figure 1-6: Block Specification of if

```
{
  "name": "if",
  "color": "#cde8d5",
  "label": "if @test \n @then",
  "connections": [
    "after",
    "before"
  ],
  "arguments": [
    {
      "dataType": "boolean",
      "socketType": "internal",
      "name": "test",
      "options": []
    },
    {
      "dataType": "command",
      "socketType": "nested",
      "name": "then",
      "options": []
    }
  ],
  "returnType": "command"
}
```

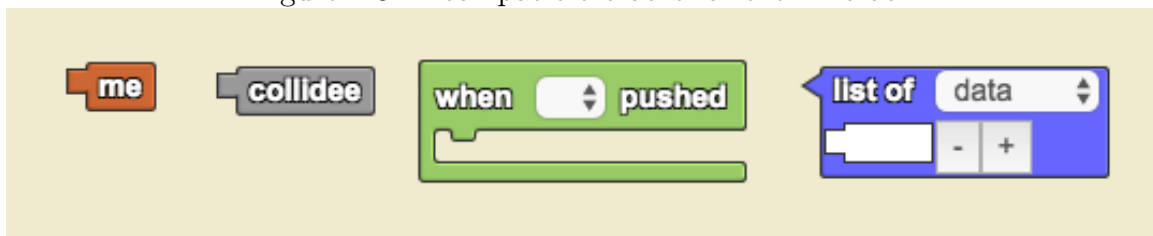
The block specification defines the name, color, label, collections, and arguments of a block. Arguments are typed with the `dataType` field.

Figure 1-7: Compatible blocks for the if block



The =, and, and < blocks are boolean blocks. The forward, teleport, call, and play sound blocks are command blocks. Note that blocks that can fit in the first socket (top) cannot fit in the second socket (bottom) and vice versa.

Figure 1-8: Incompatible blocks for the if block



None of these blocks can fit in either socket for the if block (shown in figure 1-7).

For a given state of the editor, ScriptBlocks internally represents the user's code as a forest of trees, with each tree corresponding to a stack of visual blocks. Socketed blocks are represented as the children of the parent block. When a user performs an action (such as creating, connecting, or disconnecting a block), ScriptBlocks updates its internal state, updates the DOM with this new state, and sends signals to the execution engine corresponding to the change event.

The execution engine is responsible for accurately and efficiently simulating the agents in the environment. For each agent and each script, the execution engine constructs a thread object and runs all the threads "in parallel". To provide the illusion that many agents are running in parallel when there is a single execution thread, threads are divided into small slices with `yield` blocks. Whenever the execution hits a `yield` block, the engine pauses execution of the current thread and begins stepping the next thread. `yield` blocks are invisibly attached to the end of `if-button-pressed` blocks and can also be explicitly added by the user. To further the illusion that agents are running in parallel as opposed to taking turns, all direct interactions between agents act on the previous states of the agents: upon collision, all agents create a copy of themselves and collision code runs on these previous states.

When all regular threads have been run once, the engine computes collisions and then begins running collision threads (if any exist) for agents that have collided. This concludes one cycle of the engine. Figure 1-9 shows pseudocode for one cycle of the engine.

Motivated by computational performance, the engine operates in a manner similar to a CPU that jumps after each instruction. It maintains a linked list of instruction nodes for each script. Because execution can be nondeterministic (there is support for random number generators), this list is constructed at runtime: whenever the engine executes an instruction node, it will pop arguments from a per-thread data stack and push return values onto this stack. Then the instruction will return a reference to the next instruction node to execute⁷.

⁷There is a compilation stage that sets up all instructions, giving each instruction references to the set of all instructions that could possibly be executed next.

Figure 1-9: StarLogo Nova Execution Engine Pseudocode

```
for each agent:
  for each thread in agent.threads:
    done = thread.step()  // run each thread until it yields
    if done:
      agent.threads.remove(thread)

for each (agent1, agent2) that have collided:
  for each script for (agent1.breed, agent2.breed) collisions:
    // construct a thread based on the script and colliding agents
    thread = new Thread(script, agent1, agent2)
    done = thread.step()
    // if this collision thread has not finished,
    // schedule it for running again the next cycle
    if not done:
      agent1.threads.add(thread)
```

This pseudocode represents one cycle of the Execution Engine and the Engine is expected to run many times per second.

The renderer retrieves a snapshot of all the agents' state and updates the DOM. Because the execution engine often takes a long time to run a cycle, the renderer will draw up to 12 frames per engine tick, interpolating between the previous snapshot of the agents' state and current snapshot to present the illusion of a smooth real-time simulation.

1.7 StarLogo Nova Block and Functionality Overview

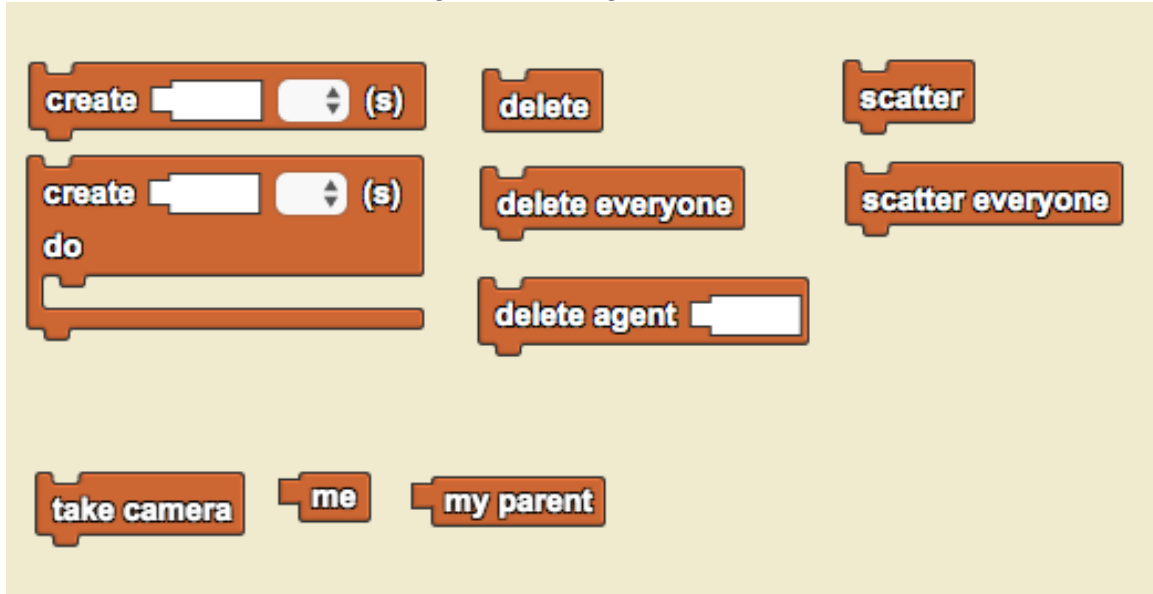
This section gives an overview of all the blocks supported in StarLogo Nova and their intended functionality. All the blocks are shown in figures 1-10 through 1-23.

1.7.1 Agent blocks (figure 1-10)

Agent blocks are used to create, delete, and scatter agents. Additionally, there are blocks for possessing the camera and passing the agent and its parent around as

arguments.

Figure 1-10: Agent Blocks



There are two blocks for creating agents: **agent-create** and **agent-create-do**. **agent-create** accepts a number and breed and creates that many agents of the given breed, inheriting the position of the parent agent that created these new agents. Although the agents all share the same *x*, *y*, and *z* position, the new agents' headings are uniformly distributed about a circle so that if they all move forwards, they will separate. In addition to the number agents and breed arguments that **agent-create** accepts, **agent-create-do** accepts a list of command blocks and will run that list of commands immediately for each agent it spawns.

At the start of each simulation, there is a single invisible agent spawned in the center called "The World". This agent is not allowed to move or die, but can function as an agent in all other situations, such as creating other agents.

There are three delete blocks: **delete**, **delete-everyone**, and **delete-agent**. The first two don't accept arguments and delete the calling agent and all agents (except the world), respectively. **delete-agent** takes in an agent as an argument and deletes that agent. For example, **delete-agent** could be used in collisions to

delete the `collidee`⁸ or more generally, delete an agent stored in a trait⁹.

There are two scatter blocks: `scatter` and `scatter-everyone`. `scatter` will move the current agent to `x` and `y` coordinates chosen uniformly at random. `scatter-everyone` applies `scatter` to every agent (except the world).

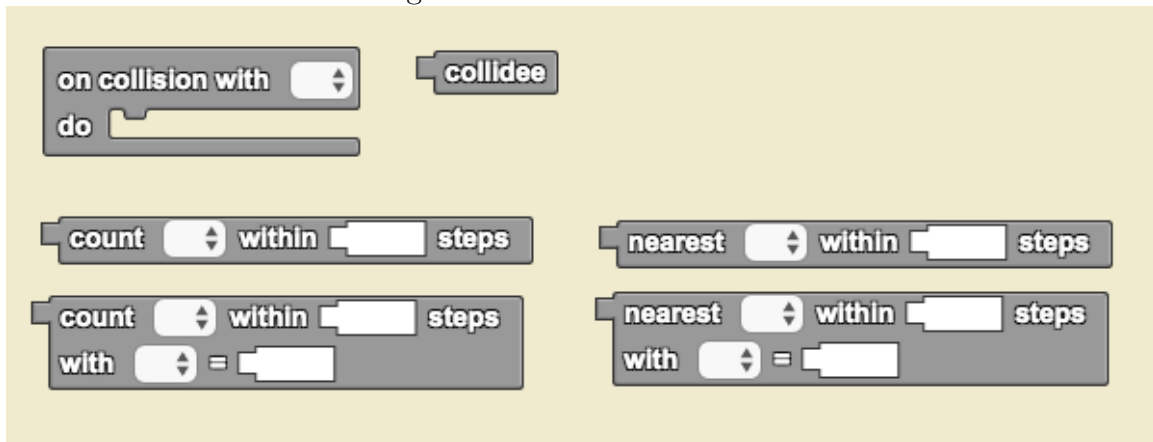
`take-camera` is a special instruction to the renderer to draw the scene from the current agent's point of view. If the agent that has taken the camera dies, then the camera is restored to an aerial view.

`me` returns a reference of the current agent. `my-parent` returns a reference of the current agent's parent (the agent that created the current agent, which may not necessarily be of the same breed).

1.7.2 Detection blocks (figure 1-11)

Detection blocks enable agents to sense their environment and react to collisions with other agents.

Figure 1-11: Detection Blocks



`on-collision` is a top level block that allows users to control how agents behave upon collision with other agents. It accepts an argument determining the breed of the other agent, which allows for precise collision targeting. `collidee` is a special block that returns the agent that the current agent has collided with.

⁸1.7.2 describes collisions in greater detail

⁹Section 1.7.12 describes traits in greater detail.

`count` and `count-with` return the number of agents of a specified breed within the given radius (steps). `count-with` allows users to impose a constraint on the traits of the agents that are included in the count.

`nearest` and `nearest-with` return the nearest agent of the specified breed within a radius. Like `count-with`, `nearest-with` allows users to impose a constraint on the trait of the agent to be returned.

1.7.3 Environment blocks (figure 1-12)

Environment blocks control how agents can act with the scene, the world (a persistent agent described in section 1.7.1), and time.

Figure 1-12: Environment Blocks



`stamp` and `stamp-grid` allows agents to stamp an imprint of themselves onto the ground with the given color: `stamp` draws a circle proportional to the size of the agent and `stamp-grid` draws a unit circle centered at the agent. The `pen` block toggles the state of an agent's pen. If it is on, it traces a path of where it has traveled on the ground whenever it moves.

`world-trait` returns the given trait of the world agent. `set-world-trait` sets the given trait of the world agent to the given value. `the-world` returns a reference to the world.

At the end of each cycle, the engine increments a numerical clock. The `clock` instruction returns the current value of the clock and `set-clock` can set the clock

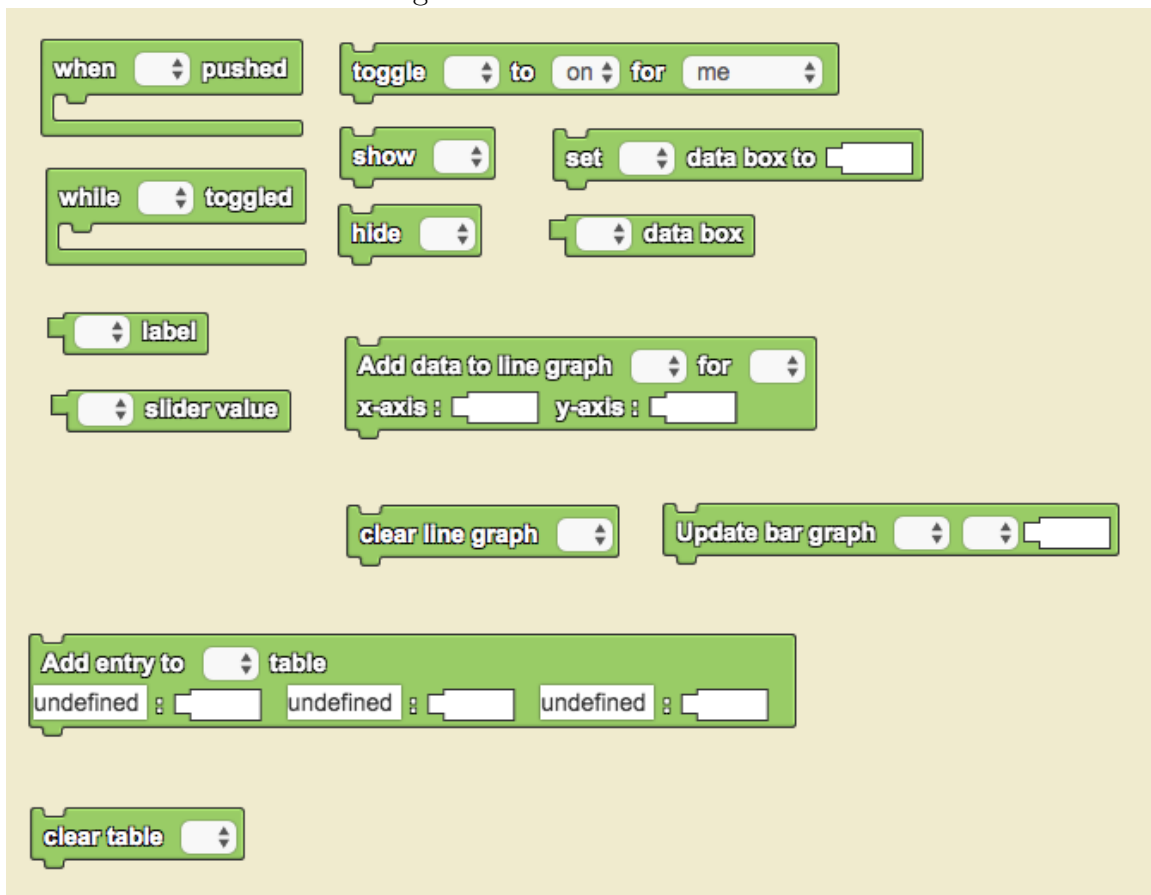
value.

`terrain-color` returns the color underneath the current agent. `clear-terrain` clears the terrain of any stamps, unit squares, and traced paths by pens.

1.7.4 Interface blocks (figure 1-13)

During execution, users can interface with agents through graphical widgets such as buttons, data input boxes, and sliders. Agents can also interface with the user through labels, data boxes, tables, and graphs.

Figure 1-13: Interface Blocks



`when-pushed` and `while-toggled` are two common top level blocks that bind user actions to push and toggle buttons. For `when-pushed`, all blocks under `when-pushed` are executed once whenever the specified button is pushed. For `while-toggled`, while the given button is toggled to the on state, all blocks under the `while-toggled`

execution each cycle. The `toggle-button-set` instruction allows users to override the default `while-toggled` behavior on a per-agent basis, enabling certain agents to stop running or always run.

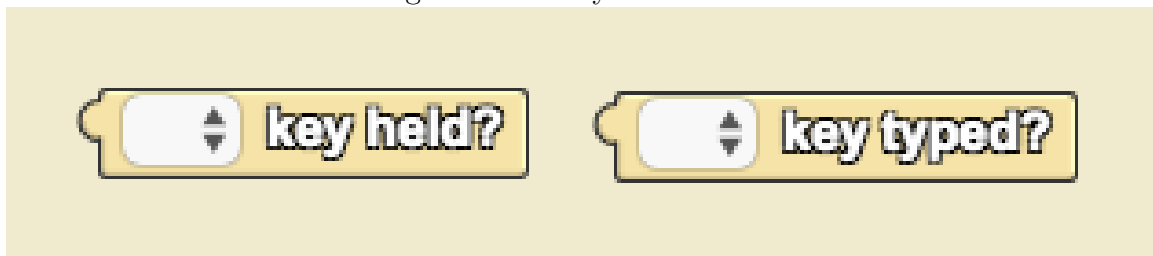
The `show` and `hide` blocks control the visibility of widgets. The `get-data-box` and `set-data-box` blocks read from and write to data input boxes, respectively. The `label` and `slider-value` blocks return the current state of the label and slider, respectively.

There are additional blocks for updating charts and tables. These blocks are primarily designed for the world agent to display the number of agents matching certain criteria in a persistent widget to help users record simulation outcomes.

1.7.5 Keyboard blocks (figure 1-14)

Keyboard blocks allow users interact with agents with their keyboard. When a key is pressed, it is added to a set of currently pressed keys and also to a queue of keys that have been pressed. When a key is released, it is removed from the set of currently pressed keys. At the end of each engine cycle, it will pop the head of the queue.

Figure 1-14: Keyboard Blocks

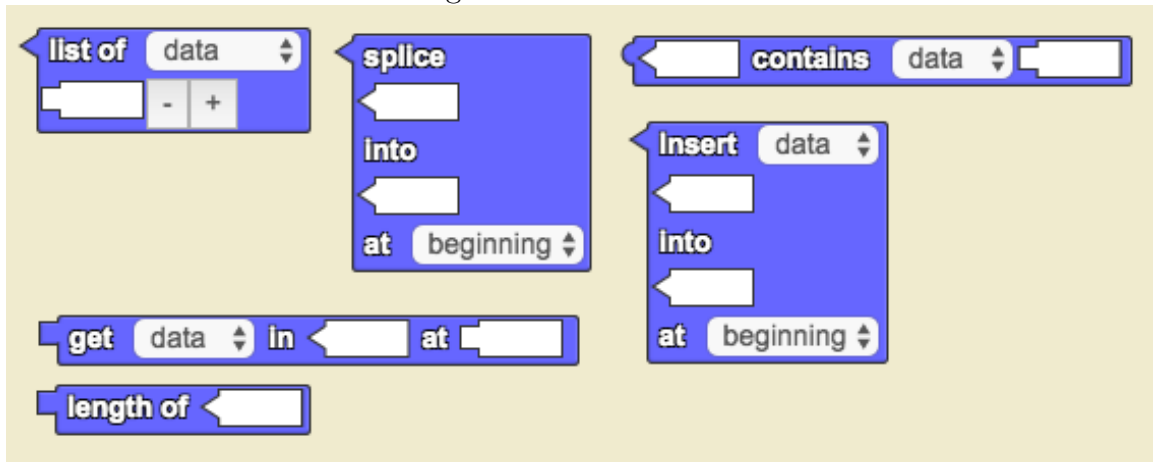


The `key-held` block returns whether specified key is in the set of currently pressed keys. The `key-typed` block returns whether the specified key is the head of the queue. This design feeds the keystrokes to the `key-typed` blocks at a rate of one key per engine cycle, preserving the order of the keystrokes and preventing keystrokes from being dropped.

1.7.6 List blocks (figure 1-15)

List blocks allow users to work with lists of generic data. Lists are currently not fully integrated with other blocks and can only be used in a limited number of places.

Figure 1-15: List Blocks



`list-of` is a constructor to create a list of data. The `-` and `+` buttons allow the users to dynamically change the number of arguments this block accepts. `splice` and `insert` allow users to combine lists together.

`contains`, `get`, and `length` behave as one would expect: they check if the list contains an object, get the n^{th} item in the list, and return the length of the list, respectively.

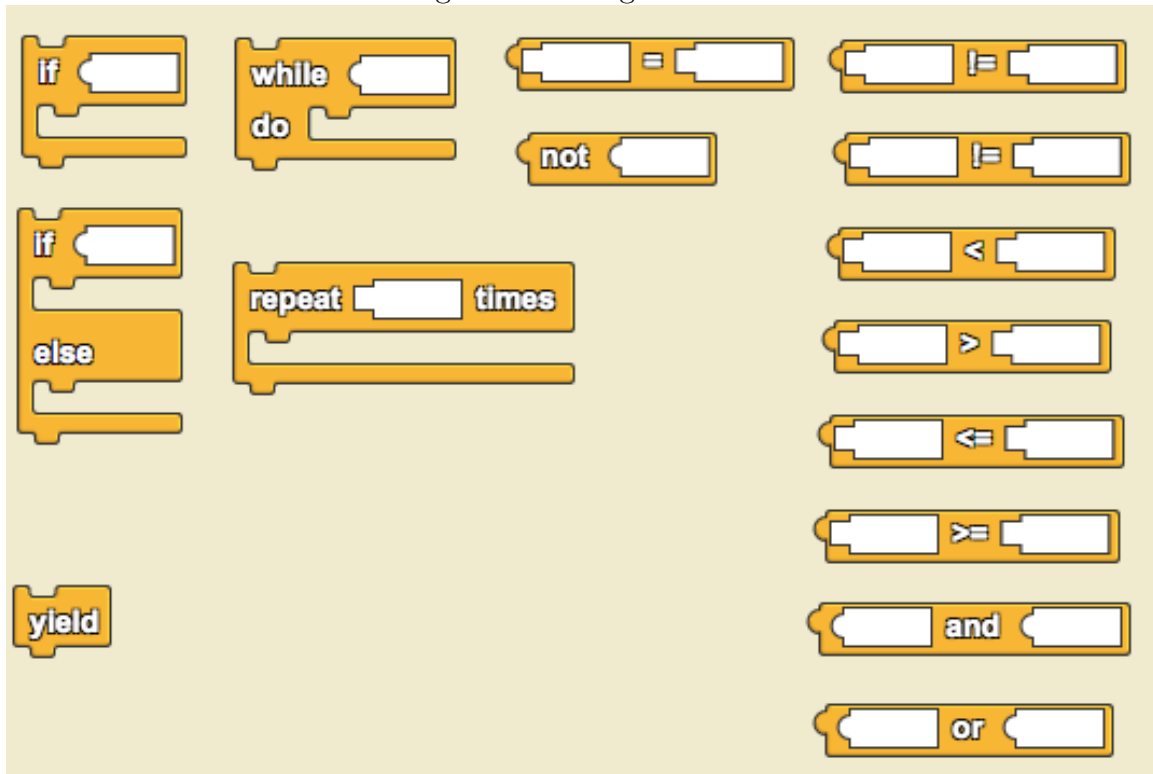
1.7.7 Logic blocks (figure 1-16)

Logic blocks allow agents to branch, loop, check logical operators, and yield.

The majority of these blocks operate intuitively and do not require explanation. Similar to many programming languages, the `and` and `or` blocks short circuit evaluation if the first argument is `false` or `true`, respectively.

The `yield` block signals to the engine to pause execution of the script for the current cycle and resume execution right where it left off the next cycle. This allows for the renderer to step in and draw temporary state during a lengthy computation and also for complicated multithreaded behavior: an agent could jump upon keyboard

Figure 1-16: Logic Blocks



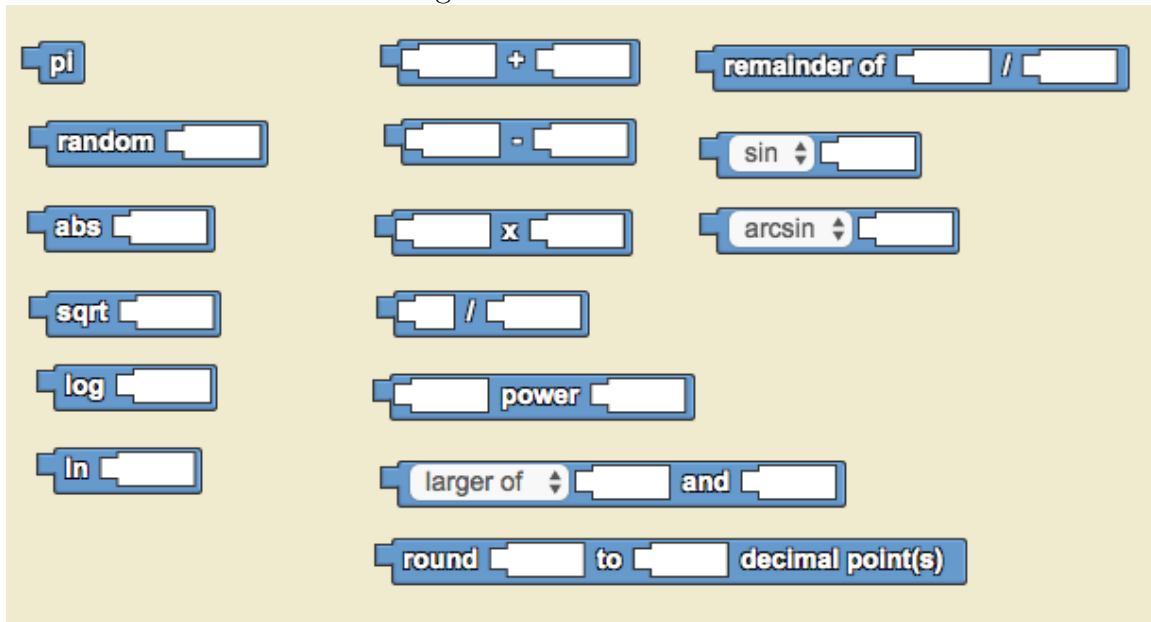
input and yield between a list of `move-up` and `move-down` blocks. Although `yield`-like behavior could be accomplished by creating state machines, yielding is a much simpler approach to reasoning about multithreaded behavior.

For an interpreter-based engine with per-thread data stacks, one can implement `yield` by saving the state of the stack and the cursor position in the stack of blocks and breaking out of the interpreter loop. However, implementing `yield` in other kinds of engines is less straightforward.

1.7.8 Math blocks (figure 1-17)

Math blocks allow users to access common constants, perform calculations, and generate random numbers. All of these blocks function intuitively.

Figure 1-17: Math Blocks



1.7.9 Movement blocks (figure 1-18)

Movement blocks allow agents to move and rotate. Most of these blocks are straightforward with the exception of `face-towards`. `face-towards` takes in an agent and rotates the current agent to face the other agent. Because agents, numbers, and strings are lumped together in a generic datatype, ScriptBlocks unfortunately does not provide much type safety for this block.

1.7.10 Procedure blocks (figure 1-19)

Procedure blocks allow users to create and for agents to call procedures.

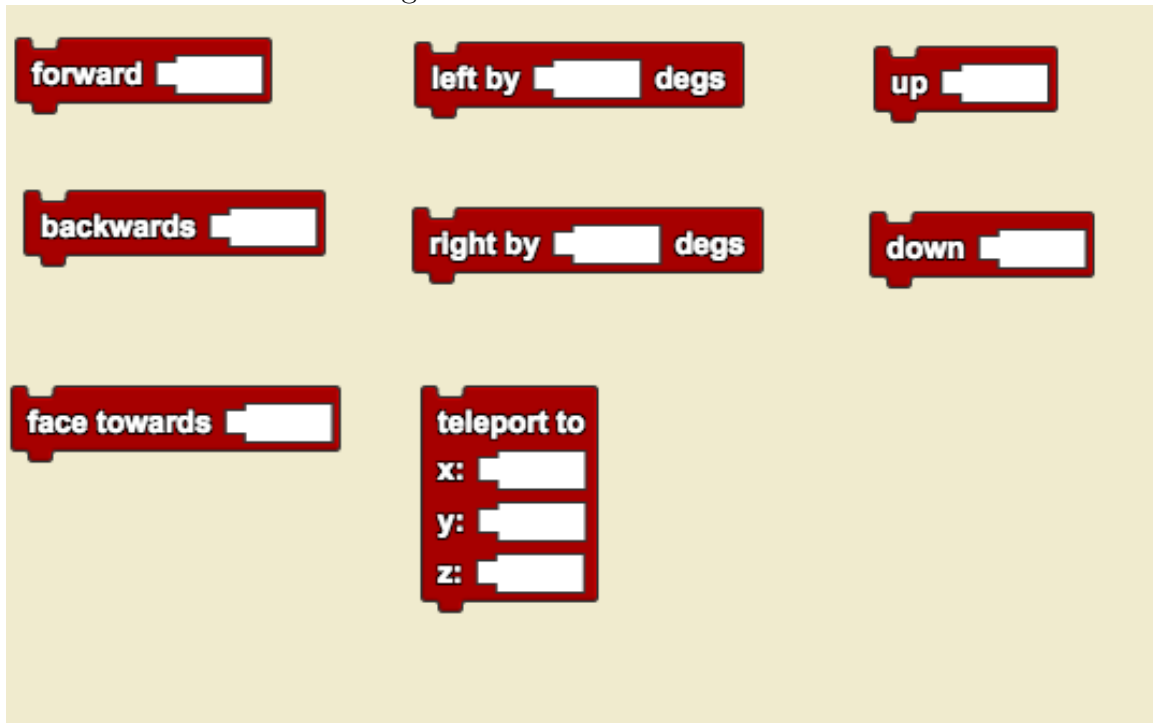
The top level procedure block allows users to define a procedure and adjust the number of parameters, their names, and their data types. At the bottom of the procedure block is a dropdown that allows the procedure to return a value.

The two types of `call` blocks accept a procedure name and upon receiving a name, expand to provide arguments for the procedure.

`parameter` blocks are used within a procedure to read the parameters given to the procedure.

The `return-early` block allows the user to halt execution of the procedure and

Figure 1-18: Movement Blocks



optionally return a value.

1.7.11 Sound blocks (figure 1-20)

Sound blocks are used to play sounds during execution and manage sounds while users are building their programs.

The **sound-play** block accepts a sound-options block. The **sound-options** block contains an argument to allow users to select a sound. The buttons allow users to play or delete the current sound and to record a new sound using their system microphone.

1.7.12 Trait blocks (figure 1-21)

Traits are instance variables on every agent. A breed has a set of traits and each agent of that breed has an instance of each trait. All breeds have a set of universal traits: **id**, **breedname**, **x**, **y**, **z**, **heading**, **color**, **shape**, and **size**. Users can also add additional custom traits. When agents are constructed, they inherit shallow copies of traits from their parent.

Figure 1-19: Procedure Blocks

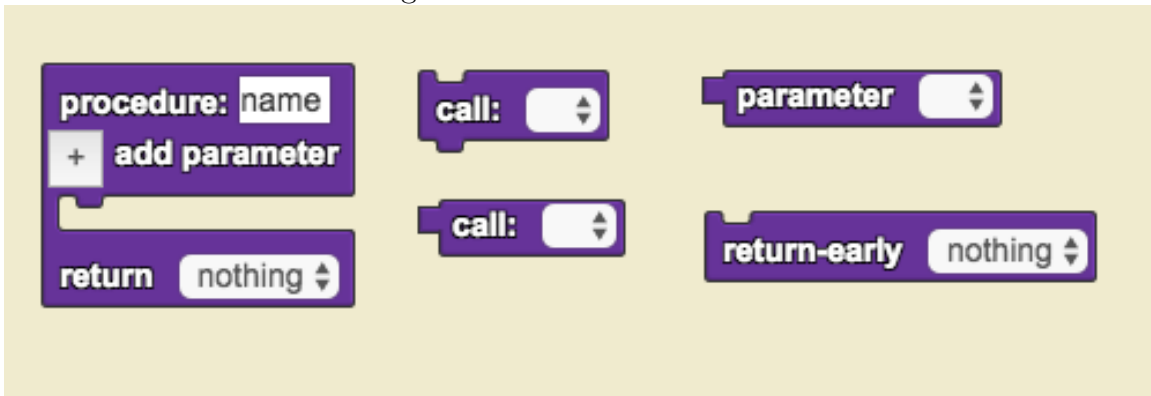


Figure 1-20: Sound Blocks



The `my` and `set-my` blocks get and set the current agent's traits, respectively. Some traits, such as `id` and `breedname`, are read-only so those are not presented in the dropdown and cannot be set.

There is also a `trait-of` block that allows users to get a trait of another agent. There are also environment blocks that allow users to get and set traits of the world agent (See section 1.7.3).

Internally, colors are represented as numbers from 0 (black) to 0xFFFFFFFF (white) and shapes are represented as URLs to shape files. Since these values are not very human friendly, there are argument blocks (`color` and `built-in shape`) that with human-readable dropdowns that allow the user to pick colors and shapes. Script-Blocks dropdowns offer a layer of indirection that maps human readable names to encoded values.

1.7.13 Variable blocks (figure 1-22)

Variables are named values that exist within a block stack.

The three variable blocks allow users to declare, set, and get variables and they

Figure 1-21: Trait Blocks

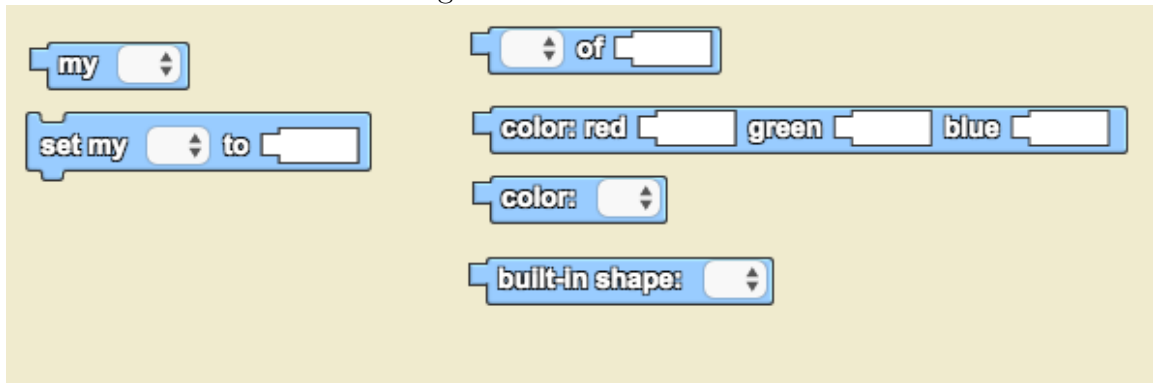


Figure 1-22: Variable Blocks



function as one would expect.

If a variable is accessed within an `agent-create-do` instruction, then reads and writes to that variable are within the scope of the outermost variable block.

1.7.14 Debugger blocks (figure 1-23)

Debugger blocks allow users to inspect the state of their scripts through logging.

Figure 1-23: Debugger Blocks



The `print-to-JS` blocks print the given value to the browser's JavaScript console.

Chapter 2

StarLogo Nova Limitations and Rationale for Contributions

Despite being a useful tool for students and educators, StarLogo Nova has many internal challenges. StarLogo Nova was originally written in ActionScript3 (AS3), which is slated for deprecation by multiple browsers and we must move on to more modern and secure platforms.

Additionally, the StarLogo Nova codebase has few build processes in place internally, making it difficult to deploy new features. For example, the lack of test cases makes it time consuming and error prone to verify that a change does not introduce regression bugs.

The StarLogo Nova engine can also be slow, which limits the number of agents and the complexity of their behavior in large simulations.

The following sections describe these limitations in greater detail and present outlines of solutions to these challenges. The following chapters describe these solutions in significantly greater depth.

2.1 ActionScript3 Migration

StarLogo Nova requires fast and local computation in order to simulate up to thousands of agents in real time. The local computation requirement effectively restricts

the choice of programming platform to either native JavaScript or browser plugins as Adobe Flash (which executes AS3). At the time of initial development, JavaScript performance significantly lagged AS3 and browsers poorly supported native graphics (only 5% of users could run WebGL on their browsers). In light of these conditions, the decision was made to build the core engine that simulated agents in AS3.

However, the Flash runtime is notorious for security vulnerabilities. There are frequent reports of zeroday vulnerabilities, which are security vulnerabilities known to researchers (potentially malevolent black hats) that can exploit fully updated Flash runtimes with minimal user interaction and take control of the host operating system. StarLogo Nova has a high potential for attack because of reliances on external content delivery networks and a rotating student development team that may not be necessarily well versed in security best practices. The nail in the coffin for Flash is that major browser vendors, such as Google and Mozilla, have announced in 2016 that they will be blocking flash by the end of the year due to the recent number of high profile security incidents[1, 5, 3]. Thus, it is vital to pursue redeveloping the core engine in a more secure platform.

JavaScript is presently a vibrant and healthy web development platform with open standards, multiple competing and performant interpreters (including ones that run on mobile devices), a large community, and exciting proposals for the future. Additionally, the majority of browsers support native graphics through WebGL. To advance StarLogo Nova to modern web standards, I migrate the core execution engine to JavaScript. This large effort will protect the safety of users, support modern standards, and make the installation even easier for new users.

One major downside of JavaScript is that it is an untyped language that will silently fail when incompatible types are combined¹. TypeScript is an open source typed weakly language created by Microsoft that compiles to JavaScript, performing type checks during compilation. This compile-time type checking helps prevent bugs in a large project while not slowing down execution. Furthermore, since both AS3

¹On Chrome v52, `[] + []` returns "", `+ []` returns 0, `"A" + 1` returns "A1". For all of these examples, JavaScript silently type converts and does not throw an exception.

and TypeScript are based on the ECMAScript standard, their syntaxes look visually similar, allowing simple string substitution to do a substantial portion of the conversion.

2.2 New Tooling

The StarLogo Nova codebase has been through the hands of many MIT undergraduates studying computer science. Although it is packed with ingenuity and clever snippets, it is also brittle, needlessly verbose, difficult (and scary) to change in many places, and contains subtle bugs.

One way to address these concerns is to add automated test cases to verify correctness of the code. With automated tests, it is significantly easier to refactor code to make a feature as simple and coherent as possible, without having to go through a laborious manual testing process. This can help cut down on unused variables, unreachable code, and inefficient algorithms bloating the source code. I introduce a test suite to ease verification for development and refactoring.

With many languages, there are many valid and efficient ways to accomplish one thing. Typescript offers several ways to assign `foo` to a value of 10: for example, a developer could write `const foo: number = 10;` or `FOO = 10`. While these two snippets generally accomplish the same task, the first approach is significantly preferred over the latter because it gives the reader (whether it is a human, the TypeScript compiler, or the JavaScript interpreter) much more information about the volatility, scope, and type of `foo`. Additionally, using (mostly) lowercase letters for local variables and reserving all capital letters for global constants helps enforce a sense of style consistency that eases the flow of reading. To enforce good coding practice, I introduce a linting step to the testing process.

Ultimately, there is no substitute for human supervision when contributing a feature. Even with test cases and linters in place, it is possible to fail edge cases (possibly by omitting a critical test case) and contribute unnecessarily convoluted implementations. To improve the human aspect of development, I worked with fellow MEng

student William Qian to establish a reasonable version control workflow for developers to follow, discussed in section 4.3.

2.3 Performance Engineering the Engine

At the heart of StarLogo Nova is an engine that executes the user scripts. A more efficient engine enables users to simulate more agents and more complicated behaviors and cuts down on waiting for simulation results. Students can take advantage of the time saved by faster simulation times to test new hypotheses and see just how far their decentralized designs can scale.

To render a simulation in real time at 60 frames per second (the typical refresh rate of a monitor), the engine can take no more than 16.6 milliseconds to execute a slice of all agent scripts. To allow time for rendering the actual image, the time budget per step drops to about 10 milliseconds. The previous engine was unable to reliably meet this deadline so the engine was run only once every 12 frames, with the renderer interpolating between agent states to maintain the illusion of rapid execution. This leads to undesirable lag with games receiving keystrokes up to 11 frames (or 200 milliseconds) after they have been pressed.

I performance engineer the engine to make it run in real time, taking advantage of the incredible performance advances in browser JavaScript engines.

Chapter 3

Flash Migration

JavaScript by itself is a generic programming language with little to no type safety. The runtime will silently allow users to add a string to a number and put `falsey` values (sometimes 0, sometimes undefined, sometimes NaN) onto the stack for other operations, ultimately causing unexpected errors downstream. As such, large JavaScript codebases exceeding a few thousand lines of code can be difficult to work with because there are few mechanisms to protect the programmer from teammates or even past versions of him or herself. Instead of reimplementing the core engine in JavaScript, TypeScript (TS), an open source gradually typed programming language initially created by Microsoft that transpiles into JavaScript, was chosen as the development target. The type safety checks are done at compilation time, helping reduce the risk of introducing bugs even before any testing is done.

3.1 Initial Migration Steps

ActionScript3 and TypeScript look similar syntax-wise since they were both developed to be type-safe object-oriented extensions of ECMAScript, the official standard for JavaScript. Because of this, it is possible to automatically convert the bulk of the code with naive string substitution. `As3-to-typescript`¹, an open source tool for automatically translating AS3 to TS, was used to do exactly this. Although it automated

¹<https://github.com/photonstorm/AS3toTypeScript>

many mundane substitutions, its shallow understanding of the underlying code led it to make frequent errors in translation.

I was unable to find a conversion tool with a deeper understanding of the underlying abstract syntax tree of the AS3 code so I did the rest of the migration process manually. I chose to first migrate files with few to no dependencies because it allows for immediate testing of modules whose dependencies have already been satisfied. This decision traded off initial understanding of the big picture for immediate results, but was worth it in this case because getting small modules to fully compile helped me learn the TypeScript language.

To migrate from the bottom up, it is necessary to identify which files constitute the bottom and have satisfied dependencies. The next step in migration was to construct a dependency tree of the AS3 files to identify which files are leaves, which files whose dependencies would be satisfied by migrating those leaves, and so on. Although leaves were straightforward to identify, it was actually impossible to construct a tree because AS3 supports circular imports (and the codebase made some use of them), whereas TypeScript imports do not natively work well with circular references².

After migrating leaves and some nodes with few dependencies, I refactored parts of the codebase to eliminate import cycles. One technique I used was to create an interface for popular classes so that files that just require the type for declaration purposes can import the interface rather than the class itself. This resolves the case where the **Agent** class relies on some file that passes an **Agent** around and requires the agent type just for the type declaration. For more strongly coupled files, I resolved circular imports by merging the two files and having two class declarations within the same file.

²There are import modules for TypeScript that support circular references, but I did not explore configuring those because support for circular dependences can encourage unnecessary coupling.

3.2 Migrating the Engine Core

The most complicated part about the StarLogo Nova engine is how it actually runs the scripts which control the agent actions. ScriptBlocks stores each script in a tree-like data structure, but the engine actually stores each script in a list-like data structure of `StaticInstructions`. Each script is run in the context of a thread, which contains useful attributes such as a reference to the agent, the agent that this agent collided with (if any), and a data stack. `StaticInstructions` all have a `fn` method, which can pop arguments off the thread's data stack and push a return value onto the stack. The `fn` method also returns a reference to the next `StaticInstruction` to run. In order for `StaticInstructions` to know what `StaticInstruction` to return, they are given references to nodes that may succeed them during an initial compilation step. Figure 3-1 and 3-2 show the original AS3 implementation for the `If` and `CalcSum` instructions.

There is a special `yield` instruction that signals to the Engine that this thread is finished running for the current cycle and the Engine should begin executing the next thread. When the Engine returns to this thread in the subsequent cycle, it should resume execution as if it had not been stopped at all - the stack state and instruction pointer should be preserved.

3.2.1 A Difficulty in Migrating the Engine Core

Compiling scripts into a list of `StaticInstructions` from scratch was measured to be computationally expensive. One feature that a previous contributor had worked on was incremental compilations: this feature listens for change events on the blocks and mutated the instructions list accordingly. Unfortunately, the migrated TS threw errors during incremental compilations for certain input sequences of modifications, in spite of repeated checking that the TS mirrored the AS3. Upon closer inspection, the AS3 feature itself also exhibited the same bug as TS³.

³Fortunately, incremental compilation did not make its way into production so this was an undiscovered bug in the development branch.

Figure 3-1: AS3 Interpreter code to compute if

```
public class If extends BranchingStaticInstruction
{
    public function If()
    {
        super();
        branchLocations = new <int>[1];
    }

    override public function fn(a:Agent):StaticInstruction {
        var test = Boolean(a.thread.dataStack[--a.thread.dataStackIdx]);
        // if the test is true, pass execution to the body. Otherwise,
        // move on to the "next", which is whatever comes after the if.
        if (test && branches.length >= 1 && branches[0] != null) {
            return branches[0].first.head();
        } else {
            return next;
        }
    }
}
```

Executing `fn` pops a `test` value off of the thread data stack. If `test` is evaluated to be `true`, then it returns a reference to the head of its branches list. Otherwise, it returns a reference to the instruction that follows the `if` instruction.

Figure 3-2: AS3 Interpreter code to compute CalcSum

```
public class CalcSum extends StaticInstruction
{
    public function CalcSum()
    {
        super();
    }

    override public function fn(a:Agent):StaticInstruction {
        var tempVal2:Object = a.thread.dataStack[--a.thread.dataStackIdx];
        var tempVal1:Object = a.thread.dataStack[--a.thread.dataStackIdx];

        var tempNum1:Number = Number(tempVal1);
        var tempNum2:Number = Number(tempVal2);

        if (isNaN(tempNum1) || isNaN(tempNum2)){
            // this is going to be a string append operation so we add in
            // reverse order that values were popped from the stack.
            a.thread.dataStack[a.thread.dataStackIdx++]= tempVal1 + tempVal2;
        } else {
            // order doesn't matter because it's a numeric sum
            var sum:Number = tempNum1 + tempNum2;
            a.thread.dataStack[a.thread.dataStackIdx++]=sum;
        }
        return next;
    }
}
```

Executing `fn` pops two values off of the thread data stack: `tempValOne` and `tempValTwo`. `fn` then tries to convert these values to numbers. If successful, it adds their sum to the stack. Otherwise, it tries to do string concatenation (using the built-in `+` operator) on the two temporary values and pushes the new value onto the stack. At the end, it returns a reference to the instruction that follows this `CalcSum` instruction.

This feature requires very precise handling of the `Instruction` list data structure to work in every case. There were complicated edge cases involving detachment of procedure calls with a corresponding complicated implementation. Furthermore, the implicit structure of the script made it difficult to determine whether the nodes were correctly attached during debugging.

Because I did not know the full extent of how incomplete the incremental compiles were and because of the absence of a test suite, it was unclear how difficult it would be to fix incremental compilation. There could be additional undiscovered bugs in the codebase: the mechanism that `ScriptBlocks` uses to send update signals is both complicated and not fully tested.

Additionally, the current execution design is difficult to scale up for efficiency. Because the StarLogo Nova interpreter must run in JavaScript, it is very difficult to make an implementation of the addition block competitive with an `add` assembly instruction. Since we intend to replace this current execution design with a faster approach, it is not worth investing a significant amount of time into fixing this old approach and immediately discard it.

Instead of migrating the complicated model of a linked `StaticInstruction` list and trying to do what the original author failed to accomplish, I designed a different approach to interpreting user scripts, with an emphasis on correctness and ease of debugging. After correctness is established, I could focus on performance and eliminating bottlenecks.

Even if the interpreter is ultimately not used because of performance limitations, it centralizes execution, which can make instrumenting execution for tracing and debugging purposes simple. It also introduces a much needed AST structure to provide a solid backbone for scripts.

3.2.2 A Simpler Interpreter

The `ScriptBlocks` representation of the blocks is designed for displaying the blocks and responding to user interaction, but not for the actual interpretation of the code. For each block, `ScriptBlocks` maintains a list of sockets that children blocks can plug into,

but does not differentiate between argument blocks (blocks that are consumed by the execution of that block) and branch blocks (blocks that can be conditionally executed after that block). Furthermore, some blocks like `procedure` and `list` contain `+` and `-` buttons that are also represented as sockets but are not used by the engine.

This lack of differentiation requires instructions themselves to explicitly use a stack to pass arguments and return values around. The choice of using a stack allows small bugs, such as an extra push or pop from the stack, to easily corrupt the execution state. Instead of using the `ScriptBlocks` representation of blocks as the basis for the new interpreter, I constructed an explicit abstract syntax tree (AST) for each script. This provides a layer of insulation between `ScriptBlocks` and the Engine to reduce coupling, allowing future contributors to change the `ScriptBlocks` interface or even use another library, such as `Blockly`[2], to handle the user interface.

This explicit AST structure differentiates between arguments and blocks and is designed for fast access during interpretation. An AST is composed primarily of two data structures: an `ASTNode` and `ASTList`. `ASTList` inherits from `Array<ASTNode>` and contains some useful helper methods such as validating that all `ASTNodes` are correctly set up. `ASTNodes` have an `ASTList` instance for all of their arguments and an `ASTList` instance for each branch. This AST structure allows the interpreter to explicitly gather the return values of a list of argument nodes, allowing for easy debugging.

I replaced each `Instruction` definition with one that extended the `ASTNode` class and changed the input output model of the execution of the `Instruction`. Instead of reading arguments from the stack and adjusting the stack pointer, they would instead read arguments from their `ASTList` arguments member. After the execution, the `Instruction` would write two values in a list⁴: the return value of the instruction (if any), and an integer representing control state to the interpreter. These control states are described later in this section. Figure 3-3 shows the `If` and `CalcSum`

Figure 3-3: Simpler Interpreter code to compute if

```
export class If extends ASTNode {
  constructor() {
    super(1, 1); // numArgs = 1, numBranches = 1
  }

  public fn(a: Agent, scope: Map<string, any>,
            args: Array<any>, rets: Array<any>): void {
    if (args[0]) {
      rets[1] = 0;
    } else {
      rets[1] = Constants.AST_DONE;
    }
  }
}
```

Executing the `if`'s `fn` is simpler now. If the first argument is true, then it will write a 0 into the control state, otherwise `Constants.AST_DONE`, which is a negative integer. If the interpreter reads a nonnegative value in the control state, it will take that as the branch index and execute all the nodes in the branch. Otherwise, it will function according to the specific control state.

instructions implemented as `ASTNodes`.

This shifts the burden of running user scripts from the compiler to the interpreter. Instead of having a medium level of complexity across all the instructions, this approach reduces the complexity across all the instructions and centralizes complexity at the interpreter. For each thread, a JavaScript generator is constructed representing the state of the interpreter. This generator serves a critical role in the design because it pushes the burden of saving the execution state upon yield to the browser JavaScript engine.

The interpreter provides two generator methods: `executeNode` and `executeList`. `executeList` effectively calls `executeNode` for each element in the list. If any nodes

⁴Returning the two values in a list could increase pressure on the garbage collector so the interpreter instead passes a reference to a return array to all nodes it returns. However, some modern browsers (see section 5.1.2 for an overview of how the Firefox SpiderMonkey JavaScript engine does this) can automatically deconstruct this return array into individual values defeating the purpose of this optimization. This assumption about garbage collection performance should be revisited in the future

Figure 3-4: Simpler Interpreter code to compute CalcSum

```
export class CalcSum extends ASTNode {
  constructor() {
    super(2, 0); // numArgs = 2, numBranches = 0
  }

  public fn(a: Agent, scp: Map<string, any>,
           args: Array<any>, rets: Array<any>): void {
    rets[0] = a + b;
    rets[1] = Constants.AST_DONE;
  }
}
```

Executing `CalcSum`'s `fn` is also simpler now.

return `RETURN PROCEDURE` as their control state, then `executeList` returns immediately and propagates the `RETURN PROCEDURE` control state.

`executeNode` will first set up an infinite loop. For each iteration of the loop, the generator will call `executeList` on the `ASTList` representing the arguments of this node, storing return values in an array. Then, it will pass those return values as the parameters to the `ASTNode`, call the main function, and look at the output of the main function. Depending on the control state, the interpreter will take different code paths.

A nonnegative integer: it will run `executeList` corresponding to this branch index. If `executeList` returns `RETURN PROCEDURE` as its control state, then `executeNode` will return and propagate the `RETURN PROCEDURE` state upwards. Otherwise, it break out of the infinite loop and return the `DONE` control state. All other control states are negative integers.

`DONE`: it will break out of the infinite loop to execute the next node.

`YIELD`: it will literally yield the generator and then break out of the infinite loop when execution resumes.

REPEAT: it will look at the return value. If it is nonnegative, then it will construct a finite loop to repeat running this node's first branch that many times. If the return value is negative, then it will run this node's first branch and continue to the next iteration of the main infinite loop.

YIELD AND REPEAT: it will yield the generator and then continue running the loop when execution resumes.

CALL PROCEDURE: it creates a new generator based on the procedure. Then, the parent generator yields on this new generator, which has the effect of stepping through the child generator and yielding each time the child yields. This child generator retrieves the procedure parameters from the scope and then recursively constructs a generator to execute the main `ASTNode` holding the procedure.

RETURN PROCEDURE: it will break out of the infinite loop. This signal is propagated to the the runner of the `ASTList`, which will halt execution and return from the generator running this procedure.

The recursive nature of this interpreter design helps an implementor reason about the interpreter and debug any problems. Because of the straightforward design, I was able to build the interpreter, performance engineer it (section 5.3), and simulate a paintball game featuring dozens of moving targets, in the span of a week.

A major drawback of this interpreter is that it is incredibly cautious and creates complicated constructs to accommodate all possible edge cases. For example, the infinite loop is constructed only to handle the `repeat` block case. Consequently, early benchmarks revealed that the execution speed of this interpreter lagged the AS3 code by over an order of magnitude. There is room for many opportunities for optimizations down the line, and I describe the application of optimization techniques to this interpreter in detail in section 5.3.

3.3 Miscellaneous Migration Details

This section describes some minor design decisions that were also made in the migration process.

3.3.1 JavaScript Maps vs Objects

Hash maps data structures are implementations of an associative array, optimized for rapid key access. Given a key-value pair to store, a hash map will compute a numeric hash of the key and place the key-value pair into a bucket corresponding to that hash. For lookup of a value given the key, the hashmap will compute the numeric hash of the key and search through the appropriate bucket for the key.

The original AS3 code relied extensively on hashmaps to keep track of data. In JavaScript, there are two primary implementations of a hashmap: (1) a general `Map` class introduced in the 2015 ECMAScript 6⁵ update and (2) objects which can map strings to arbitrary objects. The latter is used almost ubiquitously by web developers because they have been widely supported for years so browser vendors have invested major resources to tune their implementations of JavaScript objects. In contrast, `Maps` are just beginning to see adoption and the current implementations of `Map` objects often trail the performance of objects in benchmarks.

However, using objects as a hashmap also has its drawbacks. Objects can only use strings as keys and consequently, it is necessary to serialize objects to strings before using them as keys. Furthermore, two references to the same object must serialize to the same key, but two copies (but not references) of the same object must serialize to distinguishable strings, increasing implementation complexity. Furthermore, TypeScript allows developers to declare the types of keys and values for maps, but not for objects.

In light of these benefits and tradeoffs, I choose to generally use `Maps` for hashmaps because of the immediate benefits of type safety. If it is later discovered that some specific `Map` access pattern is a bottleneck, then it is possible to use a more opti-

⁵JavaScript is an implementation of the ECMAScript standard

mized data structure, but optimizing for performance at the migration phase is too premature, particularly for an interchangeable component.

3.3.2 Singletons and Static Classes

AS3 had awkward support for static methods so previous StarLogo Nova contributors constructed singleton instances. Instead of reimplementing these singleton instances faithfully, I chose to implement them idiomatically in TypeScript.

As a general rule, I took the liberty to implement the engine as idiomatically as I could in TypeScript because there are few benefits to keeping the style of the AS3 code.

3.3.3 BlockTable

The engine makes use of a BlockTable to translate between ScriptBlocks names for blocks to the Instructions. Based on the name, one would expect the translation to look up the construction function from the name of the instruction and return a newly instantiated instruction. However, the AS3 BlockTable operated more like a complex switch statement and executed many compilation steps in the BlockTable.

I refactored BlockTable to reduce it to a map from string (instruction name) to `ASTNode` constructor. Previously, the AS3 engine had a 1 to many map from ScriptBlocks blocks to engine Instructions and this refactoring makes the mapping 1 to 1⁶, significantly reducing the amount of setup logic in the BlockTable. This reduction in complexity focuses debugging of the compilation stage onto just the Compiler and `ASTNode` constructor.

3.3.4 Key Manager

StarLogo Nova uses a KeyManager keep track of keystrokes entered into the display for the `key-held` and `key-typed` blocks (section 1.7.5). The KeyManager maintains

⁶For example, `compare-equals` and `compare-not-equals` would map to the same instruction, but would pass in a parameter to the constructor of the instruction to invert the instruction return value in the case of `compare-not-equals`

a set of currently pressed keys and a queue of keys that have been pressed: `onKeyDown` adds the key to the set of keys, adds the key to the queue if the key was not already in this set of keys, and prevents the default behavior; `onKeyUp` removes the key from the set of keys.

Preventing the default behavior is necessary to prevent arrow keys and other keys from scrolling the page. However, we do not want intercept `super-f` and prevent the user from searching the page. Thus, we do not prevent the default behavior for keys with the meta key on.

One initial problem with this approach is that for some `onKeyDown` events, we never receive the corresponding `onKeyUp` event. For example, if a user presses `super-t` and opens a new tab, the browser will switch focus away from the StarLogo Nova tab and the KeyManager will not see the `onKeyUp` event, even if the user switches back to the StarLogo Nova tab. To remedy this bug, I clear the set of pressed keys 100 milliseconds after a meta key is pressed in combination with another key.

3.3.5 Dead Code

Some parts of the codebase were deprecated and unreachable. For example, there used to be separate instructions for reading and writing an agent's `x`, `y`, `z`, and other traits. These instructions have since been replaced by two instructions, one for reading traits and another for writing traits. Instead of transpiling the dead code and keeping it, I verified that the dead code was indeed unreachable and deleted it.

Dead code can make the code base confusing and potentially slow down the performance of the engine. If it is necessary to revisit old implementations, one can always use version control to view a deleted function and potentially restore it.

3.3.6 General Statistics

The original AS3 engine contains 10,899 lines and 328,442 characters of code split across 172 files. This count excludes code for rendering and media operations, which is not part of the new engine.

In contrast, the migrated TypeScript engine contains 10,220 lines and 314,196 characters of code split across 141 files. This count excludes code for test cases, which was not part of the old engine.

The migration process did not significantly alter the size of the codebase. However, the resulting object payload sizes are dramatically different. The compressed AS3 swf file is over 300 KB, whereas the compressed TypeScript engine (after being compiled to JS) is under 50 KB. This TypeScript engine can be further compressed with whitespace minification, comment removal, and other tricks to reduce its footprint.

Chapter 4

New Tooling

I instrument some tooling to make it easier to keep the core engine healthy and reduce the potential for future technical debt. Tooling helps multiply developer efficiency, but may also steepen the learning curve for a new contributor. I carefully choose easy-to-understand tools that boost efficiency. In the following subsections, I describe specific pain-points and the rationale behind choosing specific tools.

4.1 Migration Tools

To ease the migration from AS3 to TypeScript, in addition to completing a first pass with the automatic transpiler, I incorporated several tools into the build process. Although these tools started showing their utility in the migration process, these tools will help future StarLogo Nova contributors create new features without accumulating large amounts of technical debt.

4.1.1 `make` and `tsc`

Writing large quantities of code without validation introduces enormous opportunities for errors. During the migration process, I wanted assurances that I was going on the right track, but did not have reference unit tests to run. In place of testing, I would frequently run my code through the TypeScript compiler (`tsc`) just to ensure that

what was written was in the correct language. This validation caught many bugs and actually saved time since the compiler errors pointed out code that I had just been working on so I could rapidly fix bugs in the moment. However, manually validating files is a nuisance because it requires repeatedly typing out `tsc -out /dev/null file/to/be/compiled.ts` for each edit, no matter how minor. If I were to make five quick edits to files, I needed to painstakingly go back and recompile each file. Furthermore, it is difficult to imagine that all future contributors would go through the trouble of doing this check for every change so I automated this task.

I created a `Makefile` with an `all` target, which initially included a serial list of instructions to compile the individual files, excluding those that were unfinished. However, this approach has three primary drawbacks: 1) files that were ready to be compiled needed to be explicitly added to this list of instructions, 2) files that were already built were wastefully rebuilt each time, and 3) this list of instructions imposes a serial order on execution, restricting modern multicore machines to use a single core for compilation.

Instead of using a list of instructions, I instead created a list of TypeScript source files and explicitly filtered out unfinished files. This choice of blacklisting files from being compiled as opposed to whitelisting files to be compiled ensures that the contributor is aware of all files that are not being compiled and what work remains to be done. If there is a new file created, it will automatically be incorporated into the build process, unless explicitly excluded.

Before `make` builds a file, it will first check for the existence of and modification time of the target. If the target exists and the source has not been modified since it has last been built, then `make` will not build this file. Since contributors will usually only edit a small number of files, this will save a significant amount of build time.

The `make all` target depends on all of the source files in no particular order between files. This means that although the steps to build each file have a serial dependency, `make` is free to build files in any order. In particular, this means that `make` is allowed to schedule build stages for multiple files simultaneously, allowing the build process to be trivially parallelized. To run a build, a contributor can run

the command `make all -j <number of concurrent jobs>`. Figure 4-1 shows the critical parts of the Makefile.

However, `make` does not understand the import structure of TypeScript source files so a successful `make all` can still leave bugs if a changed dependency breaks something upstream, but the upstream target has already been built successfully and not changed since. To combat this, contributors should run incremental `make alls` as quick sanity checks and a full `make clean; make all` prior to making a commit in version control as a more comprehensive check¹.

4.1.2 `tslint`

While the TypeScript compiler checks for matching types at runtime, it can be very lenient because it permits variables to be untyped. This design decision was made to ease the transition of an existing JavaScript codebase into TypeScript, but this leniency does not significantly benefit StarLogo Nova because the new execution engine fully embraces TypeScript. Consequently, I also introduce the use of `tslint`, an open source code linter for the TypeScript language created by Palantir. `tslint` requires all variables to have a type declaration, effectively strictly typing the language.

`tslint` also offers many other checks that keep the codebase in good shape. It enforces consistent whitespace and variable naming practices, alerts the user of unused variables and imports, and forbids potentially bad practices such as the use of `eval`. One of the biggest benefits of the linter is that it helps the source of core engine look consistent, as if it were written by a single person in a single sitting. This helps future developers understand the codebase more easily and also reduces decision fatigue: there is now much fewer ways to write code.

To enforce a consistent style guide for the codebase, I add a `tslint` check to every file during the `make` process.

However, there are cases where it is difficult to do exactly as the linter wants.

¹While it is possible to configure `make` to include dependency information, the effort to keep that dependency information up to date outweighs the cost of running a comprehensive build. This assumption may be invalidated if there is a significant increase in the amount of code that needs to be compiled or if future versions of the compiler significantly slow down.

Figure 4-1: Makefile all Target

```
TSC = tsc
TSC_FLAGS = --target es6 --noEmitOnError
TSFMT = tsfmt
TSLINT = tslint

INCOMPLETE = Compilation/Instructions/CameraTake.ts \
Compilation/Instructions/ShapeOptions.ts \
Compilation/Instructions/SoundPlay.ts \
Compilation/Instructions/SoundPush.ts \
Execution/PopUpUtils.ts \
WebLand/BreedEditor.ts

SOURCES = $(filter-out $(INCOMPLETE), \
$(wildcard *.ts) \
    $(wildcard */*.ts) \
    $(wildcard */*/*.ts))

OBJECTS = $(SOURCES:.ts=.js)

all: $(OBJECTS)
    echo "SUCCESS"

%.js: %.ts
    $(TSFMT) --verify $< || $(TSFMT) -r $<
    $(TSLINT) $<
    $(TSC) $(TSC_FLAGS) $< -out $@
```

`SOURCES` is a list of all the typescript files, with the `INCOMPLETE` files filtered out. `OBJECTS` is a mapping of sources to targets. The `%.js: %.ts` target provides instructions on how to build each file: 1) the file is automatically formatted (discussed more in section 4.1.3), 2) the file is verified by the linter (section 4.1.2), and 3) the file is compiled. The flag `-noEmitOnError` prevents the file from emitting the target if there is an error during the build process. If this flag did not exist, then `make all` would fail the first run, emit the target, and skip building the target on a subsequent run, effectively allowing the failed file to skip the build process. It is also important that the compiler is run last, after the formatter and linter. If the compiler ran and successfully produced an output file and the linter subsequently failed, `make` would produce an error and abort the current process. However, on the next `make all`, `make` would detect that the target has been built and skip rebuilding the target, sidestepping the build and verification process for that file.

Figure 4-2: Overriding `tslint`

```
let color: Array<number> = [  
  /* tslint:disable no-bitwise */  
  ((this.state.color & 0xFF0000) >> 16) / 255,  
  ((this.state.color & 0x00FF00) >> 8) / 255,  
  ((this.state.color & 0x0000FF) >> 0) / 255,  
  /* tslint:disable no-enable */  
];
```

The `/* tslint:disable no-bitwise */` comment disables the `no-bitwise` linter rule until the `/* tslint:enable no-bitwise */` comment.

One common case where we want to disobey the linter is if the best practice harms performance (performance engineering is described in section 5.5.2). For example, one can declare variables in TypeScript with the `var` keyword, but these variables are scoped to the function, not to the local block. Block scoping can prevent some bugs, but is deleterious for performance in critical loops. Another example of something that `tslint` does not permit the use of bitwise operators because they are infrequent and likely typographical errors for boolean operators. However, some manipulations are best expressed with bitwise operators and we know better than the linter.

Instead of disabling the linter check everywhere, I explicitly turn off `tslint` for sections of the code that are in violation of the linter for good reasons. Figure 4-2 shows an example of this mechanism for bitwise operators. Again, this style of opt-out as opposed to opt-in helps keep the build process relevant and useful for the majority of files.

4.1.3 `tsfmt`

While there are many benefits to making code look consistent, there is one immediate major downside: it is time-consuming. Making code look consistent requires several mindless edits in throughout the codebase. As I went through the files and corrected errors made by the transpiler, I found myself spending unreasonable amounts of time manually fixing whitespace inconsistencies line by line.

Figure 4-3: `tsfmt` invocation

```
$(TSFMT) --verify $< || $(TSFMT) -r $<
```

This runs the formatter in verification mode on the source file. If verification fails, then the formatter is run in replace mode, replacing the source with a formatted copy. `$<` represents the input file name.

This costly process of manually fixing of code will only clean up the state of the codebase in the short term and future contributors may resent this process so much that they will just disable linting for large swaths of code or even delete the linting process entirely.

To make future contributions to the codebase painless and consistent, I add an automatic TypeScript formatter (`tsfmt`) to the build process. I lightly configure `tsfmt` to adhere to existing style conventions. The addition of the formatter also sped up the remainder of the migration process since I did not have to carefully edit my code - a contributor could be sloppy and let the formatter take care of the rest.

Although `tsfmt` is good at adjusting whitespace to align code, it cannot perform more complicated edits such as adding type annotations or automatically renaming unconventional variable names. Even with `tsfmt` as part of the build chain, it is necessary to keep the lint check for further code validation.

The invocation of `tsfmt` in the `Makefile` (reproduced in figure 4.1.3) looks a bit strange and requires some explanation. The command `tsfmt -r <source file>` will format the source file and replace whatever was there previously, even if `tsfmt` does not make any changes. This will bump the modification time of the source file to the current time and for subsequent builds, `make` will detect that the file has been modified since the last build and proceed to build the target from scratch. This significantly increases duration of the build process since each build effectively becomes a full build. To combat this, I first run `tsfmt` in verification mode and if that fails, then I run `tsfmt` in replacement mode.

4.2 Automatic Testing

Test cases are fantastic for catching regression bugs early on. As the number of features in StarLogo grows, it becomes intractable for a human to try and verify that all previous features still work. Test cases can automatically verify intended previous behavior and reveal regressions immediately. This saves time because the contributor remembers exactly what change they made introduced the bug at the time of the automated test and prevents end users from discovering the problem.

Test cases also promote good coding practices internally. In order to make a function easily testable, contributors must write their functions with minimal side effects, a small number of arguments, limited scope, and a small set of responsibilities. They can also serve as documentation on how to use a certain function.

In many environments, a testing workflow consists of running a test command into a shell session and receiving the output of the test. While it is possible to use `node` to automatically test the generated JavaScript, this only tests the code for Chrome's V8 engine. This provides insufficient coverage for platforms such as the iPad. For client-facing JavaScript such as the core engine, it is necessary to test with actual clients to ensure cross browser compatibility.

I create internal test suite using `Jasmine`, which is a JavaScript testing framework. Under `Jasmine`, source files and test specs are imported as scripts on a `html` page and the `Jasmine` library evaluates the test specs and displays the result in a friendly format to the user. To quickly run this test, I also create a Selenium script to launch a headless browser to visit the test page and report back the results.

4.3 Version Control Workflow

The best internal tools can still let bad code slip through the cracks. Thus, it is vital to have a good procedure for code review and version control in place to facilitate human sanity check.

I worked with William Qian to put together a document helping new contributors

work with git. Effectively, we heavily endorse the feature branch workflow where contributors branch off of the development branch to create their feature and issue pull requests to merge their feature branch back into the development branch. There is a separate production branch that picks commits from the development branch to merge in.

When a contributor begins a feature, they create a new branch named with the format `yyyymmdd-username-feature_description`. This specific format helps keep track of who owns what branch, how old it is, and what the feature is.

Then, they do their work in that branch, rebasing their feature branch on the tip of development as other pull requests land on development. Contributors are encouraged to checkpoint their progress frequently with small commits. Small commits are then expected to be squashed into a small number of large commits describing their full feature. After testing their feature and running the full test suite, they issue a pull request and work with an experienced contributor to review the pull request and merge in the code. Upon merge, that feature branch should be deleted globally.

Chapter 5

Performance Engineering the Engine

The faster the StarLogo engine can run a script, the more agents it can simulate and the faster it can deliver results to the user. This can reduce the amount of time the user spends waiting for the simulation to complete and directly lead to more exploration of hypotheses and learning experiences.

This section first presents an overview of how JavaScript engines work (section 5.1) and presents how benchmarks were conducted to minimize skew and variance (section 5.2). Then this section describes both successful and unsuccessful attempts at speeding up the execution of the core engine (sections 5.3 through 5.5).

5.1 Overview of JavaScript Engines

JavaScript is a high-level, untyped, garbage collected, and interpreted programming language that is generally executed on browsers. Running JavaScript quickly is a difficult feat and modern JS engines are marvels of engineering. The previous generation of JS engines relied on interpreting JavaScript directly and the current generation of JS engines implement Just-In-Time (JIT) compilation of JavaScript to native code for greater performance.

This section gives an overview of how major JavaScript engines for Chrome, Firefox, Safari, and Edge (formerly known as Internet Explorer) operate in greater detail.

5.1.1 Chrome V8

The Chrome V8 JS engine utilizes two compilers: a full non-optimizing compiler that quickly emits native code and an optimizing compiler named Crankshaft. When a function is initially run, the non-optimizing compiler emits code that runs the function¹ and keeps track of the number of invocations of that function with a counting profiler². If a function is repeatedly run, Crankshaft will emit optimized code for that function.

When a function is repeatedly run and deemed hot, Crankshaft kicks in to emit more optimized code. Crankshaft operates in a series of stages: parsing the JavaScript into an AST representation, generating an architecture-independent control flow graph based on the AST and scope analysis, optimizing this control flow graph³, and generating architecture-dependent machine code. Currently, Chrome uses coarse-grained locking to protect the heap so Crankshaft must stop the world while generating code. Since nothing else can run while the world is stopped, Crankshaft must run optimize quickly and get out of the way, forcing it to trade off deep optimizations for compilation speed. This constraint may change in the future with more parallel versions of Chrome, but at present, it is sometimes necessary to supply redundant hints to help the optimizing compiler efficiently generate fast code.

When optimizing the control flow graph, Crankshaft tries to infer variable types based on types seen previously so it can emit efficient native machine instructions. For example, if it sees that an array only ever contains small integers, it may represent that array with an array of 16 bit integers and emit `int16` machine code for arithmetic operators on elements of that array. If the inferred type is wrong, then Crankshaft will de-optimize that function and fall back to the code generated by the non-optimizing

¹This non-optimizing compiler offloads much of the execution to runtime functions, not unlike a traditional interpreter.

²The advantage of the counting profiler over a sampling profiler is that the counting profiler is deterministic and lowers variance in performance, at a minor time cost.

³A control flow graph is a graph of the order of how JavaScript statements can be invoked. A node's predecessors represent all the nodes that can be executed immediately before this node, and a node's successors are all the nodes that can be executed immediately after this node. Control flow graph analysis can reveal dead code (partitioned nodes) that can be elided and inlineable code (nodes with indegree and outdegree 1)

compiler.

Crankshaft also attempts to inline small functions⁴ so it is beneficial to tersely implement functions. Crankshaft will also hoist invariant code out of loops, reducing the need for the programmer to manually hoist code.

V8 also features an incremental generational garbage collector, which targets newly created objects more aggressively and rarely scans long-lived objects. Chrome's internal scheduler schedules garbage collection to run when there is a large amount of time before the next frame draw, helping reduce jitter.

Crankshaft's successor, Turbofan, operates on similar principles. Its main differentiation from Crankshaft is a more layered architecture with clearer separations between JavaScript, V8, and the underlying CPU architecture. This clearer separation will allow for more sophisticated optimizations, such as instruction reordering, to run in realtime.

5.1.2 Firefox SpiderMonkey

The Firefox SpiderMonkey JS engine is based on a bytecode interpreter, a low latency JIT compiler named The Baseline Compiler, and an optimizing JIT compiler named IonMonkey. The SpiderMonkey interpreter is primarily a long function that steps through the bytecode instruction by instruction, using a `switch` statement to interpret each instruction. The interpreter is slow, but records type information to be used by the compilers later.

Functions that are somewhat hot are JIT'd by the Baseline Compiler, which is a fast compiler that generates partially optimized code and collects further statistics. IonMonkey takes a large amount of time to generate highly optimized machine code so incorrect type information is very costly.

Functions that are really hot are recompiled by IonMonkey in a separate thread and undergo complex optimizations, on top of type inference and inlining. One opti-

⁴Small functions are functions that have fewer than 600 source characters and can be represented in fewer than 196 AST nodes. Note that inlining is a binary optimization: a function is either inlined or it is not and that it can be sensitive to small changes to the code.

mization analyzes how objects are used: if the object's attributes are only statically referred to, the object can be deconstructed into scalars and all operations on those scalars are inlined. IonMonkey also looks at trace statistics and prunes rarely used branches to reduce code size and improve cache efficiency. If a rare branch is hit, then IonMonkey will bailout and fall back to the Baseline code.

SpiderMonkey also features a incremental mark-and-sweep garbage collector.

Firefox will introduce OdinMonkey, an Ahead of Time compiler, primarily aimed at running `asm.js`. `asm.js` is specially-crafted Javascript that contains type annotations that provide hints to JavaScript compilers on how to quickly compile it to native code.

5.1.3 Safari JavaScriptCore

The Safari JavaScriptCore⁵ JS engine is comprised of a Low Level Interpreter (LLInt), baseline JIT, low-latency optimizing JIT (DFG⁶) and a high-throughput optimizing JIT (FTL⁷).

The LLInt executes bytecodes produced by the parser and is designed to have minimal latency. It obeys calling, stack, and register conventions used by the other JavaScriptCore compilers to make lowering optimized code simple and also records type information to feed into later compilers.

The Baseline JIT is invoked for somewhat hot functions and generates correct code that may be potentially inefficient, similar to Firefox's Baseline Compiler. If any further compilers make optimizations that are too aggressive and incorrect, then JavaScriptCore will fall back to this Baseline JIT.

The DFG is a partially optimizing compiler invoked for hotter functions that inlines small functions and takes advantage of the stored type information to elide many type checks.

For very hot functions, JavaScriptCore will use a heavy duty compiler such as

⁵JavaScript Core is also known as SquirrelFish and Nitro

⁶DFG is an acronym for Data Flow Graph

⁷FTL is an acronym for Fourth Tier LLVM and Faster Than Light

LLVM or WebKit's Bare Bones Backend (B3) to generate machine code.

JavaScriptCore uses a generational garbage collector and can actually run the LLVM compilation in parallel with the garbage collector.

5.1.4 Microsoft Edge Chakra

The Edge Chakra JS engine is comprised of a profiling interpreter, simple JIT, and full JIT. As with other JavaScript engines, Chakra will first execute JavaScript with the profiling interpreter, optimize warm functions with the simple JIT, and fully optimize hot functions with the full JIT.

5.1.5 Summary of JavaScript Engines

All of these JavaScript engines operate based on similar principles:

1. Run code as safely as possible and collect profiling information on what functions are hot and what types variables are
2. Upon detection of frequently used code, try to speed it up by inferring type information, inlining code, and other optimizations. If these optimizations turn out to be incorrect, then fall back to the safer executor.
3. Upon detection of even more frequently used code, go back to the previous step and optimize more aggressively, trading off compilation time for JavaScript throughput.

All of these JavaScript engines go through great lengths to optimize type inference. Without type information, a simple `object.a + object.b` statement must go through a tremendous amount of cases:

1. If `object` overrides the default property access policy, then we must execute the overridden behavior.
2. Then, the engine must check the types of `object.a` and `object.b` through reflection. If either type is object, then the engine must call `valueOf()` on the

object to get its value to be used here. Calling `valueOf()` may trigger more recursive `valueOf()` calls.

3. Once the types of `object.a` and `object.b` are ascertained, if either operator is a string, the engine will perform string concatenation. If both operators are integers, then an `integer` addition is attempted. If this addition overflows or if either operator is a floating point number, then a `double` addition is performed.

In contrast, good type inference will allow the engine to skip all of these checks and simply run a machine instruction for addition (or more instructions for string concatenation)⁸. Thus, it is vital to help the JavaScript engine generate optimized code by using consistent types.

5.2 Setting up a Testbed

The benchmarks for guiding optimizations were executed on an Intel i7 M-5Y51 CPU running at 1.1Ghz. Although it is an "Intel i7" processor, this CPU is highly optimized for power efficiency and can be significantly slower than other processors, especially when Turbo Boost is disabled. Therefore, we expect the engine to run faster on students' machines than it runs on this machine.

Reproducible benchmarks can be difficult to conduct on modern processors because of a variety of factors. This section describes some of the techniques used to mitigate factors that may confound benchmarking results.

To optimized single threaded performance, modern multicore processors will shut down some cores to allow the remaining cores to scale up their frequency and take advantage of the increased power and thermal budget. However, they can only operate at the higher frequency for a limited amount of time before expending the thermal budget and will then reduce their frequency. This factor can contribute to great variances between consecutive runs of the identical code. To eliminate this factor, I turn off CPU frequency scaling for the test machine, preventing it from scaling up any

⁸We can think of type inference as really strong branch prediction.

CPUs. I verify that CPU frequency has been disabled with the Intel Power Gadget.

As mentioned in the overview, modern JavaScript engines will identify hot spots of code and optimize them. All user code continually run in StarLogo Nova are hot spots and we want to measure how quickly the user code runs at steady state with the most optimized compiler, as opposed to how quickly the code runs with less optimized compilers or how long the compiler takes. To ensure that we are not measuring cold unoptimized code or the compiler time, I run a few cycles of the test case to allow the JavaScript engine to generate optimized code.

Some of the StarLogo Nova code is nondeterministic: the scatter block randomly places an agent in a different location. This nondeterminism can significantly affect the state of the simulation and consequently the time it takes to run. Since there is no way to seed the pseudorandom number generator (PRNG) in JavaScript, for the purposes of this benchmark, I replace all calls to `Math.random()` with a simple seedable PRNG. This seedable PRNG is about twice as slow as `Math.random()` and using it results in conservative benchmark figures.

To keep testing complexity low, I test for performance exclusively on the Chrome browser on OSX. The majority of StarLogo Nova users either use Chrome as their primary browser or have Chrome installed.

There are a multitude of other confounding factors and it is difficult to individually address all of them. To be robust against other confounding factors, I make the assumption that the noise that these factors contribute to measuring is independent of when I test the code. Then, I ensure that all tests I run take at least a minute so that the rule of large numbers reduces the likelihood that a test result is significantly skewed by some confounding factor. To make a sub-minute test take at least a minute, I run it several times in a loop, and report the mean per-loop time here.

The script used to test is shown in figure 5-1. To report the time per cycle, we take the total number of iterations (8000 in the case of the example) and divide it by the time reported by `console.timeEnd`. The figures reported are effectively wall time and include the overhead of the profiler.

Figure 5-1: Benchmark Script

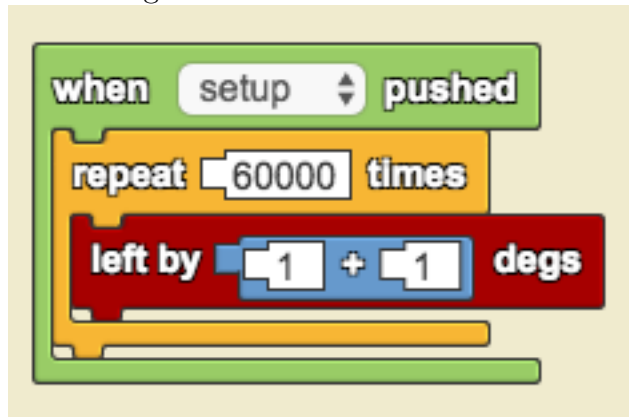
```
var num_it = 8000;
console.time(num_it + "it"); console.profile(num_it + "it");

for (var i=0; i<num_it; i++) {
    Execution.Engine.tick();
}

console.profileEnd(num_it + "it"); console.timeEnd(num_it + "it");
```

`console.time` and `console.timeEnd` are used to measure the time spent between the two statements. `console.profile` and `console.profileEnd` set up a profiler to inspect the time spent running the inner loop.

Figure 5-2: Rotate Benchmark



This is a simple benchmark to test the speed of the JavaScript interpreter.

5.3 Interpreter Optimization Attempt

This section describes optimization of the StarLogo Nova interpreter, originally described in section 3.2.2. Although this interpreter is ultimately succeeded by a more performant design (section 5.4), I describe the optimization of the interpreter as an example of how to optimize a critical section of code.

The basic interpreter design contains many opportunities to optimize. Daniel Wendel constructed a simple benchmark (figure 5-2) to measure the speed of the JavaScript interpreter relative to the AS3 interpreter.

Without optimization, the JavaScript interpreter ran this loop in 600 milliseconds. In contrast, the AS3 interpreter was able to run this loop in 30 milliseconds. The first thing I did to optimize the interpreter was to run the code under a profiler and find hotspots.

A large amount of time was spent constructing generators, but this is wasteful because this code does not yield. Because we expect that the majority of user blocks will rarely yield, it will be profitable to identify blocks of code that cannot possibly yield and create fast paths that do not require generators. Additionally, a large amount of time was spent logging so I removed logging statements. These optimizations brought the runtime down to 80 ms.

I expected the JS engine to inline the `executeList` function, which simply loops over a list and executes each node, but it did not. I manually inlined this function for the fast path, bringing the speed of the JS interpreter down to 25 ms and making it competitive with the AS3 interpreter.

I attempted to unroll the list evaluation code for small loops with a `switch` statement on the size of the loop, but this did not significantly affect runtimes.

Another trick I tried was to create very fast paths for simple nodes that simply emit `DONE` as their control output. At compilation time, it is possible to identify subtrees whose nodes only emit `DONE` as their control output and create fast paths without the infinite loop and run as ordinary non-yielding functions. Executing these subtrees can be further sped up by inlining recursive calls, significantly reducing the number of function calls in the tight loop above. This optimization brought the cost of the JS interpreter down to 10ms.

Table 5.1 summarizes the impact of these optimizations on the interpreter performance.

However, the interpreter was quickly gaining many lines of code and complexity with these optimizations. The choice of which path to use had to be very precise and it would be increasingly difficult to squeeze further optimizations out of this design. Furthermore, the interpreter at this point is highly optimized to run the fragment of code in figure 5-2 but may perform poorly in the general case.

Table 5.1: Interpreter Optimization Results

Optimization	% speedup
Create fast paths for nonyielding blocks and remove logging statements	650%
Aggressively inline <code>executeList</code>	220%
Create very fast paths for simple computation blocks <code>executeList</code>	150%

This table shows the incremental impact of interpreter performance optimizations.

An alternative approach is to take the AST structure and transpile that to JavaScript, similar to how browsers will compile JavaScript down to native code. This would put the difficulty of interpretation onto the host browser JavaScript engine, which is significantly more optimized than the StarLogo interpreter. I conducted a benchmark of the code in figure 5-2 and it completed in under a millisecond, showing promise for the transpilation approach. There are numerous other benefits to the transpilation approach, such as easier debugging for contributors, significantly higher performance in the short term, and the possibility of allowing advanced users to directly write scripts. There are embeddable JavaScript runtimes for V8 and other JavaScript engines so this design does not bind the StarLogo Nova to the browser: a future native StarLogo Nova implementation can take the browser JavaScript engine with it. With these benefits in mind, I pursue transpiling user scripts to JavaScript.

However, it is still worth preserving the interpreter implementation because there may be potential future use cases for it. Future use cases are described in greater detail in section 6.

5.4 Transpilation to JavaScript

To transpile the AST structure derived from `ScriptBlocks`, the first step is to create a string representing a generator for a given script.

5.4.1 Outline of how JavaScript is generated

The simplest model of having a single `to_js` method per `ASTNode` that returns a string representing the JavaScript code for that block is insufficient because procedures can yield before returning a value. For example, if an argument to an addition block were a call to a procedure that yields, then there is no straightforward way to call `to_js` on that procedure call and both yield main thread and return the value for use by the addition block.

Thus, for each block, I create two methods: `to_js_setup` and `to_js_final`. Whenever a block uses arguments, it must first call `to_js_setup` for each argument to allow the block's arguments to process and potentially yield. Then, it can emit a return value in `to_js_final` but is no longer allowed to yield in this second stage. This allows procedure calls to do all their yielding in the setup phase and for consumers of those procedures to gather the return value in the final phase. Figure 5-3 shows the implementation of the `If` instruction and figure 5-4 shows the implementation of the `ProcCall` instruction.

This approach leads to the construction of possibly many temporary variables because of the setup stack. To avoid collisions, all temporary variables are allocated with the prefix `__w1_<node.id>` so that even nested blocks of the same type reference distinct temporary variables. Figure 5-6 shows an example of StarLogo Nova blocks transpiled to JavaScript. Note how if `__w1_8_a` and `__w1_6_a` were just named `__w1_a` without the node identifier, then the emitted program would be incorrect.

5.4.2 Nuances to generating the JavaScript

The engine must be careful when executing code because there may be side effects. For example, the evaluation of `(a || b)` is expected to evaluate `b` only if `a` evaluates to `false`. Additionally, `a` is expected to be evaluated exactly once. While this detail does not matter for the majority of cases, procedure calls may affect state and thus may not be idempotent. Because of this, we must carefully generate JavaScript that only executes nodes at most once and skips execution in cases of short circuiting.

Figure 5-3: JavaScript Code Generation for If

```
export class If extends ASTNode {

  constructor() {
    super(1, 1); // numArgs = 1, numBranches = 1
  }

  public to_js_setup(): string {
    return '
      ${this.args[0].to_js_setup()};

      let __w1_${this.id}_cond = ${this.args[0].to_js_final()};
    ';
  }

  public to_js_final(): string {
    return '
      if (__w1_${this.id}_cond) {
        ${this.branches[0].to_js()}
      }
    ';
  }
}
```

Each argument is setup by calling the `to_js_setup` method and their return value is captured by calling the `to_js_final` method.

Figure 5-4: JavaScript Code Generation for ProcCall

```

export class ProcCall extends ASTNode {
  constructor() { super(1, 0); } // numArgs = 1, numBranches = 0
  public to_js_setup(): string {
    let ss: Array<string> = []; let fs: Array<string> = [];
    for (let i: number = 0; i < this.a.length - 1; i++) {
      ss[i] = this.a[i + 1].to_js_setup();
      fs[i]='let __wl_${this.id}_p_${i}=${this.a[i+1].to_js_final()}';
    }
    let fname: string = String((<UtilEvalData>this.a[0]).getData());
    let params: Array<string> = Procedure.getByName(fname).params;
    let args: Array<string> = new Array();
    for (let i: number = 0; i < params.length; i++) {
      args.push(","); args.push('__wl_${this.id}_p_${i}');
    }
    return '
      ${this.a[0].to_js_setup()};
      ${ss.join("; ")}; ${fs.join("; ")};

      let __wl_${this.id}_fn_n = ${this.a[0].to_js_final()};
      let __wl_${this.id}_fn=State.procGens.get(__wl_${this.id}_fn_n);

      let __wl_${this.id}_gen=__wl_${this.id}_fn(__wl_a, __wl_t
                                                ${args.join("")});

      let __wl_${this.id}_ret;
      while (true) {
        let iterRes = __wl_${this.id}_gen.next();
        if (iterRes.done) {__wl_${this.id}_ret=iterRes.value; break;}
        yield "${Constants.YIELD_NORMAL}";
      }
      ';
  }
  public to_js_final(): string {
    return '
      __wl_${this.id}_ret;
      ';
  }
}

```

The bulk of the setup work revolves around processing the arguments to be passed into the instruction. Some of the code has been edited to fit within the margins of this document by renaming variables and trimming whitespace.

Figure 5-5: Blocks for a Transpilation Example

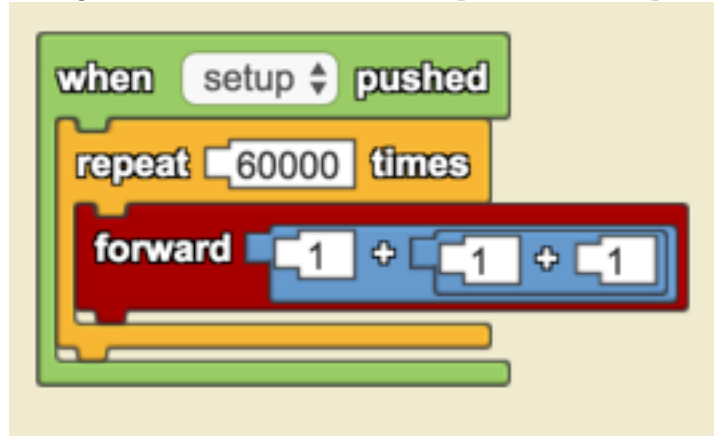


Figure 5-6: JavaScript for a Transpilation Example

```
f = function*(__wl_agt, __wl_scp) {  
  let __wl_1_btn_name = "setup";  
  while (true) {  
    if (Common.State.pushedButtons.has(__wl_1_btn_name)  
      && !__wl_agt.isButtonDisabled(__wl_1_btn_name)) {  
      let __wl_3_iters = 60000;  
      for (let __wl_3_i = 0; __wl_3_i < __wl_3_iters; __wl_3_i++){  
        let __wl_8_a = 1;  
        let __wl_8_b = 1;  
        let __wl_6_a = 1;  
        let __wl_6_b = __wl_8_a + __wl_8_b;  
        let __wl_5_amount = __wl_6_a + __wl_6_b;  
        __wl_agt.moveForward(__wl_5_amount);  
      }  
    }  
    yield "YIELD NORMAL";  
  }  
}
```

The emitted JavaScript for the blocks in figure 5-5. The emitted JavaScript shown has been cleaned up to make indentation consistent and remove unnecessary newlines and semicolons. The emitted JavaScript has been further edited to fit within the margins of this document by renaming variables and trimming whitespace.

To make this feasible for the fast path where we only have a single statement to ingest our arguments and emit a return value, I create helper functions that the generated JavaScript calls. While there are cases where it is possible to be excessively clever and do it in one line, it is not worth the confusion for a future reader.

When we execute the JavaScript string representing our program, calling `eval` on the function string is significantly slower than calling a regular function, even after the JavaScript engine gets around to optimizing the `eval`ed function. To run our JavaScript program string at full speed, I `eval` the declaration of the function, saving it into a `Map` of functions or generators. Because declaring a function once takes a negligible amount of time, this tradeoff significantly boosts the speed of executing scripts.

5.5 Optimizing the Transpiled Code

To optimize the transpiled code, I measure performance across a suite of computationally expensive StarLogo Nova projects. These key tests were chosen by Daniel Wendel as a representative suite of demanding user projects.

5.5.1 Benchmark Suite Description

The benchmark suite is comprised of 7 StarLogo projects that stress test different aspects of the Execution Engine.

10000 fish 10000 agents are spawned and scattered. For each cycle, they move forward by one. This simple benchmark tests for the overhead of simulating many agents.

www.slnova.org/djwendel/projects/308260/edit/

Analyse Up to 50 starch chains of 5 starch agents are created and chains move as a unit. The head of the chain can detach either with low probability or on collision with an active amylase enzyme. There are large numbers of simultaneous collisions and heavy trait interaction so this doubles as a correctness test in

addition to being benchmark test.

www.slnova.org/djwendel/projects/308261/edit/

Bacteria A single bacteria is created and wanders in the scene. If there are available resources locally, it will consume some of those resources and divide to generate new bacteria. If there are insufficient resources, it will die. The number of bacteria increase exponentially initially, reaching a peak of several thousand bacteria before declining, testing how quickly the engine can allocate new agents. This also tests the speed of terrain interaction.

www.slnova.org/djwendel/projects/308253/edit/

Yielding Dragon Fractal A single turtle traces the dragon fractal recursively. It will occasionally yield, returning control to the browser so the fractal can be rendered in realtime. This benchmark measures how fast the yielding performance is.

www.slnova.org/djwendel/projects/308255/edit/

Non-Yielding Dragon Fractal Similar to the yielding dragon fractal, but it doesn't yield. This benchmark measures the straightline performance of the engine.

www.slnova.org/djwendel/projects/308257/edit/






Paintball A player controlled by keyboard input can shoot colored balls at moving targets. When a colored ball and target collide, the target changes color and the ball is destroyed.

www.slnova.org/djwendel/projects/308259/edit/

Something's Fishy Up to 50 fish and hundreds of algae are created. The fish try to find algae and eat them to increase their energy levels to reproduce. Because algae periodically reproduce to replenish the supply of algae and fish continually die and respawn, this benchmark measures how the system responds to high agent turnover and memory churn.

www.slnova.org/djwendel/projects/308259/edit/

Figure 5-7: Profiler with no optimizations

Heavy (Bottom Up) ▾ 🔍 ⌕ ↺					
Self ▾		Total		Function	
24169.7 ms	36.91 %	29624.2 ms	45.24 %	▶  getJSThreads	app.is:7759
10184.6 ms	15.55 %	56746.1 ms	86.65 %	▶  tick_to_js	app.is:8038
6731.1 ms	10.28 %	6748.5 ms	10.30 %	▶ toString	app.is:7968
4664.7 ms	7.12 %	4664.7 ms	7.12 %	(garbage collector)	
4639.9 ms	7.08 %	4639.9 ms	7.08 %	▶ set name	
4016.2 ms	6.13 %	4016.2 ms	6.13 %	(program)	
2725.9 ms	4.16 %	5963.9 ms	9.11 %	▶  g	VM3330:1
1786.1 ms	2.73 %	7751.7 ms	11.84 %	▶ step	app.is:489
1321.8 ms	2.02 %	1548.0 ms	2.36 %	▶  getAllAgents	app.is:1272
1169.3 ms	1.79 %	1169.3 ms	1.79 %	▶ assert	
661.7 ms	1.01 %	661.7 ms	1.01 %	▶ set length	
581.8 ms	0.89 %	1818.3 ms	2.78 %	▶ isButtonDisabled	app.is:7964
561.4 ms	0.86 %	1406.9 ms	2.15 %	▶ moveForward	app.is:7922
548.1 ms	0.84 %	1165.5 ms	1.78 %	▶  runCollisions	app.is:8061
310.9 ms	0.47 %	310.9 ms	0.47 %	▶ bounce	app.is:7893
304.0 ms	0.46 %	304.0 ms	0.46 %	▶ cos	app.is:1
232.8 ms	0.36 %	232.8 ms	0.36 %	▶ push	
230.6 ms	0.35 %	230.6 ms	0.35 %	▶ sin	app.is:1
230.5 ms	0.35 %	230.5 ms	0.35 %	▶ <symbol>	

Breakdown of the most expensive functions, sorted by self time. The yellow triangles adjacent to some functions indicate that the JavaScript engine was unable to fully optimize the function. Note that the times are significantly larger than the 362 ms per cycle because this is in aggregate for the minute-long test.

Although we compare the speed of the compiled JavaScript against the AS3 interpreter, the goal is not to beat the interpreter. Rather, it is to try to bring the average execution speed under 10ms so StarLogo Nova can run the engine in real time.

5.5.2 Universal Optimizations

I first optimize the engine with respect to the first test in the benchmark, 10000 fish. At steady state, the engine completed 200 iterations in 72.4 seconds, averaging 362 milliseconds per iteration. This is about 2.5 times slower than AS3, which takes about 147 milliseconds per iteration at steady state. Figure 5-7 shows a screenshot of the Chrome profiler.

Interestingly, the profiler shows that only 9.11% of the time is spent running the user generated code (g), suggesting that over 90% of the time is spent as overhead. The most expensive function is getJSThreads, which is called once per agent per iteration. This function scans the list of scripts on the agent page and adds it to the list of scripts that an agent is running, if it doesn't exist. This repeated checking allows for hot code reloading where agents will instantaneously pick up new scripts,

but consumes an outrageous amount of time.

An alternative design that supports hot code reloading is checking once per tick for changed blocks and then updating all the agents threads if there is a block change. Because code reloading is a relatively rare event, this alternative design would significantly speed up iteration.

Other interesting hotspots are the `toString` method consuming 10% of the time and assertions consuming 1.7% of the time. Upon closer inspection, `toString` methods are invoked in logging statements even when logging is disabled because the template strings are still evaluated despite ultimately being discarded without any externally visible impact. Although an omniscient compiler could detect that these `toString` calls do not cause side-effects and elide the `toString` calls, the current compiler is not that powerful and it is necessary to manually inform the compiler that these logging statements should not be run. To keep these log statements for debugging and prevent them from slowing down the engine performance in other cases, I introduce a `make` flag that deletes the logging statements from the emitted JavaScript: if the flag is enabled, after building the JavaScript source file, it runs a series of `sed` commands to search for log-like statements and comment them out.

Using the profiler to pin down hot spots is crucial to efficiently optimizing code. With these two optimizations, the engine runs almost 3 times faster, averaging 128 milliseconds per cycle. Figure 5-8 shows a screenshot of the new profiler.

As intended, the `toString` and assertion costs have vanished. However, `getJSThreads` still consumes an incredible amount of time when all it does is return a copy of the agent's internal map of threads. To eliminate this overhead, I made the agent's internal map of javascript threads a public member and iterated over the agent's javascript threads directly.

This optimization trades off modularity for increased performance. While modularity and separation of concerns is a central tenant to building robust software systems, many programming environments such as JavaScript make it difficult to write performant modular code. We can strive for modularity by not prematurely optimizing code until we identify hotspots and grant exceptions only when there is

Figure 5-8: Profiler with 1 round of optimizations

Heavy (Bottom Up) ▼					
Self ▼		Total		Function	
20806.8 ms	32.98 %	30233.9 ms	47.92 %	▶ getJSThreads	app.is:7766
8162.1 ms	12.94 %	8162.1 ms	12.94 %	▶ set name	
6866.3 ms	10.88 %	53706.4 ms	85.12 %	▶ tick_to_js	app.is:8028
5878.1 ms	9.32 %	5878.1 ms	9.32 %	(garbage collector)	
5398.2 ms	8.56 %	8497.5 ms	13.47 %	▶ g	VM3678:1
3532.8 ms	5.60 %	12032.3 ms	19.07 %	▶ step	app.is:489
3382.7 ms	5.36 %	3382.7 ms	5.36 %	(program)	
2300.3 ms	3.65 %	2674.6 ms	4.24 %	▶ getAllAgents	app.is:1272
1507.7 ms	2.39 %	3114.2 ms	4.94 %	▶ runCollisions	app.is:8051
1240.2 ms	1.97 %	1240.2 ms	1.97 %	▶ set length	
1018.6 ms	1.61 %	2416.9 ms	3.83 %	▶ moveForward	app.is:7912
611.9 ms	0.97 %	611.9 ms	0.97 %	▶ isButtonDisabled	app.is:7954
608.0 ms	0.96 %	608.0 ms	0.96 %	▶ bounce	app.is:7883
444.9 ms	0.71 %	444.9 ms	0.71 %	▶ cos	app.is:1
384.7 ms	0.61 %	384.7 ms	0.61 %	▶ push	
345.3 ms	0.55 %	345.3 ms	0.55 %	▶ sin	app.is:1
225.8 ms	0.36 %	225.8 ms	0.36 %	▶ copy	app.is:144
133.9 ms	0.21 %	133.9 ms	0.21 %	▶ <symbol>	
127.7 ms	0.20 %	53834.1 ms	85.32 %	(anonymous function)	VM3807:1

Screenshot of the profiler after commenting out logging statements and changing the semantics of `getJSThreads` to just return the current agent's threads instead of computing all possible threads for the agent to run.

a big performance difference at stake. In this case, there appears to be a big performance opportunity. Figure 5-9 shows the results of this optimization in the profiler.

Directly accessing an agent's threads significantly speeds up engine, allowing it to run one cycle every 45.4 milliseconds. At this stage, the new WebLogo Nova engine is about three times as fast as the AS3 engine and the actual cost of executing the scripts is significant, consuming 30% of the time.

Modern Chrome versions as of the time of this thesis (v53) do not have optimized generator implementations and the actual execution of user code can be expensive. We can take advantage of the fact that few user scripts yield to create fast ordinary functions for the engine. Furthermore, for implicitly yielding top level blocks like `when-pushed` and `while-toggled`, instead of constructing generators, we can make functions that represent the inner branch and repeatedly run those functions at the thread level.

Implementing the check for whether scripts yield is complicated by recursive procedures: naively checking all procedure calls to see if they yield can result in infinite loops. To safely check all procedure calls, the checker maintains a set of procedure calls that it has begun checking and only begins checking procedures if they are not

Figure 5-9: Profiler with 2 rounds of optimizations

Heavy (Bottom Up) ▾					
Self		Total		Function	
18285.5 ms	26.70 %	61476.2 ms	89.78 %	▶ tick_to_js	app.js:8028
13013.0 ms	19.00 %	20674.7 ms	30.19 %	▶ g	VM386:1
8547.4 ms	12.48 %	29230.6 ms	42.69 %	▶ step	app.js:489
7398.1 ms	10.80 %	8435.8 ms	12.32 %	▶ getAllAgents	app.js:1272
4686.8 ms	6.84 %	9662.2 ms	14.11 %	▶ runCollisions	app.js:8051
4123.1 ms	6.02 %	4123.1 ms	6.02 %	(garbage collector)	
2692.1 ms	3.93 %	6068.8 ms	8.86 %	▶ moveForward	app.js:7912
2608.6 ms	3.81 %	2608.6 ms	3.81 %	(program)	
1519.4 ms	2.22 %	1519.4 ms	2.22 %	▶ isButtonDisabled	app.js:7954
1261.9 ms	1.84 %	1261.9 ms	1.84 %	▶ cos	app.js:1
1168.7 ms	1.71 %	1168.7 ms	1.71 %	▶ bounce	app.js:7883
1074.8 ms	1.57 %	1074.8 ms	1.57 %	▶ push	
946.0 ms	1.38 %	946.0 ms	1.38 %	▶ sin	app.js:1
575.8 ms	0.84 %	575.8 ms	0.84 %	▶ copy	app.js:144
269.4 ms	0.39 %	61745.9 ms	90.17 %	(anonymous function)	VM403:1
77.0 ms	0.11 %	96.6 ms	0.14 %	▶ getBreedIDs	app.js:1379
73.6 ms	0.11 %	73.6 ms	0.11 %	▶ sinova.ToggleButtonWidget.isToggled	toggleButtonWidget.js:55
54.6 ms	0.08 %	54.6 ms	0.08 %	▶ <symbol>	
27.0 ms	0.04 %	124.5 ms	0.18 %	▶ getBreedPairs	app.js:7464

Screenshot of the profiler after directly accessing each agent's list of threads.

in that set.

For functions that cannot yield, we can just create a single `to_js_no_yield` method that generates the equivalent JavaScript code for each StarLogo Nova block. In the case of argument blocks, this code must be a single statement so they can be easily consumed by their parents. Arguments may have side-effects and must be evaluated at most once (0 times in the case of short circuited logic). To preserve the number of times arguments are evaluated and only issue a single statement, complicated logic is refactored into helper methods.

We refrain from using helper methods to supply the implementation from every block because this increases code size and could potentially prevent many JavaScript engines from inlining and further optimizing code. Thus, we use helper methods only where strictly necessary to prevent side effects. Figure 5-10 shows an example of faster emitted code for the blocks in figure 5-5.

The performance results of creating the function fast path are shown in figure 5-11.

Replacing generators with functions wherever possible improved the performance on this benchmark by about 33%, allowing each cycle to run in 31.1 milliseconds.

Chrome currently does not have fast paths for iterators, resulting in unoptimized

Figure 5-10: Optimized JavaScript for a Transpilation Example

```
f = function(__wl_agt, __wl_scp) {
  let __wl_1_btn_name = "setup";
  if (Common.State.pushedButtons.has(__wl_1_btn_name)
    && !__wl_agt.isButtonDisabled(__wl_1_btn_name)) {
    let __wl_3_num_iters = 60000;
    for (let __wl_3_i = 0; __wl_3_i < __wl_3_num_iters; __wl_3_i++) {
      __wl_agt.moveForward(1 + 1 + 1);
    }
  }
}
```

The emitted JavaScript for the blocks in figure 5-5. Note how much terser this function is, relative to the previous generator. The code here explicitly allocates the `__wl_3_num_iters` variable to 60000 instead of inlining the 60000 because that value could be the result of a procedure call. If that value were inlined, then that procedure would be repeatedly called, potentially causing undesirable side effects.

Figure 5-11: Profiler with 3 rounds of optimizations

Heavy (Bottom Up) ▼					
Self		Total		Function	
21263.1 ms	34.51 %	56322.8 ms	91.41 %	▶ tick_to_js	app.js:8052
8671.1 ms	14.07 %	10464.9 ms	16.99 %	▶ getAllAgents	app.js:1272
6426.5 ms	10.43 %	14837.1 ms	24.08 %	▶ f	VM1073:1
5667.7 ms	9.20 %	12016.7 ms	19.50 %	▶ runCollisions	app.js:8075
3354.5 ms	5.44 %	3354.5 ms	5.44 %	(garbage collector)	
2495.0 ms	4.05 %	17494.0 ms	28.39 %	▶ step	app.js:534
1833.3 ms	2.98 %	1833.3 ms	2.98 %	▶ push	
1790.2 ms	2.91 %	1790.2 ms	2.91 %	▶ isButtonDisabled	app.js:7978
1668.0 ms	2.71 %	5471.7 ms	8.88 %	▶ moveForward	app.js:7936
1522.1 ms	2.47 %	1522.1 ms	2.47 %	▶ cos	app.js:1
1482.4 ms	2.41 %	1482.4 ms	2.41 %	(program)	
1145.8 ms	1.86 %	1145.8 ms	1.86 %	▶ sin	app.js:1
1114.7 ms	1.81 %	1131.1 ms	1.84 %	▶ bounce	app.js:7907
1020.5 ms	1.66 %	1148.8 ms	1.86 %	▶ slnova.WidgetSpace.getWidgetByName	widgetSpace.js:1
737.3 ms	1.20 %	737.3 ms	1.20 %	▶ copy	app.js:144
473.5 ms	0.77 %	473.5 ms	0.77 %	▶ next	
143.2 ms	0.23 %	143.2 ms	0.23 %	▶ push	
128.3 ms	0.21 %	128.3 ms	0.21 %	▶ [Symbol.hasInstance]	
86.8 ms	0.14 %	86.8 ms	0.14 %	▶ isButtonDisabled	

Breakdown of the most expensive functions, sorted by self time.

Figure 5-12: Profiler with 4 rounds of optimizations

Heavy (Bottom Up) ▼					
Self		Total		Function	
18522.6 ms	30.85 %	58361.5 ms	97.22 %	▶ tick_to_js	app.js:8049
17403.7 ms	28.99 %	39556.3 ms	65.89 %	▶ f	VM1953:1
5014.9 ms	8.35 %	5014.9 ms	8.35 %	▶ isButtonDisabled	app.js:7975
4300.8 ms	7.16 %	14320.6 ms	23.85 %	▶ moveForward	app.js:7933
4108.2 ms	6.84 %	4108.2 ms	6.84 %	▶ cos	app.js:1
3402.9 ms	5.67 %	3402.9 ms	5.67 %	▶ sin	app.js:1
2508.8 ms	4.18 %	2508.8 ms	4.18 %	▶ bounce	app.js:7904
2442.3 ms	4.07 %	2817.0 ms	4.69 %	▶ sinova.WidgetSpace.getWidgetByName	widgetSpace.js:1
914.2 ms	1.52 %	914.2 ms	1.52 %	(program)	
753.2 ms	1.25 %	753.2 ms	1.25 %	(garbage collector)	
374.9 ms	0.62 %	374.9 ms	0.62 %	▶ [Symbol.hasInstance]	
264.2 ms	0.44 %	39829.2 ms	66.35 %	▶ step	app.js:1
9.1 ms	0.02 %	9.1 ms	0.02 %	▶ f	VM1952:1
5.0 ms	0.01 %	5.2 ms	0.01 %	▶ sinova.WidgetSpace.getWidgetsByType	widgetSpace.js:1
4.5 ms	0.01 %	4.5 ms	0.01 %	▶ tick	app.js:4251
3.9 ms	0.01 %	58365.7 ms	97.22 %	(anonymous function)	VM1965:1
0 ms	0 %	0 ms	0 %	▶ profile	

Screenshot of the profiler after replacing many **Map** data structures with **Array** data structures for iteration. These replacements are profitable because current browsers do not have highly optimized implementations for **Map** iterations. Furthermore, optimized **Map** iteration is unlikely to beat optimized **Array** iteration because **Array**-based iteration should achieve higher data cache-hit rates.

code for `tick_to_js`. Instead of storing agent threads a map of node to the thread to allow for easy update of JavaScript threads upon recompilation, we can attain better performance by using simple arrays to store threads. When updating an agent's set of threads for a given node, we can iterate through the list of threads, find the matching node, and then make the substitution. Using the array data structure trades off code reloading time for faster access during regular execution.

`getAllAgents` consumes a large amount of the time constructing an array of all the agents. We can use a similar trick as before and make the array of all agents public and loop over it directly. Again, we trade modularity for performance.

This first benchmark does not do anything upon collision but wastes 20% of the time in `runCollisions`. At compile time, we can determine whether or not collisions need to be run and speed up cases where collisions do not need to be computed. Figure 5-12 shows the result of the smarter choice for data structures and avoidance of unnecessary work.

These better choices in data structures and avoidance of unnecessary work shave

Figure 5-13: Profiler with 5 rounds of optimizations

Heavy (Bottom Up) ▼					
Self ▼		Total		Function	
25528.2 ms	40.74 %	35971.0 ms	57.41 %	► f	VM7908:1
24924.3 ms	39.78 %	60973.8 ms	97.31 %	► tick_to_js	app.js:8008
4925.9 ms	7.86 %	4925.9 ms	7.86 %	► sin	(program):1
3071.1 ms	4.90 %	3071.1 ms	4.90 %	► cos	(program):1
2445.8 ms	3.90 %	2445.8 ms	3.90 %	► bounce	app.js:7862
890.8 ms	1.42 %	890.8 ms	1.42 %	(garbage collector)	
785.5 ms	1.25 %	785.5 ms	1.25 %	(program)	
27.7 ms	0.04 %	35.0 ms	0.06 %	► slnova.WidgetSpace.getWidgetsByType	widgetSpace.js:1
21.0 ms	0.03 %	21.0 ms	0.03 %	► f	VM7907:1
16.6 ms	0.03 %	16.6 ms	0.03 %	► tick	app.js:1
7.1 ms	0.01 %	7.1 ms	0.01 %	► push	
5.9 ms	0.01 %	35997.9 ms	57.45 %	► step	app.js:1
5.1 ms	0.01 %	60983.2 ms	97.32 %	(anonymous function)	VM7918:1
4.1 ms	0.01 %	4.1 ms	0.01 %	► <symbol>	
0.1 ms	0.00 %	0.1 ms	0.00 %	► shift	
0.1 ms	0.00 %	0.1 ms	0.00 %	► [Symbol.hasInstance]	
0 ms	0 %	10442.8 ms	16.67 %	► moveForward	app.js:1
0 ms	0 %	0 ms	0 %	► profile	

Screenshot of the profiler after eliminating unused code.

off 66% of the execution time, bringing the time per cycle down to 10.1 milliseconds.

At this stage, the overhead in executing user scripts is about 30% and we are beginning to see marginal benefits in further optimizing the overhead.

Nicholas Zakas's O'Reilly book on High Performance JavaScript[10] offers some tips on improving JavaScript loop performance but none of the proposed alternative loop syntax beat C-style for loops on modern versions of Chrome. I discuss negative results in greater detail in section 5.6.

Noting that `sin` and `cos` account for over 10% of the execution time, I tried creating lookup tables instead of calling `Math.sin` and `Math.cos`. Despite being faster in microbenchmarks, this optimization actually slowed down the overall runtime of the engine.

Surprisingly, one optimization that did work was eliminating unused data structures. This elimination might have tripped a critical threshold in module size, allowing for more aggressive inlining. Figure 5-13 shows the heaviest functions after eliminating unused data structures.

These changes squeezed out 30% better performance, reducing the runtime per iteration to 7.02 milliseconds.

About 40% of the time is spent in looping over the agents and each of their

threads. This figure seems high and upon further inspection, Chrome does a poorer job of optimizing `let` compared to `var`. Both `let` and `var` allocate variables, but variables allocated with `let` are scoped only to the block, whereas variables allocated with `var` are scoped to the entire function. `let` offers better modularity and scoping but trades off performance at the moment. For very tight loops, I replace instances of `let` with `var` and observe a performance improvement (figure 5-14). However, I keep using `let` elsewhere in the codebase because the performance cost is insignificant and the block-based scoping can prevent scoping bugs.

There is a third way to declare variables in JavaScript by using the `const` keyword. This tells the JavaScript engine that the declared variable is constant, which allows it to optimize the generated machine code to leverage this information. Although omniscient JavaScript engines could prove whether a variable is constant, the current generation of JavaScript engines gain a modest speed boost from this hint so I use `const` where possible.


It is interesting to note here that we literally trade off modularity (block-scoped `let`) for performance (function-scoped `var`). Based on what we have observed so far, we seem to have to fundamentally trade off modularity for performance. Looking at this observation from another perspective, we can purchase modularity and clean code by paying with performance.

The engine now runs each cycle in 6.01 milliseconds, over 23 times faster than the original AS3 engine for this benchmark. With CPU frequency scaling enabled, the engine can execute 60 cycles in under a quarter of a second on a fairly slow hardware platform so many users can expect smooth performance.

To recap how I was able to speed up the execution engine by a factor of 23, I

1. cached the output of `getJSThread` so the engine does not have to compute what threads an agent should run each iteration. I also stripped out logging statements. It is insufficient to just prevent the logging statement from calling `console.log` since the string format will still get serialized. This optimization leads to a **183% speed up**.

Figure 5-14: Profiler with 6 rounds of optimizations

Heavy (Bottom Up) ▾ 🔍 ✕ ↺					
Self ▾		Total		Function	
28452.1 ms	39.45 %	70048.8 ms	97.13 %	▶ tick_to_js	app.js:8177
23247.1 ms	32.24 %	41442.7 ms	57.47 %	▶ f	VM8702:1
7271.1 ms	10.08 %	7271.1 ms	10.08 %	▶ cos	(program):1
6693.8 ms	9.28 %	6693.8 ms	9.28 %	▶ sin	(program):1
4230.7 ms	5.87 %	4230.7 ms	5.87 %	▶ bounce	app.js:8045
1067.9 ms	1.48 %	1067.9 ms	1.48 %	(program)	
987.9 ms	1.37 %	987.9 ms	1.37 %	(garbage collector)	
54.7 ms	0.08 %	54.7 ms	0.08 %	▶ clearDead	app.js:1261
39.3 ms	0.05 %	48.7 ms	0.07 %	▶ sinova.WidgetSpace.getWidgetsByType	widgetSpace.js:1
22.9 ms	0.03 %	22.9 ms	0.03 %	▶ tick	app.js:1
21.2 ms	0.03 %	21.2 ms	0.03 %	▶  f	VM8701:1
9.6 ms	0.01 %	9.6 ms	0.01 %	▶ push	
7.7 ms	0.01 %	70059.4 ms	97.15 %	(anonymous function)	VM8716:1
6.6 ms	0.01 %	41470.5 ms	57.51 %	▶ step	app.js:1
2.3 ms	0.00 %	2.3 ms	0.00 %	▶ <symbol>	
0.1 ms	0.00 %	0.1 ms	0.00 %	▶ shift	
0.1 ms	0.00 %	0.1 ms	0.00 %	▶ [Symbol.hasInstance]	
0 ms	0 %	18195.6 ms	25.23 %	▶ moveForward	app.js:1
0 ms	0 %	0 ms	0 %	▶ profile	

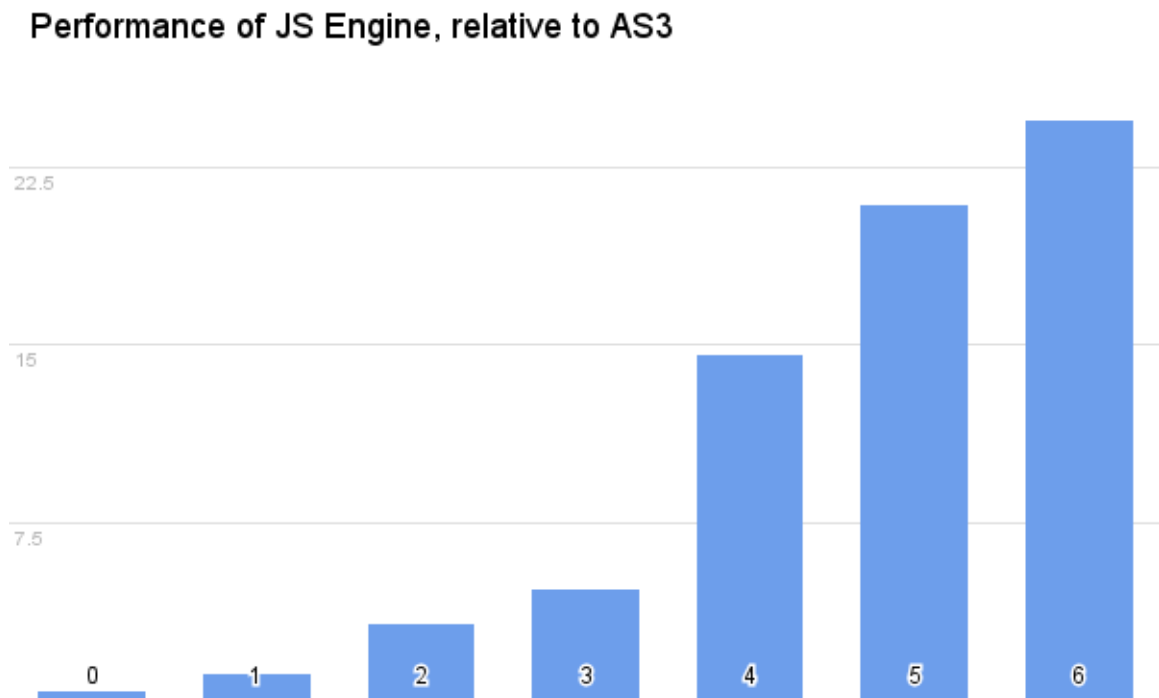
Screenshot of the profiler after replacing `let` with `var` in tight loops.

2. directly accessed an agent's threads instead of referencing a copy. This optimization leads to a **182% speed up**.
3. used functions instead of generators wherever possible. This optimization leads to a **46% speedup**.
4. replaced map iteration with array iteration, directly iterated over the array of agents, and computed collisions only when necessary. This optimization leads to a **208% speed up**.
5. deleted unused data structures. This optimization leads to a **44% speed up**.
6. used `var` and `const` instead of `let`. This optimization leads to a **17% speed up**.

Figure 5-15 summarizes the cumulative effects of these optimizations.

At this stage, the renderer is the bottleneck, consuming over 90% of the cpu during the simulation.

Figure 5-15: Cumulative Impact of General Optimizations



Summary of all the changes and their performance relative to the AS3 engine. Stage 0: unoptimized engine. Stage 1: Cached the value of `getJSThreads` and remove logging statements. Stage 2: Directly access threads. Stage 3: Use functions wherever possible. Stage 4: Replace map iteration with array iteration, directly access agents list, compute collisions only when necessary. Stage 5: Delete unused data structures. Stage 6: Use `var/const` instead of `let`.

Figure 5-16: Unoptimized Renderer Profile

Heavy (Bottom Up) ▾ 🔍 ✕ ↺					
Self ▾		Total		Function	
27835.3 ms	36.85 %	27835.3 ms	36.85 %	(program)	
13131.0 ms	17.39 %	13131.0 ms	17.39 %	(garbage collector)	
7610.0 ms	10.08 %	7620.1 ms	10.09 %	▶ render	ModelManager.js:67
5897.9 ms	7.81 %	11272.3 ms	14.92 %	▶ ⚠ add	ModelManager.js:42
4685.9 ms	6.20 %	4685.9 ms	6.20 %	▶ push	
2544.9 ms	3.37 %	34026.3 ms	45.05 %	▶ ⚠ render	Viewport.js:116
2466.6 ms	3.27 %	3622.0 ms	4.80 %	▶ f	VM17446:1
2444.6 ms	3.24 %	6075.8 ms	8.04 %	▶ tick_to_js	app.js:8014
2197.8 ms	2.91 %	5097.8 ms	6.75 %	▶ ⚠ getAllAgentsToRender	app.js:1319
1790.4 ms	2.37 %	2096.3 ms	2.78 %	▶ ⚠ toRenderAgent	app.js:7910
1030.2 ms	1.36 %	1030.2 ms	1.36 %	▶ length	three.min.js:33
440.6 ms	0.58 %	440.6 ms	0.58 %	▶ sin	OrbitControls.js:1
410.8 ms		410.8 ms		(idle)	
392.0 ms	0.52 %	392.0 ms	0.52 %	▶ cos	OrbitControls.js:1
364.3 ms	0.48 %	364.3 ms	0.48 %	▶ getAllAgents	app.js:1293
323.6 ms	0.43 %	323.6 ms	0.43 %	▶ bounce	app.js:7868
316.7 ms	0.42 %	316.7 ms	0.42 %	▶ bufferSubData	
303.9 ms	0.40 %	303.9 ms	0.40 %	▶ ⚠ RenderAgent	app.js:188
154.1 ms	0.20 %	154.1 ms	0.20 %	▶ _w	

The `add` and `getAllAgentsToRender` total times account for the time (21.67%) it takes to transfer data from the engine to the renderer, which is more than double of the actual engine execution time (8.04%).

5.5.3 Renderer Optimizations

The current renderer uses the `THREE.js` JavaScript library to draw agents on a `html` canvas. While work to speed up the actual rendering time is out of the scope of this thesis, we can optimize how data is transferred between the engine and the renderer. Currently for each tick, the renderer requests a list of render objects from the engine. For each object, it will append state data (such as the object's `x`, `y`, `z` coordinates, `rotation`, `color`, `size`) to per-shape arrays. This inefficient transfer accounts for over 20% of the time per frame and more than double the time for the actual engine to run. Figure 5-16 shows a profiler trace for the full engine-renderer pipeline.

I optimize the renderer to directly store a reference to an array of agent states so the engine does not have to construct an array with all the agent states each iteration. To construct the per-shape arrays, it first allocates the full length of all the arrays (instead of incrementally appending to the arrays). Then it iterates through the agent states list reference it has to populate the per-shape arrays.

Figure 5-17: Optimized Renderer Profile

Heavy (Bottom Up) ▾ 🔍 ✕ ↺					
Self ▾		Total		Function	
52139.0 ms	54.50 %	52139.0 ms	54.50 %	(program)	
17296.9 ms	18.08 %	17309.8 ms	18.09 %	▶ render	ModelManager.js:122
15772.2 ms	16.49 %	15773.6 ms	16.49 %	▶ addAll	ModelManager.js:65
2824.4 ms	2.95 %	4266.5 ms	4.46 %	▶ f	VM22538:1
2550.7 ms	2.67 %	6832.2 ms	7.14 %	▶ tick_to_js	app.js:8021
1586.7 ms	1.66 %	1586.7 ms	1.66 %	(garbage collector)	
904.3 ms		904.3 ms		(idle)	
539.2 ms	0.56 %	539.2 ms	0.56 %	▶ sin	OrbitControls.js:1
519.4 ms	0.54 %	519.4 ms	0.54 %	▶ cos	(program):1
404.0 ms	0.42 %	404.0 ms	0.42 %	▶ bufferSubData	
384.2 ms	0.40 %	384.2 ms	0.40 %	▶ bounce	app.js:7875
206.1 ms	0.22 %	302.7 ms	0.32 %	▶ end	stats.min.js:4
93.8 ms	0.10 %	143.7 ms	0.15 %	▶ v	three.min.js:540
67.2 ms	0.07 %	67.2 ms	0.07 %	▶ bindBuffer	
64.8 ms	0.07 %	325.5 ms	0.34 %	▶ renderBufferDirect	three.min.js:589
56.0 ms	0.06 %	1105.9 ms	1.16 %	▶ render	three.min.js:595
55.3 ms	0.06 %	55.3 ms	0.06 %	▶ requestAnimationFrame	
50.8 ms	0.05 %	50.8 ms	0.05 %	▶ requireNative	
49.8 ms	0.05 %	41075.5 ms	42.94 %	▶ render	Viewport.js:122
47.8 ms	0.05 %	47.8 ms	0.05 %	▶ appendChild	
47.4 ms	0.05 %	71.0 ms	0.07 %	domLayoutChanged	contentscript-end.js:874

The bulk addition of all the agents reduces the time to load all the states to 16.49% and significant reduces the amount of garbage collection time by nearly 90%.

This optimization has the potential to increase the coupling between the renderer and engine in exchange for higher performance. To reduce the the coupling, the renderer is given the agent state array reference during the setup process so it does not require explicit knowledge of the engine. This optimization speeds up the frame rate by 88% and figure 5-17 shows the results of this optimization.

5.5.4 Collision Optimizations

StarLogo Nova needs to calculate collisions when there are user scripts attached to collision events. Internally, StarLogo uses a binning system to compute collisions: the space is divided up in a uniform grid. Agents are inserted into any bins that they may land in and to compute collisions and each agent is checked against every other agent in each of the bins that the original agent is in. This binning system is also used to support efficient lookups of nearby agents and counting the number of nearby

Figure 5-18: Unoptimized Collision Profile

Heavy (Bottom Up) ▼ 🔍 ✕ ↺					
Self		Total		Function	
86493.3 ms	38.91 %	86718.6 ms	39.01 %	▶ getCollisions	app.js:7354
40593.1 ms	18.26 %	75544.0 ms	33.98 %	▶ f	VM416:1
11974.5 ms	5.39 %	13348.9 ms	6.00 %	▶ detectCount	app.js:1027
8960.9 ms	4.03 %	8960.9 ms	4.03 %	(program)	
8723.0 ms	3.92 %	8723.0 ms	3.92 %	(garbage collector)	
7592.8 ms	3.42 %	7592.8 ms	3.42 %	▶ sin	(program):1
7053.7 ms	3.17 %	9726.3 ms	4.38 %	▶ remove	app.js:865
6656.0 ms	2.99 %	25397.7 ms	11.42 %	▶ f	VM388:1
5712.2 ms	2.57 %	13463.0 ms	6.06 %	▶ add	app.js:849
5186.6 ms	2.33 %	204606.7 ms	92.04 %	▶ tick_to.js	app.js:7982
4623.6 ms	2.08 %	22157.0 ms	9.97 %	▶ count	app.js:1309
3919.2 ms	1.76 %	3919.5 ms	1.76 %	▶ check	app.js:1312
3673.2 ms	1.65 %	4291.8 ms	1.93 %	▶ getAllBins	app.js:1108
3459.1 ms	1.56 %	3459.1 ms	1.56 %	▶ insert	app.js:175
2672.6 ms	1.20 %	2672.6 ms	1.20 %	▶ remove	app.js:179
2077.4 ms	0.93 %	5092.4 ms	2.29 %	▶ f	VM384:1
1639.5 ms	0.74 %	1639.5 ms	0.74 %	▶ getAllAgents	app.js:1283
1383.8 ms	0.62 %	88126.8 ms	39.64 %	▶ getCollisions	app.js:7600
1316.9 ms	0.59 %	89952.0 ms	40.46 %	▶ runCollisions	app.js:8025
1030.8 ms	0.46 %	1030.8 ms	0.46 %	▶ goog.ui.Component.getChildCount	component.js:1

As expected, the call to `getCollisions` consumes a plurality of the computation time.

agents.

I optimize collisions against the Something's Fishy benchmark which spawns 48 fish agents and hundreds of algae agents. Fish try to swim towards algae and eat the algae upon collision, killing the algae and boosting the fish's internal energy. If the fish has sufficient energy, it will expend a lot of that energy to reproduce asexually. Algae also reproduce if the algae population is not too crowded locally.

Compared to the other optimized parts of the engine, the unoptimized collision engine is expected to be the slowest part. Figure 5-18 is a profiler screenshot confirming this hypothesis.

The engine takes 128ms per cycle, compared to 34.7⁹ms for AS3. We can do better by applying some of the profitable micro-optimizations from before, such as using `var` instead of `let` in critical loops, passing references instead of creating copies, and using flat arrays instead of maps and sets for iteration.

During the migration process, I found a bug in the previous collision detection code where it would incorrectly prevent two agents from colliding in an edge case. I

⁹Because I was unable to set a deterministic seed for AS3's random number generator and this test heavily relies on nondeterminism, this figure is the best of 5 runs, with each run spanning at least a minute of engine time. The individual times (in sorted order) were [34.7, 40.8, 46.1, 48.5, and 51.8] milliseconds with a standard deviation of 6.7ms.

Figure 5-19: Optimized Collision Profile

Heavy (Bottom Up) ▾ 🔍 ✕ ↺					
Self ▾		Total		Function	
12016.8 ms	15.44 %	12016.8 ms	15.44 %	▶ sin	(program):1
9151.4 ms	11.76 %	70380.4 ms	90.41 %	▶ tick	app.js:8118
7060.3 ms	9.07 %	32297.6 ms	41.49 %	▶ f	VM17204:1
5379.1 ms	6.91 %	13950.0 ms	17.92 %	▶ f	VM17176:1
4149.0 ms	5.33 %	4149.0 ms	5.33 %	(garbage collector)	
3622.7 ms	4.65 %	5209.0 ms	6.69 %	▶ add	app.js:3706
3034.3 ms	3.90 %	3034.3 ms	3.90 %	▶ remove	app.js:3721
3013.4 ms	3.87 %	3013.4 ms	3.87 %	(program)	
2868.8 ms	3.69 %	13157.5 ms	16.90 %	▶ f	VM17202:1
2166.1 ms	2.78 %	2536.3 ms	3.26 %	▶ getCollisions	app.js:3736
1765.8 ms	2.27 %	2935.3 ms	3.77 %	▶ getTrait	app.js:7956
1710.4 ms	2.20 %	5348.4 ms	6.87 %	▶ f	VM17172:1
1662.5 ms	2.14 %	1800.0 ms	2.31 %	▶ count	app.js:3770
1643.9 ms	2.11 %	1643.9 ms	2.11 %	▶ safeSetHeading	app.js:7940
1483.9 ms	1.91 %	1483.9 ms	1.91 %	▶ getTraitID	app.js:1
1372.3 ms	1.76 %	1372.3 ms	1.76 %	▶ ceil	
972.8 ms	1.25 %	1883.8 ms	2.42 %	▶ Agent	app.js:7828
944.0 ms	1.21 %	944.0 ms	1.21 %	▶ f	VM17203:1
901.5 ms	1.16 %	1032.0 ms	1.33 %	▶ slnova.WidgetSpace.getWidgetByName	widgetSpace.js:259
844.3 ms	1.08 %	844.3 ms	1.08 %	▶ cos	app.js:1
831.1 ms	1.07 %	1101.2 ms	1.41 %	▶ equals	app.js:230

After optimization, the call to `getCollisions` is no longer a bottleneck.

inefficiently patched it by constructing collision interest maps at runtime to determine whether two agents had scripts to run if they collided: if no scripts need to be run if they collide, then we can skip the check for if the agents collide. However, this collision interest map can be constructed at compile time since all the colliding breeds are known.

In the old implementation, all breeds are stored in the same bins, but this makes collision detection inefficient since agents that do not collide are looped over. I introduce finer grained collision by binning each agent breed separately.

In the previous implementation, bins were updated by taking each agent, removing it from all the bins that it was in, and reinserting it into the bins. I add a small check to only update bins if the agent has moved or resized since the last cycle. Figure 5-19 shows the results of these optimizations.

These optimizations enable the engine to run each cycle in 9.03ms, which is about 4 times faster than AS3 and 14 times faster than before. It is important to note that this isn't the fairest comparison since the nondeterminism in the simulation makes it difficult to directly compare figures.

As with the 10000 fish case, the renderer time dominates the engine time, diminishing the benefit of further optimizations.

5.6 Negative Optimization Results

The previous sections highlight successful approaches to optimizations and serve to be useful to practitioners seeking higher performance in their web applications. However, it is also useful to practitioners to describe failed optimization attempts so they know what may not work well.

Figure 5-14 shows that a significant amount of time was spent calling `Math.sin` and `Math.cos`. I attempted to reduce this cost by creating lookup tables so values for sine and cosine can be simply looked up. While lookup tables seemed promising based on microbenchmarks that computed sine and cosine in tight loops, I found that the use of lookup tables actually slowed down the benchmark. This somewhat contradictory behavior can be explained how browsers will inline functions: Chrome will only inline functions if both the called function is small and the calling function is not too large. In the case of the microbenchmark, the calling function is small so the table could have easily been inlined. However, in the case of the actual StarLogo Nova engine, the calling function could have been too large to inline into, resulting in greater overhead than the native sine and cosine calls. It is important to note that JavaScript engines cannot squeeze optimizations out of inlining everything because that could make a hot function too large to fit into the L1 instruction cache, increasing the rate of cache misses, L2 fetches, and pipeline stalls, ultimately slowing down execution.

This optimization that did not work raises an interesting point about performance engineering. It would be nice to be able to think of every performance tweak as additive and independent of the other changes. To make a program fast, all we have to do is apply several additive changes. However, there is a large amount of coupling as some changes can increase or decrease program sizes, changing the outcomes of the various optimization heuristics browsers apply. Thus, it is very important to continuously monitor the engine performance with each additional feature to track

performance regressions and fix them immediately.

Chrome 53 introduced a profiler feature to track how much time was spent on each line (figure 5-20), similar to tools like `gcov` for C. However, the numbers reported by this feature seem wildly inaccurate, possibly due to bugs mapping execution time of the compiled code back to the source. It would be interesting to revisit this feature in future versions of Chrome and use it to further guide optimizations.

Nicholas Zakas published an O'Reilly book on High Performance JavaScript (HPJ) in 2010[10]. I tried some of the applicable advice in Chapter 4: Algorithms and Flow Control, but did not find that the suggestions improved overall performance for StarLogo Nova. This is likely a reflection of the evolving JavaScript engine landscape.

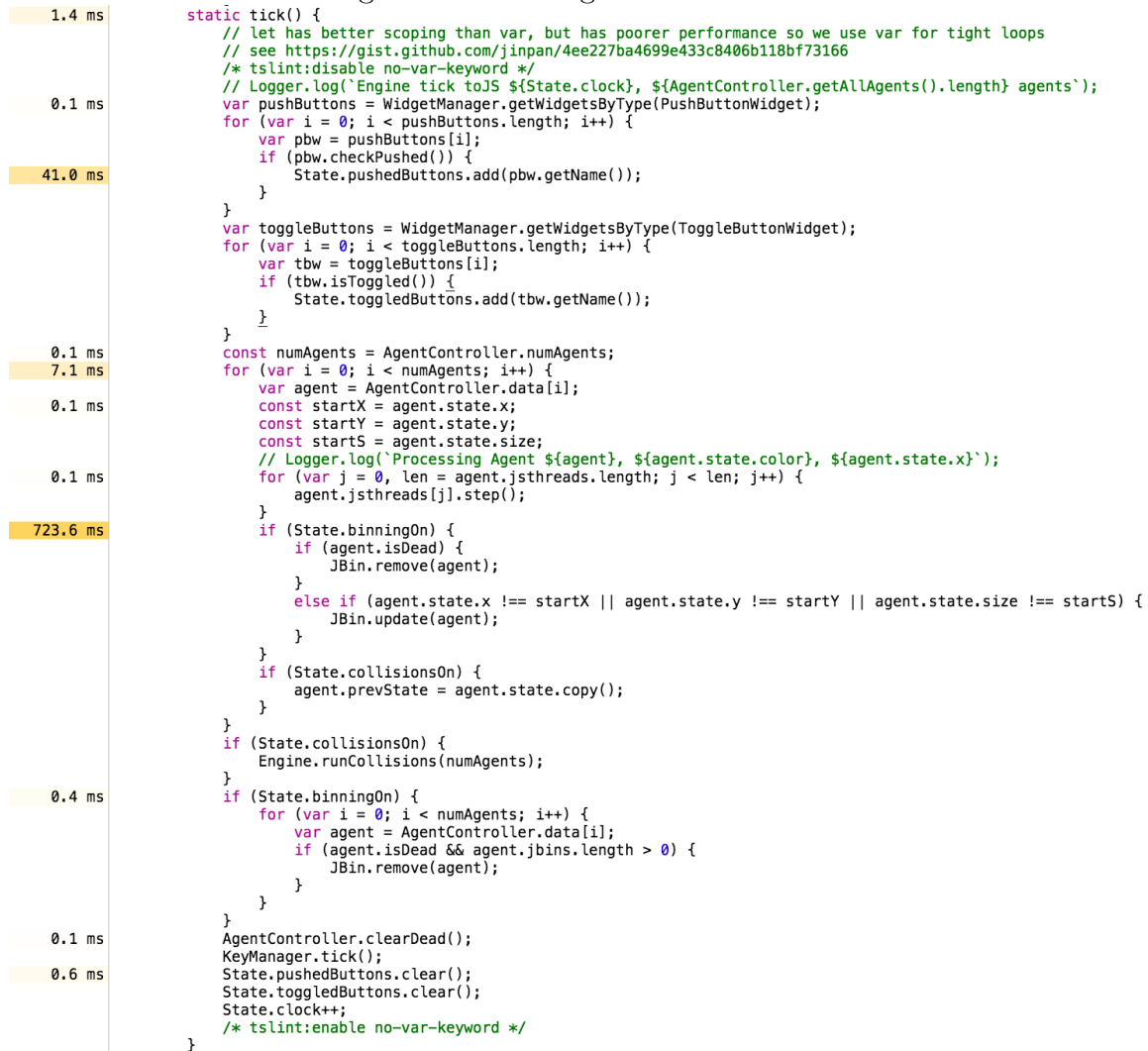
One suggestion HPJ makes is to cache array lengths in loops: instead of looping with `for (var i=0; i<arr.length; i++) ...`, HPJ suggests doing `for (var i=0, len=arr.length; i<len; i++)` The two approaches were basically indistinguishable performance-wise for large arrays (> 1000 elements) and the first approach was actually slightly faster for small arrays (10 elements). Caching array lengths did not significantly impact the actual performance of the engine, possibly because the JavaScript engine already caches array lengths behind the scene. These measurements contradict the HPJ claims that one can save 25-50% off the total loop execution time by caching array lengths.

Another suggestion HPJ makes is to try looping over arrays backwards. Instead of doing `for (var i=0; i<arr.length; i++) ...`, looping backwards would look like `for (var i=arr.length; i--;) ...`¹⁰. I found that looping backwards actually has a significant detrimental impact, taking about 20% longer than looping forwards. These measurements also contradict HPJ claims that one can save an additional 25% off of the total loop execution time by looping backwards.

One promising suggestion that HPJ makes is to use Duff's device (figure 5-21) to unroll large loops. Unrolling loops decreases the amount of overhead spent keeping

¹⁰The case for looping backwards is that we can shave off one statement by combining the array index update with the ending condition check. However, backwards loops are more uncommon than forward loops so it is likely that modern browsers have significantly more optimized paths for forward loops.

Figure 5-20: Strange Profiler Results



Profiler breakdown of how much time was spent on each line. The reported figures are highly suspect because more than 723 milliseconds (nearly 40% of the runtime) are reported to be spent checking the `State.binningOn` boolean, but an inconsequential amount of time was spent running through the `for` loop of each agent's threads.

Figure 5-21: Duff's Device

```
if (arr.length > 0) {
    var iterations = (arr.length + 7) >> 3;
    var startAt = arr.length & 0x7;
    var i = 0;

    do {
        switch (startAt) {
            case 0: foo(arr[i++]);
            case 7: foo(arr[i++]);
            case 6: foo(arr[i++]);
            case 5: foo(arr[i++]);
            case 4: foo(arr[i++]);
            case 3: foo(arr[i++]);
            case 2: foo(arr[i++]);
            case 1: foo(arr[i++]);
        }
        startAt = 0;
    } while (--iterations);
}
```

An example of Duff's device unrolling a loop 8 times. The number of iterations is $\frac{1}{8}$ th of the original array size, significantly reducing the amount of checks and increments of the loop counter (iterations). Since `startAt` is 0 for every iteration of the loop past the first loop, branch prediction should be very accurate and the cost of the switch statements should be minimal.

track of the number of loops to run. Preliminary microbenchmarks show that Duff's device does improve loop performance over 40% for loops with 10000 iterations. However, when we replace the main loop over all the agents with Duff's device, we actually observe a minor performance regression. I suspect that this performance regression is due to the increase in code size and prevention of other aggressive inlining tricks that Chrome applies.

To its credit, HPJ pointed out that object iteration and function-based iteration were slower than their direct array iteration counter parts. We were able to corroborate these claims, but since we do minimal object and function iteration, this information does not help us speed up the execution engine.

Table 5.2: Overall Benchmark Results Table

Benchmark	AS3 time	JavaScript time
10000 Fish	147 ms	6.0 ms
Amylase	29 ms	7.7 ms
Bacteria	128 ms	10.6 ms
Yielding Dragon Fractal	40 ms	6.2 ms
Non-Yielding Dragon Fractal	209 ms	13.6 ms
Paintball	3.8 ms	0.43 ms
Something’s Fishy	35 ms	9.0 ms

I also tried to unroll the loop over each agent’s threads by creating a switch statement on the length of the thread array. If the thread array were greater than some length k , then it would enter the default case where it would fallback to a loop. Since most agents have the same number of threads, the branch prediction should be incredibly accurate, but this optimization also slowed down the execution engine, presumably due to the larger code footprint preventing inlining.

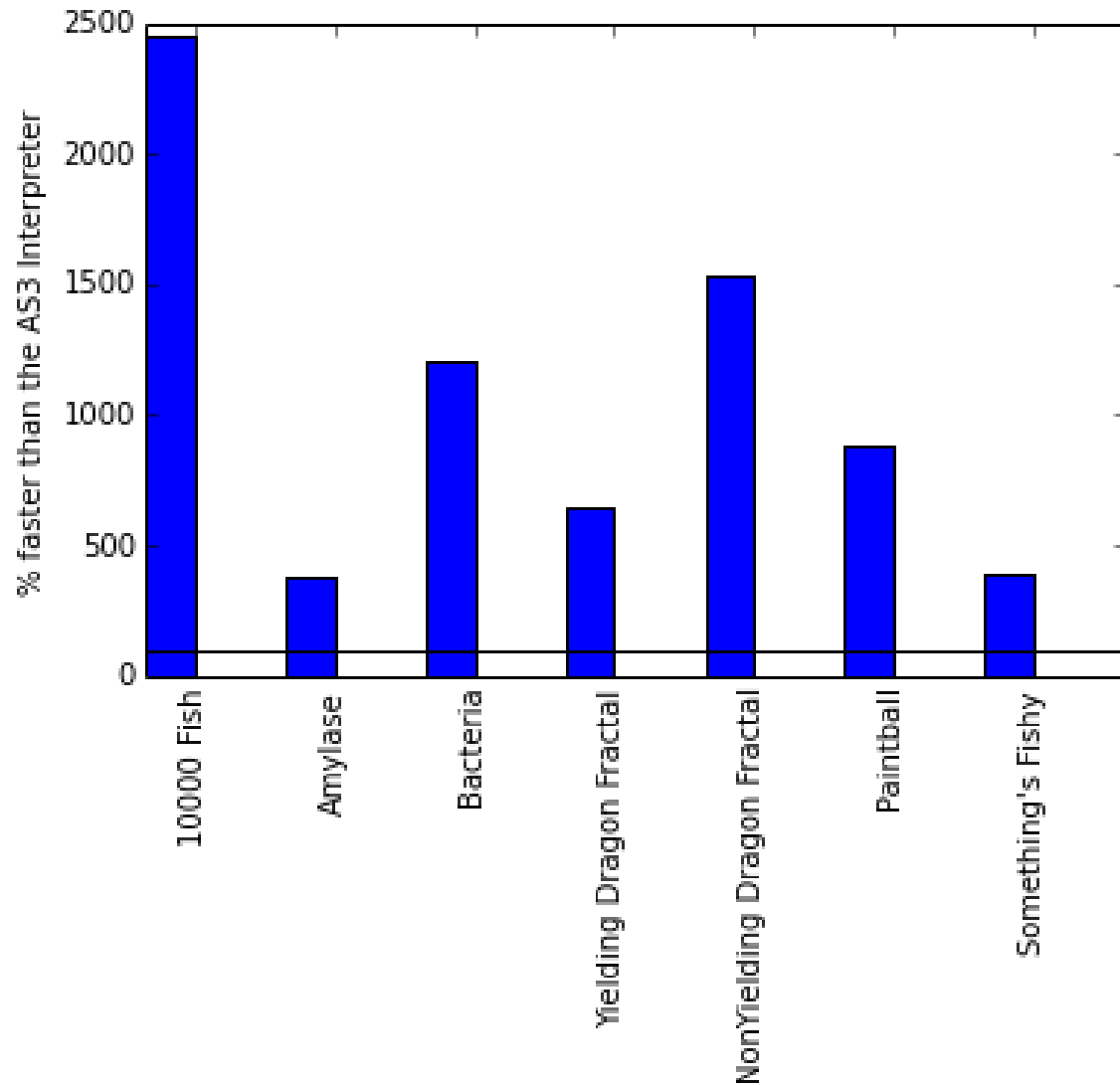
Many of these unsuccessful attempts bring up another general idea in performance engineering: microbenchmarks are useful for validating assumptions about how quickly certain code constructs run, but they are insufficient for proving that something will be faster or slower in the context of the entire program. However, they are a useful starting step because they can be a low-cost way to demonstrate optimizations that are unlikely to make a difference (for example, `i++` vs `++i` at the end of loop and declaring a variable inside a loop vs outside a loop do not make a performance difference).

5.7 Overall Benchmark Results

Overall, we are able to bring the average execution time per cycle under 10ms for most benchmarks.

Figure 5.2 shows figures across the entire benchmark suite.

Figure 5-22: Overall Benchmark Results Graph



The new execution engine is 54% to 2350% faster than the previous engine across this benchmark suite. The black horizontal line represents the baseline performance of the AS3 interpreter across all the benchmarks.

Chapter 6

Future contributions

6.1 Better Type Handling

6.1.1 Current Type System

One area that StarLogo Nova suffers is type safety. As first described in section 1.6, StarLogo Nova lumps together many data types such as numbers, strings, colors, shapes, agents, and nulls into a generic data type. This lack of type safety can allow users to assemble incorrect programs. For example, a user could use the `trait-of` block to look up a trait on a number, which does not make sense: trait lookups are only defined for agents. Since the agents are dynamic, we cannot use a drop down to restrict the `trait-of` block (and other similar blocks) to operate solely on agents.

The StarLogo Nova engine's policy towards incorrect programs is that the show must go on. `trait-of` will return undefined in the case where the trait lookup fails (either the lookup is performed on a non-agent or the requested trait does not exist on the agent).

Although this policy makes all programs "run", it can make agents run in unexpected and confusing ways. A stronger type system would help users create correct programs.

6.1.2 Stronger Typing Challenges and Solutions

One problem with a stronger type system is that it may require many more blocks. Instead of a single addition block that is overloaded both as a numeric addition operator and a string concatenation operator, there would be a separate block for each type. This undesirably increases the size of the blocks drawer and can confuse users: before, they could use an overloaded addition block for both actions and now they must use two different blocks.

An alternative design is to create the visual analog of generic blocks that accept arguments only if their types match. For example, a generic block would initially accept arguments of any type. However, all further arguments would have to match the first accepted argument's type. This way, StarLogo Nova could create data types for strings and numbers without having to double the number of blocks.

This generic block requires type knowledge and ScriptBlocks can supply that information for a limited number of cases. For example, the `count` instruction must return a number, the `collidee` instruction must return an agent, and an addition of two strings must return a string. However, there is not enough type information on every block to determine the type of their inputs. For example, the generic data sockets allow users to type in string or number literals, but it is difficult to distinguish between strings and numbers (Does 7 mean "7" or the numeric 7? Does 0xDEADBEEF mean the string literal "0xDEADBEEF" or the decimal value 3735928559?) Furthermore, scripts are allowed to write values to widgets and read values back, but ScriptBlocks cannot guarantee that if all written values are of a certain type, then all read values must of the same type: a user is allowed to manually mutate those values between write and read stages so if a script wrote a number, it can get a string back.

Because it is impossible to ascertain type information automatically, stronger typing requires the user to supply type information about blocks that ScriptBlocks cannot determine. However, requiring the user to supply a type for every input box can be tedious and verbose, which is a common argument against strongly typed languages.

StarLogo Nova can instead operate as a weakly typed language, where it will allow users to declare types but not require those type declarations. It will infer types as much as it can, preventing users from setting the shape of an agent to a color, but it will also retain the generic data type for cases where the type of arguments is ambiguous.

When users suspect that they have bugs in their simulation arising from mismatched types, they can then declare types in ScriptBlocks and use the static type checker to help them catch type errors.

6.2 Live code editing

One limitation of StarLogo Nova is that users must reset the state of their simulation each time they update their code. For example, if a user were creating a path-finding algorithm to solve a maze and wanted to see the effects to tweaking the search step size, the user must reset their simulation, click on the setup buttons, and wait for the simulation to run to an interesting state where the effects of their search step size can be observed. This tedious workflow can take long periods of time and the manual stages can be error prone.

One feature that would speed up this workflow is the support of live code editing: users can update their blocks and immediately see agents execute this new behavior. This section outlines some of the work done to support live code editing and proposes semantics and user-interface for live code editing.

6.2.1 Live code editing groundwork

I started laying some of the groundwork for live code editing from the engine's point of view.

Under the previous assumption that everything would be reset between compiles of the code, it is acceptable to directly refer to all renamable objects, such as breeds, traits, and procedure calls by their name since the names are immutable until a global reset. However, this assumption is no longer valid if we want to allow seamless code

editing without resetting the entire state.

It can be easy to forget to rename a particular data structure that stores the name of a renameable object, resulting in data corruption. To eliminate the possibility of this kind of bug, I restructure how these renameable objects can be accessed: each renameable object is assigned an immutable numeric ID and all users of these objects have a reference to this immutable ID. For cases where we want to look up the object from its name, all lookups go through a single `getObjectByName` method. By forcing this pattern of going through a `getObjectByName` method, we can update the name in a single place and be confident that all consumers of this object will be able to detect the change.

6.2.2 Live code editing semantics

There are two main approach to support live editing. One way is akin to time travel, where we go back in time to the start of the simulation and rerun all the cycles with the updated code, replaying all user interactions such as widget button presses and keyboard events. The other approach is to leave the past untouched and only run the new code from now on[6].

Although the first approach is much more powerful and guarantees that any state reached by the simulation after editing is a reproducible state, it is also considerably more difficult to implement and performance will be inferior to leaving the past untouched. If a simulation has used a minute of engine time, then this time travel approach will use a significant amount of time to recalculate the present¹, which fails to address one of the fundamental reasons why we want to support live editing.

We recommend pursuing the second approach of leaving the past untouched and just running the new code. One downside of this "preserve-the-past" approach is that the simulation could enter a state that would be impossible to enter if the simulation had started from the beginning with either the old code or the new code, making some results irreproducible. However, we anticipate that the user will be interested

¹If we skipped the renderer steps to speed up the recomputation, this would be jarring and hang the simulation for a significant amount of time.

only in the high level details of the simulation and those details should be resilient to this reproducibility problem.

The majority of code can be swapped out easily on reload: for threads that do not explicitly yield, we can easily swap out the function for these threads at the end of the cycle.

The difficult case is for when these blocks do yield. Since the problem of determining how many iterations it takes these yielding threads to stop is undecidable², we have to build heuristics for when to kill these yielding threads and restart them with the new code. One heuristic is to run the interface thread for up to `n` iterations, where `n` is a small number (`<60`). If at any cycle the interface thread execution returns to the top, then we kill the old thread and construct a new one. If at the end of these iterations, the thread execution never returns to the top, then we kill the old thread and construct a new one.

These heuristics reduce the likelihood of corrupting agent state. For example, if an agent had a block stack that implemented jumping by checking for `key-pressed`, and yielded between vertical movements if the given key were pressed, then agent state would be safe if `n` were greater than the number of yield statements. A more naive implementation that immediately halted old threads could leave the agent stuck in midair.

It is possible to deliberately construct programs that throw this heuristic off, but we suspect that those programs will not be a common occurrence (since the yield block is used highly infrequently). If a user does run into an issue with the semantics of live code reloading, they can always recover by executing a hard reset and running their code normally.

6.2.3 Live code editing user interface

Although it is desirable to quickly update agent behavior to match user code, it is undesirable to update the agent behavior upon every small change to the user code.

²If it were decidable, then one could construct a program that runs the checker on itself and yield one more time than the checker thinks it would, leading to a contradiction.

If a user were in the middle of updating a parameter from 10 to 11, the user would not intend for the simulation to read a parameter value of 1. Another reason why it is infeasible to update agents each time is because the scripts could be in an intermediate state that cannot run normally due to missing arguments. Furthermore, the cost of recompiling code upon every user update could lead to undesirable input lag.

An alternative interface would be to show that the script has deviated from the currently loaded script and that a recompile is necessary to update the agents' behavior. Upon a change, ScriptBlocks would run an internal consistency check to verify that all required sockets are filled. If this check succeeds, ScriptBlocks could present a button for the user to press to load the new code. Upon loading the code, the Execution Engine would recompile the code and swap it in as discussed above. It could then fire a signal to the user interface to let the user know that the code has been reloaded.

If the consistency check is computationally expensive to run, ScriptBlocks could defer execution of this consistency check until some number of milliseconds after the last user action, preventing the checker from stopping the world and making the user interface unresponsive.

Performance-wise, compilations to JavaScript are fast. However, there will be an ephemeral performance penalty because the underlying JavaScript engine needs to optimize this new code. We do not anticipate this penalty to be significantly expensive or noticeable to the user.

6.3 Debugging

Even with the friendly blocks-based editor, users can make mistakes with their programs and need to figure out why their agent behavior doesn't match their intentions. As previously mentioned in section 6.1, a stronger type system for ScriptBlocks could help users quickly fix type-related problems. However, not all problems can be fixed by type checking.

There are two primary ways of debugging programs: logging and stepping. Log-

ging is the high throughput approach, where programmers add print statements to critical sections and watch the output of their program for these print statements. By observing enough logged data, programmers can get a sense for what sections of code are reached and a general idea of internal state. Logging is especially useful for when there is a large volume of code being executed and they need to figure out what section the error is in.

Stepping is a much more precise approach than logging, but can be much slower. Programmers can step through code line by line with the aid of a debugger and inspect the state of any variable. They can also step into called functions to get a sense for what is happening underneath the surface and even alter variables to force the system into a certain state. Although stepping is very powerful, it is a slow process that requires the programmer to know exactly what needs to be stepped.

To help users, StarLogo Nova should implement functionality for both approaches to debugging.

6.3.1 Logging

By nature of being a blocks-based programming environment, StarLogo Nova can do much better than just outputting logs to a console. Similar to the YinYang editor made by Microsoft Research, StarLogo Nova can encode information about how frequently code is run in the blocks[6], perhaps by altering the color or saturation of frequent code paths. This can help users quickly determine if a certain `if` condition is never met.

For data types, StarLogo Nova can show a distribution of the values of data adjacent to the blocks, allowing users to get a feel for the range of values actually being used in their program.

Although it is possible to instrument the generated JavaScript with tracing and recording of values, it would be simpler to instrument the StarLogo interpreter to trace and record values: instead of having to update every single instruction to generate a different set of JavaScript just for debugging purposes, we can instead just update the interpreter. This requirement underscores the necessity of test cases to help maintain

the correctness of all three methods of executing blocks: interpretation, possibly yielding code, and non-yielding code.

6.3.2 Stepping

Stepping can also take advantage of the StarLogo interpreter to allow users to walk through their scripts block by block. Breakpoints, step in/out, and many powerful stepping tools can be implemented by requiring the interpreter to yield information about the node to be executed and the type of node for every single instruction (not just yield instructions). If a user chooses to resume execution, the stepping layer can suppress yields and continue stepping until it reaches a breakpoint block. If a user chooses to step into a procedure call, then the stepping layer can just step normally. If the user chooses to step over a procedure call, then the stepping layer can suppress procedure calls, maintaining a counter of procedure entrances and exits. Once the number of exits equals the number of entrances, then the stepping layer can stop suppressing yields. If the user chooses to step out of a procedure call, then the stepping layer can suppress yields, maintaining a counter of procedure entrances and exits. Once the number of exits exceeds the number of entrances, then the stepping layer can stop suppressing yields.

The interpreter can be further modified to eval user-supplied scripts immediately before the execution of each node to allow deep inspection of the current state of execution.

All of these features add considerable weight to the interpreter and can make it incredibly slow. However, we expect the user to understand the tradeoff between debuggability and performance and to not expect fast simulations while they are tracing execution.

6.3.3 Alternative Stepping Design

An alternative to creating a StarLogo Nova specific stepping debugger is to show users how they can inspect the state of their code directly with their built-in browser

debugging tools. Although this alternative approach can have a higher learning curve than an in-house stepping debugger, it can introduce students to a powerful tool that many web developers use.

Furthermore, the browser debugger can do more than inspect the state of their code: it can inspect all of the StarLogo Nova engine and show the curious user how agent management and thread scheduling is done underneath the hood. It can also show users how their blocks map to an actual scripting language. Another benefit to this alternative approach is that it already works — all that there is to do for this feature is to write up instructions for users.

6.4 User Scripting

One common criticism of blocks-based programming is that it can be very verbose and require many individual user actions to build something simple. For example, a simple expression for calculating the average of 4 numbers $\frac{a+b+c+d}{4}$ can take over 8 blocks and many more clicks to construct.

Allowing advanced users to create their own text-based scripts will allow them to harness the execution environment and 3D renderer to run very powerful simulations. Since the engine already runs JavaScript threads, much of the backend support for running user defined scripts is already in place.

To give the user examples of how to write their text-based scripts, we can show the generated JavaScript for blocks they have in their ScriptBlocks workspace.

6.4.1 Security

One area that we must be careful about with user scripting is security. The current security policy for StarLogo Nova projects is relaxed since all current blocks have limited access to the browser scope. No blocks have the ability to directly interact with the DOM or make `XMLHttpRequests`. Because this guarantee makes StarLogo Nova projects safe, anyone can run anyone else's public project and even remix that project.

However, once we open the door to user scripts, all of those guarantees fly out the window. A malicious user could construct a user script that steals the current session cookie and sends them to a remote server the malicious user controls and anyone running this script could lose their account to this malicious user. Another attack could be constructing username and password boxes in the DOM, letting the browser’s password manager fill in the appropriate credentials, and sending those credentials off to the attacker-controlled server. The list of possible attacks is endless if an attacker is allowed full access to execute JavaScript under the same origin as StarLogo Nova.

To mitigate all these attacks, the compiler should generate functions and generators by using `evalInSandbox` instead of `eval`. `evalInSandbox` operates like `eval` in many cases, but has a very restricted set of global variables, preventing it from accessing cookies or making remote requests.

There may be demand by power users to access some of these sandboxed off utilities. For example, a user may want to interface with networked devices during their simulation. To serve this segment of the StarLogo Nova community, it may be worthwhile having separate sandbox semantics depending on who is executing the program: if the person executing the program is the original author (and the project is not a remix), then they are be allowed to run arbitrary JavaScript.

6.5 Tooling

While many tools are in place to keep the codebase tidy and correct, there are no tools in place to keep the code fast. Tooling is especially important in this case because it is difficult for a human to ascertain that a change won’t cause performance regressions. As described in section 5.5.2, even a seemingly innocuous change to add a few auxiliary data structures can have a deleterious impact on performance.

StarLogo Nova should run automated performance tests across a range of browsers and potentially a range of various hardware devices. To automate performance tests, StarLogo Nova can use Selenium, a scriptable browser driver that can be configured

to run benchmarks. These automated benchmarks can help StarLogo Nova adapt to future browser releases that may alter timing profiles and change fundamental assumptions about what techniques are fast and what techniques are slow.

The current build process operates only locally: a contributor must opt into this build process. If a contributor forgets to run unit tests, there currently isn't an automated mechanism to catch errors. Thus, it would be good to run a build server (**buildbot** and **travis** are possible candidates) for all pull requests into the development. It would also be good to prevent users from directly merging their feature branches directly into development. They should first issue a pull request, which triggers the test suite (see sections 4.1 and 4.2). After the test suite verifies the correctness of the branch and the branch has been peer reviewed, then contributors should be allowed to merge in their branch.

Chapter 7

Conclusion

StarLogo Nova is an online blocks-based simulation platform designed to help pre-college students create powerful decentralized models to understand the world. This platform was initially built in ActionScript3, which is being deprecated by many browsers because of Flash security vulnerabilities. To respond to this, I migrate the ActionScript3 codebase to TypeScript, a weakly typed language that compiles to JavaScript.

During the migration process, I introduce a build and validation and testing process aimed at keeping the new codebase healthy. This process includes automatic code formatters, linters, and test cases.

StarLogo Nova is a demanding application in terms of performance since it tries to simulate potentially hundreds or even thousands of agents in realtime. I engineer the StarLogo Nova engine for performance, beating the previous engine by margins of 280% to 2300% in key benchmarks. These performance enhancements enable the engine to run a cycle in under 10 milliseconds on relatively slow hardware and open the door to faster, larger, and smarter simulations.

Bibliography

- [1] Flash and chrome. <https://chrome.googleblog.com/2016/08/flash-and-chrome.html>. Accessed: 2016-08-15.
- [2] Google for education > blockly. <https://developers.google.com/blockly/>. Accessed: 2016-08-15.
- [3] Reducing adobe flash usage in firefox. <https://blog.mozilla.org/futurereleases/2016/07/20/reducing-adobe-flash-usage-in-firefox/>. Accessed: 2016-08-15.
- [4] What made lisp different. <http://www.paulgraham.com/diff.html>. Accessed: 2016-08-15.
- [5] Yet more bad news for flash as google chrome says goodbye. <https://nakedsecurity.sophos.com/2016/05/18/yet-more-bad-news-for-flash-as-google-chrome-says-goodbye-sort-of/>. Accessed: 2016-08-15.
- [6] Sean McDirmid. Usable live programming. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 53–62. ACM, 2013.
- [7] Mitchel Resnick. Decentralized modeling and decentralized thinking. In *Modeling and simulation in science and mathematics education*, pages 114–137. Springer, 1999.
- [8] Kevin Wang, Corey McCaffrey, Daniel Wendel, and Eric Klopfer. 3d game design with programming blocks in starlogo tng. In *Proceedings of the 7th international conference on Learning sciences*, pages 1008–1009. International Society of the Learning Sciences, 2006.
- [9] Uri Wilensky. {NetLogo}. 1999.
- [10] Nicholas C. Zakas. *High Performance JavaScript*. O'Reilly Media / Yahoo Press, 2010.