StarLogo TNG:
The Convergence of Graphical Programming and Text Processing

by

Corey McCaffrey

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 26, 2006

Copyright 2006 Corey McCaffrey. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author____
Department of Electrical Engineering and Computer Science
May 26, 2006

Certified by____
Eric Klopfer
Thesis Supervisor

Accepted by____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

StarLogo TNG:
The Convergence of Graphical Programming and Text Processing
by
Corey McCaffrey

Submitted to the
Department of Electrical Engineering and Computer Science

May 26, 2006

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

StarLogo TNG is a robust graphical programming environment for secondary students. Despite the educational advantages of graphical programming, TNG has sustained criticism from some who object to the exclusion of a textual language. Recognizing the benefits of text processing and the power of controlling software with a keyboard, I sought to incorporate text-processing techniques into TNG's graphical language. The key component of this work is an innovation dubbed "Typeblocking," by which users construct block code through the use of a keyboard.

Thesis Supervisor: Eric Klopfer
Title: Director, MIT Teacher Education Program

Acknowledgments

First, I would like to thank my research advisor, Eric Klopfer. I feel extremely lucky to have joined his team as an undergraduate researcher at the beginning of my sophomore year. I admire his passion for education, commitment to his students, and great sense of humor, all of which make him a pleasure to work for. I would also like to congratulate Eric for receiving tenure this month. מזל טוב!

I would also like to thank Eli Meir and his company, Simbiotic Software, for providing me with new and exciting opportunities to extend StarLogoBlocks and for helping to support my graduate studies.

Thank you to Daniel Jackson and Srini Devadas, professors of 6.170: Software Engineering Laboratory, who gave me the opportunity to teach and helped support my graduate studies.

I would like to thank Andrew Begel, the lead designer and developer of LogoBlocks and many versions of StarLogo, including TNG, who was my mentor for the past four years. Andy contributed immeasurably to my technical skills for which I feel greatly indebted.

Thank you to Daniel Wendel, my programming partner and fellow graduate researcher, who helped make this past year's work and courses seem fun and easy.

Thank you to David Greenspan, Adam Rosenfield, William Jacobs, Lauren Clement, and Lawrie Gibson, undergraduate researchers who tirelessly pushed the limits to make StarLogoBlocks what it is today.

I would like to thank the many undergraduate researchers who contributed to other components of StarLogo TNG, including David Blau, Viknash Samy, Radu Berinde, Aleksander Zlateski, Ricarose Roque, John Jackman, Mike Matczynksi, Mike Lin, Jane Lanyue, and Kevin Wang.

Thank you to Rob Miller, who teaches an excellent course called 6.831: User Interface Design and Implementation, where I learned many of the techniques and methodologies I applied to this work.

Thank you to my parents Kevin and Jan, and my sister Caitlin, who showed me incredible love and support during my years at MIT and always reminded me not to work too hard.

Finally, thank you to Michelle Desnick, who always reminded me to work at least hard enough to graduate on time. Her love and encouragement over the past four and a half years have meant the world to me. Michelle, I cannot thank you enough.

Table of Contents

Acknowledgments **3**

Table of Contents **4**

List of Figures **6**

1 Introduction **7**
- 1.1 The Questionable Inevitability of Text 7
- 1.2 My Solution 8
- 1.3 Outline 9

2 Text Processing **9**
- 2.1 Keyboard Entry 10
- 2.2 Word Processors 11
- 2.3 Text-Assist IDEs 11

3 Graphical Programming **13**
- 3.1 Direct Manipulation 13
- 3.2 WYSIWYG Markup 14
- 3.3 GUI Builders 15
- 3.4 Tangible Programming 15
- 3.5 Visual Syntax 16
  - 3.5.1 Correctness and Control Flow 17
  - 3.5.2 Fun and Educational 18
- 3.6 The Magnetic Poetry Effect 19

4 Hybrid Systems **20**
- 4.1 Leogo 20
- 4.2 Pet Park Blocks 21
- 4.3 Alice2 21
- 4.4 Tinkertoy 22

5 Keyboard Shortcuts **23**
- 5.1 Efficiency 23
  - 5.1.1 Flow 24
  - 5.1.2 Speed 25
- 5.2 Expectations 26
  - 5.2.1 User Expectations 26
  - 5.2.2 Developer Expectations 27
- 5.3 Learnability 27
  - 5.3.1 Visibility 28
  - 5.3.2 Social Factors 29

6 StarLogo TNG **30**
6.1 History 30
6.1.1 StarLogo 30
6.1.2 LogoBlocks 32
6.1.3 StarLogo TNG 33
6.2 Goals 34
6.3 Users 34
6.3.1 Students 35
6.3.2 Teachers 35
6.3.3 Researchers 35
6.3.4 StarLogo 2 Users 36
6.4 Spaceland 36
6.5 StarLogoBlocks 37
6.5.1 Overview 37
6.5.2 Managing Complexity 41
6.5.3 Reactions 47

7 The Mythical Algebra Block **49**
7.1 Algebraic Expressions 49
7.2 Calculator Syndrome 50
7.3 The Myth of the Algebra Block 51

8 Typeblocking **53**
8.1 Block Cursor 53
8.2 Workspace Navigation 54
8.2.1 Search 54
8.2.2 Zoom 56
8.3 Context Awareness 56
8.4 Auto-Completion 58
8.5 Animation 58
8.6 Typing Modes 60

9 Future Work **62**
9.1 Alternative Input Modalities 63
9.1.1 Speech Recognition 63
9.1.2 Gesture Recognition 64
9.2 Accessibility 64
9.3 Collaborative Programming 65

10 Conclusion **65**

References **67**

List of Figures

Figure 1: StarLogo 2.2 - Textual code and 2D rendering of agent-based model 31
Figure 2: StarLogo TNG - Graphical language and 3D renderer 33
Figure 3: Rich 3D environments in StarLogo TNG [10] 37
Figure 4: Overview of StarLogoBlocks 38
Figure 5: Properties are magenta, and commands are colored by category. 39
Figure 6: Triangular port shape only accepts number blocks... 40
Figure 7: The "If" block stretches to accommodate the blocks inside. 40
Figure 8: User-defined parameterized procedure and dynamically generated call block. 40
Figure 9: Easy substitution for blocks with a family 42
Figure 10: Blockdoc tool tips help users remember what each block does. 42
Figure 11: Procedures collapse to utilize space more efficiently. 43
Figure 12: The equality block morphs to contain Boolean, number, and string values.... 45
Figure 13: The Breed Bar with the "Turtles" drawer open 46
Figure 14: Zoom in to focus on a particular stack of blocks. 46
Figure 15: Zoom out to get an overview of a project... 47
Figure 16: The Minimap provides easy navigation of the block workspace.................... 47
Figure 17: Search reveals blocks and categories that match the query............................ 55
Figure 18: Scatter PC accepts multiple blocks of the same type. 57

1 Introduction

The art of programming has come a long way since the days of mechanical punch cards. Today, programmers write code in increasingly higher-level languages that make short work of once arduous tasks, giving them time to take on more challenging endeavors. The tools of their craft have evolved as well. Although the keyboard still resembles those found on the typewriters of old, intelligent computer software now processes each keystroke.

As technology improves, educators look for ways to improve the art of teaching programming as well. Today's complex software technology raises the bar higher for students and teachers who want to create modern-looking programs. Fortunately, it also provides opportunities to create more advanced programming teaching tools that go beyond traditional text processing. The relatively recent innovation of the Graphical User Interface (GUI) has given rise to many graphical programming environments designed to teach the fundamentals of computer programming to students without the steep learning curve of more powerful traditional textual languages.

1.1 The Questionable Inevitability of Text

This thesis examines the trade-offs between programming and learning to program in graphical and textual environments and explores the following question: Can a graphical programming environment support novice and expert users alike without relying on an underlying textual language?

1.2 My Solution

In order to study this question, I worked with StarLogo TNG. "The Next Generation" of StarLogo is a programming teaching tool under development within the Teacher Education Program at the Massachusetts Institute of Technology [5, 13, 14, 16, 20]. The curriculum guidebook written for the previous version of StarLogo begins with an explanation of the purpose of StarLogo:

> StarLogo is designed especially for helping people create models of decentralized systems—that is, systems in which patterns arise from interactions among lots of individual objects. For example, StarLogo is well designed for exploring how bird flocks arise from interactions among individual birds, or how traffic jams arise from interactions among individual cars [7].

The wide variety of decentralized systems in life and the depth of understanding they provide students make StarLogo an excellent tool in classes ranging from biology and physics to math and economics.

StarLogo TNG is a purely graphical programming environment. Featuring a puzzle piece-like procedural language called StarLogoBlocks and a three-dimensional output renderer called Spaceland, StarLogo TNG seeks to motivate today's students raised on visually compelling video games to learn to program. In addition to serving as a programming teaching tool, the models developed with StarLogo TNG provide interdisciplinary learning opportunities. Designed to support novice students as well as expert researchers, I chose StarLogo TNG as the ideal software system for studying the convergence of graphical programming and text processing. It sports an advanced graphical programming environment and, as a design goal, maintains no underlying textual language [5]. Instead of grafting on a textual language to make the environment more appealing to seasoned programmers, I extracted techniques common to text processing environments and applied them to StarLogoBlocks. To some extent, this task consisted of carefully designing keyboard shortcuts to control the block code with

efficient keystrokes. More importantly, this work led to an innovation that I call "Typeblocking," which provides a framework for constructing block code by typing instead of dragging a mouse.

1.3 Outline

This thesis begins with an overview of text processing and graphical programming environments, including their relative strengths and weaknesses. Next, it presents the state of the art of existing hybrid solutions for graphical programming and text processing. Based on the premise that many of the benefits of a textual language are derived from the use of a keyboard, the following section discusses recent findings regarding the benefits and usability of keyboard shortcuts. Then it introduces StarLogo TNG, along with the application of Typeblocking and the convergence of graphical programming and text processing in depth. This thesis concludes with a proposal for further study and an assessment of the potential value of this work.

2 Text Processing

Fundamentally, textual languages are composed of just that—text, and nothing more. Written characters and words, typically formed by typing on a keyboard, appear on the screen or printed page interspersed with symbols such as braces and parentheses. In fact, one of the advantages of a textual language is that it does not require any special software to change it. Programmers are free to read and write textual code in any text editor they like. This section considers some of the features provided by modern text processors to facilitate writing code.

2.1 Keyboard Entry

The keyboard is the ideal tool for crafting textual code, because it provides a key for each character in the code, and users can train themselves to type quickly. In fact, early text editors were designed for exclusive control with a keyboard, since they were created before the advent of GUIs and mice.

There is a substantial body of evidence suggesting that the keyboard is a more efficient input device than the mouse for expert users [1, 15,19]. Nevertheless, keyboards can be intimidating to novices. Typing comes with a steep learning curve and requires extensive training in hand posture and movement. It also requires a degree of literacy that younger children may not have developed yet. With over a hundred keys at their disposal, sometimes even experienced typists need to look at the keyboard, taking their eyes and their attention away from the task on the screen.

Furthermore, the act of programming with a keyboard multiplies the difficulty level. Just like a writer, a programmer also suffers from writer's block, struggles with word choice, and scans carefully to ensure correct spelling and syntax. The difference is that a human being can often understand even the poorest writing, whereas a computer demands unambiguous perfection. "Entering programs as text can be much harder than alternatives such as direct manipulation or form filling but often gives the student more power," according to a survey of programming environments for novices [11]. My impression of their results is that the power comes specifically from the speed of the keyboard, but the power is difficult to obtain.

2.2 Word Processors

There is more to understanding the power of text processing than simply the keyboard input device. Even the most trivial word processors do so much more than typewriters, which simply echo the characters you type to the printed page. A blinking cursor allows the writer to navigate through pages of digital text, deleting, overwriting, and even inserting characters anywhere. Users can cut, copy, and paste large swaths of text without ever picking up scissors or tape. Type a word or phrase into a search bar, and the program will show the user every instance of text within dozens or even thousands of pages. These commonplace features that we now mostly take for granted were revolutionary for typists accustomed to mechanical typewriters and printed documents.

2.3 Text-Assist IDEs

Like word processors, which were designed to alleviate some of the hassles for writers, many advanced text editors and Integrated Development Environments have been created to ease some of the pain that programmers used to experience. These include Emacs, Eclipse, Sun Microsystem's NetBeans, Microsoft's Visual Studio, Apple's XCode, and JetBrain's IntelliJ IDEA, to name just a few. These IDEs support a multitude of textual computer languages such as Java, C/C++, Perl, Python, the .NET platform, and more.

Although it is entirely possible to write thousands or millions of lines of Java code in Microsoft Notepad, text-assist IDEs incorporate many features besides integrated compilers and debuggers to make writing code easier:

- *Syntax Highlighting:*

  By automatically color-coding categories of text such as method names, variable names, keywords, and comments, programmers can quickly identify key components of the code.

- *Outline View:*

  By organizing the component hierarchy of a code file, programmers can quickly jump to the section they wish to modify.

- *Search:*

  In addition to providing the search capabilities of a typical word processor, most IDEs provide advanced filters to search for or within specific code modules.

- *Auto-Complete:*

  This feature eases two burdens. First, higher-level languages encourage verbose naming conventions, and auto-complete finishes partially typed words to save time. Second, component names and parameter lists are hard to remember, and auto-complete provides a list of valid completions for a particular section of code.

- *Refactor:*

  Recognizing that programmers occasionally need to move, rename, or change their component interfaces, many IDEs can scan entire projects and make appropriate changes to affected code.

- *Auto-Format:*

  Textual languages have both mandatory formatting rules and conventional styles. IDEs perform auto-formatting to remind the programmer of the rules and often provide preferences to customize the style applied when formatting.

- *Find Declaration:*

  Typically for procedural languages, in which reusable code segments are componentized and called from other parts of the code, it is often useful to review the original code segment. Find Declaration is a special search that automatically jumps from a procedure call or other instances of component usage to the code that defines that component.

- *Call Hierarchy:*

  In many ways the inverse of Find Declaration, Call Hierarchy is a specialized search that scans an entire project to construct a tree of potential stack traces. It allows programmers to find quickly every method that calls a particular method, and which other methods call those methods, all the way to the beginning of a potential execution sequence.

Some of these features such as syntax highlighting are applied automatically, whereas other features are invoked by the user, either via a GUI element such as a toolbar button or by a keyboard shortcut that the user has memorized. It is worth noting that while these features are now common to professional IDEs supporting textual languages, all of these techniques can be applied to graphical programming languages as well.

3 Graphical Programming

Despite the many advantages afforded to contemporary programmers by the latest IDEs, textual languages remain challenging, to novices in particular. Graphical programming environments strive to ease those challenges with the help of a well-designed GUI.

3.1 Direct Manipulation

Direct manipulation is a common catchphrase for GUIs that highlights their key usefulness; everything is point-and-click. Users *press* buttons drawn on the screen. They *drag* files and *drop* them into folders. Users *slide* a bar along a track. Graphical languages such as StarLogo TNG utilize common GUI metaphors to create an intuitive interface. The development team noted that as the student users "were already familiar with using a computer by dragging and dropping objects, StarLogo TNG came very naturally to them [20]."

Of course, applying the term “direct manipulation” to GUIs is somewhat misleading. Sure, GUIs emulate real knobs, buttons, and switches with startling realism, but pointing and clicking with a mouse is hardly direct manipulation. A touch screen might be better, and a tactile, tangible interface better still. Nevertheless, GUIs still deliver more natural direct manipulation than a keyboard-controlled interface, which is a significant advantage for novices learning a new system.

3.2 WYSIWYG Markup

Recognizing the pain associated with writing code from scratch, many software programs provide features to make the job easier. When creative writers and publishers first wanted to use computers to create fancy documents and web pages, they had to learn markup languages such as HTML to tag their text with appropriate formatting. This challenge gave rise to What-You-See-Is-What-You-Get (WYSIWYG) markup, software that shows you the available formatting options and what the formatted text will look like up front without having to code a single tag. If a user wants bold text, with the push of a button a block of text suddenly looks bold. The best word processors all rely heavily on GUI elements such as menus and toolbars to expose a myriad of formatting options to the user, and the mouse allows the user to drag images and other media within a document instead of describing its position with textual tags. Nonetheless, WYSIWYG word processors and web editors are not graphical programming environments; they are computer-aided publishing tools in the same sense that architects and engineers use CAD software to perform Computer Aided Design.

3.3 GUI Builders

A GUI builder is a tool, usually part of an IDE, which facilitates the construction of GUIs by allowing programmers to drag and drop components such as buttons and text fields in a window. Instead of describing the GUI in textual code, the user is treated to a WYSIWYG experience. Some languages, such as Microsoft Visual Basic, were designed around this idea. However, "'visual' is a misnomer, especially given that, when Microsoft says 'visual' they really mean textual augmented with a direct manipulation GUI builder [17]." The distinction is that programmers construct the graphical component of their software graphically as opposed to constructing their code itself graphically.

3.4 Tangible Programming

Some language designers who believe in the principles of graphical programming also view its pseudo-direct manipulation as a serious flaw. This belief has resulted in many efforts to create tangible programming environments, such as Timothy McNerney's Tangible Programming Bricks [17]. The basis of McNerney's thesis is that, "constructing and modifying programs using even the most modern GUIs is an unnecessary obstacle to programming [17]." Specifically, McNerney perceives:

> Screen-based graphical programming languages suffer from a number of limitations: Tools for manipulating textual programming languages are much more mature than graphical programming tools. Textual programming languages make better use of screen real estate than graphical programming languages, which often include extra decorations around each functional block [17].

However, his criticism of the most modern graphical programming implementations at the time of his writing does not adequately reflect the state of current solutions, nor does it explain why tangible programming survives the same criticism.

Nevertheless, working under the assumption that physical bricks better satisfy the educational goals of graphical programming, McNerney compares his system to Andrew Begel's LogoBlocks, the innovative graphical programming language that was also the precursor to StarLogoBlocks [2]. Tangible Programming Bricks are "much like the Logo Blocks [sic] system, which uses a 'puzzle pieces' metaphor for connecting blocks, and makes a distinction between control flow and data flow connections [17]." Although tangible programming certainly supports direct manipulation better than GUIs, it has other more serious shortcomings. For example, McNerney correctly observes, "One limitation of Logo Blocks that is also an issue for tangible programming is the fixed size of the blocks, which sometimes makes it cumbersome to assemble certain legitimate programs without introducing 'padding' blocks. In Logo Blocks, this could be solved by making certain blocks stretchable [17]." While LogoBlocks never did make certain blocks stretchable, a key feature of StarLogoBlocks is dynamically stretching blocks that expand to accommodate other blocks that they contain.

Unconstrained by physical reality, GUIs evolve in ways that are more difficult or impossible for tangible programming systems. Finally, though tangible programming systems embrace many of the benefits of graphical programming, they forego the possibility of incorporating the many advantages of text processing.

3.5 Visual Syntax

As it applies to programming, syntax comprises the set of rules for a particular language that dictate the arrangement of characters and symbols that create well-formed code. Unlike written human languages that can still be understood with some errors, computer code must be flawless because today's computers require unambiguous instructions as determined by the positions of the symbols defined in the language. The universal and fundamental feature of all graphical programming environments is the use of visual

syntax. "Using graphical representations of objects, you can more concretely show object orientation … eliminate annoying syntax (like {}'s and ()'s in C, BEGIN and END's and ()'s in Pascal, and ()'s in Lisp) and better visualize the pathways that your program is following [2]."

3.5.1 Correctness and Control Flow

In recent classroom testing of StarLogo TNG, we observed that the "visual blocks provided a certain amount of implicit programming guidance that text does not offer [20]." Specifically, there is no risk that a student will use a curly brace where a square bracket was required or a number where the language required a Boolean value, nor is there a risk of unmatched parentheses or misspelled commands, because the visual syntax enforces these rules automatically. "Novice programmers need only to recognize the names of commands and the syntax of the statements is encoded in the shapes of the objects, preventing them from creating syntactically incorrect statements [11]."

Programmers frequently describe two classes of code defects. The first kind is a syntax error such as the ones described above, and they prevent their code from running at all. It is similar to a teacher telling a student that he will not read a paper until the student has crossed every 't' and dotted every 'i'. "Often a major stumbling block to teaching kids to program is that they find the syntax overwhelming [2]." The more complex the syntax, the greater the number of defects, and the sooner a novice will give up in frustration.

The second kind of code defect is a logic error. These arise when the programmer runs a syntactically correct program, but the program does not behave the way the programmer intended. It indicates a problem with the content of the code as opposed to the form of the code. For our StarLogo TNG students, "When bugs did happen, the students ended up debugging their programming logic rather than syntax [20]." Despite that logic errors are typically more insidious, programmers may derive greater satisfaction and learning

from finding and correcting a logic error. This apparent paradox is due to the fact that fixing logic errors leads to a deeper understanding of the user's program, whereas fixing syntax errors leads to increased frustration directed at the restrictions imposed by the rules of the language.

In addition to eliminating syntax errors, visual syntax also depicts control flow more clearly than sequences of characters in textual languages. When tracking logic errors, students "often pointed to and followed the programming blocks as they were debugging [20]." There was never a question as to whether a particular brace terminated a loop or a conditional branch; the shape of the enclosing block made it easier to comprehend where the computer was going to go next. Furthermore, "Parallelism can also be made more explicit; all of the different program clusters on your screen can run at the same time," unlike sequences of characters that appear linear but may not run linearly [2]. This feature is especially important for StarLogo, because hundreds or thousands of simulated agents could be running different clusters of code simultaneously.

3.5.2 Fun and Educational

As a corollary to the notion that visual syntax eliminates syntax errors, it makes the act of programming more fun: "The nature of the programming blocks prevented students from making errors that would usually frustrate them; it kept their interest level high without getting bogged down [20]." The shallow learning curve and lively animated blocks "usually yielded instant gratification that the kids enjoyed [20]."

Fortunately, the students were not only having fun but learning to program as well. In spite of only weekly meetings, "the highly visual aspect of StarLogo TNG made it easy for students to recall blocks they learned in the previous weeks. They only needed to recognize the blocks as opposed to the commands plus the syntax [20]." In other words,

visual syntax encourages students to exercise their programming skills, not their memory of arbitrary rules and symbols.

3.6 The Magnetic Poetry Effect

With visual syntax and a repository of command blocks, graphical programming environments give rise to what I call the Magnetic Poetry Effect. The vast majority of people are not poets. When confronted with a blank page, many would be hard-pressed to compose a poem that they would proudly share with the rest of the world. And if one imposes any structural requirement, be it iambic pentameter or rhyming, fewer still could produce a sonnet or even a limerick.

Enter magnetic poetry. This extremely popular toy consists of nothing more than a box of flat magnets with single words printed on them. Far from depicting every word in the dictionary, a magnetic poetry set contains carefully selected words to aid the aspiring writer in waxing poetic. Give a magnetic poetry kit to the same person who sat dumbstruck in front of that blank page, and odds are his refrigerator will be littered with philosophical musings in no time.

The irony is that the refrigerator surface is still a blank page, and the user has fewer options than he did with a pen, which is capable of tracing out any word in the dictionary. Thus, the Magnetic Poetry Effect is twofold:

- The concept of sliding magnetic words together is novel, making the act of composing poetry with magnets new and fun.
- Fewer options, all laid bare before the hopeful poet, reduce the time it takes to select a sequence of words and improve the poet's confidence that the words selected will sound poetic.

Graphical programming environments share the Magnetic Poetry Effect. The concept of dragging blocks together is novel, making the act of programming with blocks new and fun. Also, a limited set of commands laid out in a list of visible blocks reduces the time to find a useful block and improves the programmer's confidence that the blocks selected will form a correct program. In our recent study, we concluded that "the advantages provided by block programming lowers [sic] the entry point for programming, and the built-in error prevention mechanisms give students a more structured programming environment than that of the traditional text entry model [20]." Just as text poets and magnetic poets both start with a blank slate, both textual and graphical languages begin with an empty page. Nevertheless, thanks to the Magnetic Poetry Effect, graphical programming environments provide structure that enables novices to program in situations where they otherwise would not have.

4 Hybrid Systems

Up to this point, the systems described have been purely textual or purely graphical. Some developers recognize that both have advantages as well as shortcomings. In some environments, developers added graphical modes to express preexisting textual languages, and in other environments, developers added underlying textual languages to support preexisting graphical frameworks. The latter developers believed that pure graphical languages "[lead] to frustration for sophisticated programmers who want to concisely express a statement that might be better represented using text [2]."

4.1 Leogo

According to the 2005 survey of programming languages for novices, Leogo was the only programming environment classified as, "Provide[s] Multiple Methods for Creating Programs [11]." Although several systems provided some degree of both graphical and

textual programming, Leogo was the only system they studied that was designed to support three different program creation methods equally. Specifically, Leogo "provides three [methods]: a typed syntax similar to Logo, a direct manipulation interface in which the turtle is dragged around and his actions are recorded, and an iconic language which contains templates for defining structures and using common turtle commands [11]." Whenever the user changes one of the representations of the program, the other two representations automatically update to reflect that change.

4.2 Pet Park Blocks

According to the 2005 taxonomy, "Pet Park Blocks is a graphical programming language, inspired by LogoBlocks… Pet Park Blocks provides a button that allows users to see their Blocks program as a textual program. This allows users to gradually transition to text-based programming [11]." Unlike Leogo, which provides a textual language mode as a primary interface, Pet Park Blocks only provides text for users to read a textual representation of the blocks code, under the assumption that it helps users learn to use traditional textual languages.

4.3 Alice2

Alice2 has the distinction of being one of few complex graphical languages: "Where many no-typing programming systems present users with only a few of the standard programming constructs, Alice allows students to gain experience with all of the standard constructs taught in introductory programming classes without making syntax errors [11]." Specifically, these constructs include parameterized procedures, conditional branching, variables, loops, and parallelism [10]. Incidentally, StarLogoBlocks is another complex graphical language that supports all of these constructs as well.

Results of user testing of an older version of Alice with a textual language supported the benefits of visual syntax. Their tests "revealed that the necessity to enter programs by typing was frustrating for beginning programmers: 65% of users cited the need to type and 45% cited difficulty with remembering the syntactic details as one of the worst three things about Alice. For these users, typing was a dominant problem in learning to program [10]." The Alice developers interpreted these results as justification for shunning the keyboard in Alice2. There is no data available indicating whether the 35% of users who did not cite the need to type as one of the worst three things about Alice would cite the need to drag with a mouse as one of the worst three things about Alice2.

Like Pet Park Blocks, despite the inability to type programs, Alice2 also believes that some exposure to textual languages is important to ease the transition to them later. The Alice2 developers have added "the capability to render programs in Java-style syntax [10]." They intend to study whether using this feature will make it easier for college students to learn Java, but no results are available at this time [10].

4.4 Tinkertoy

The hybrid programming environment with goals most similar to StarLogo TNG is Tinkertoy, whose creator wrote, "Although many of the long term benefits of going from text based systems to systems like Tinkertoy come from the graphic representation, in the short term, fast interaction is more important [8]." For Tinkertoy, fast interaction is achieved with a keyboard.

Like the other hybrid environments described above, users can create programs entirely with blocks, without using a keyboard. After all, Tinkertoy is first and foremost a graphical representation of Lisp, which originated as a text-only language. On the other hand, users already familiar with Lisp may prefer to type at least some portions of their code. Unlike Leogo, which could run a program written entirely in text, Tinkertoy could

only convert a chunk of Lisp code into an icon and vice versa. Though the limited textual representation may sound less powerful at first, it has unique advantages over Leogo. The interface suggests that the graphical representation is the primary one on which the user ought to focus, and the textual representation is merely a tool to construct the graphical representation more quickly.

While Pet Park Blocks and Alice2 developers view their graphical languages as transitional to ones leading to the use of traditional text languages, Tinkertoy's developer viewed the textual language as a facility for creating the more innovative graphical structures. Though left unimplemented, he even suggested employing keyboard commands and text editor functionality to improve the efficiency of editing the blocks directly, much in the way that StarLogo TNG does [8].

5 Keyboard Shortcuts

In light of the relative advantages of graphical programming and text processing, as well as the various solutions attempted by existing hybrid environments, the premise of this thesis is that the optimal hybrid environment should not have an underlying textual language at all. Rather, proven techniques for enhancing text processing should be applied to block processing, and the primarily graphical programming environment should support the use of a keyboard for manipulating blocks. This section focuses on the benefits of keyboard shortcuts and why the use of a keyboard is the primary advantage of textual languages over existing graphical ones.

5.1 Efficiency

One of the biggest shortcomings of direct manipulation, the hallmark of a GUI, is that it really is a drag, both literally and figuratively. Users typically do not save time with a

mouse. For example, moving one's hand from the keyboard while typing, doing a visual search to find the "Bold" button, and moving the mouse across its pad to reach the button cannot compete with the speed of typing Command-B on the keyboard.[1]

5.1.1 Flow

A 2004 study on designing interfaces for staying in the "flow" describes being in the flow as being "fully engaged and in control of an activity, … immersed in that activity to the exclusion of all else. Furthermore, people regularly describe these experiences as some of the best of their lives [1]." Programming is one activity on a short list that the author says is "likely to result in 'being in the flow [1].'"

The study explains that expert users of a particular programming interface achieve and sustain flow more easily when they reach the autonomous usage stage:

> The final autonomous stage applies to expert users that can execute an interface element without feedback from the interface. This is commonly found in GUIs with keyboard shortcuts. An expert user with touch typing skills might press the 'ctrl-c' key combination to execute the Copy command without waiting for or receiving feedback from the interface [1].

The author goes on to describe that some feedback such as an animated response from the GUI is sometimes more helpful to sustaining flow than no feedback, but emphasizes that the key point is that the user executes the interface element quickly and easily from the keyboard [1].

The author stresses that when users are no longer learning the interface, when they become experts who crave flow, it is essential for the interface to satisfy that craving. "In general, balancing the needs of novices and experts remains a daunting problem. But, it is crucial to support experts – something that is often overlooked, or left just to shortcut

---

[1] The Command key, or "Apple" key, is the rough equivalent of the Control key on a Windows or Linux PC. Similarly, Apple's Option key is the rough equivalent of Alt.

key accelerators. Many computer users become experts at specific programs over time, and providing ways for them to be extremely efficient must not be ignored [1]." We believe that this point is particularly relevant for StarLogo TNG. Unlike some of the other hybrid systems that are treated as transitional learning tools, StarLogo TNG is meant to support expert users to design increasingly complex models as they master the unique system.

Finally, it is insufficient to sprinkle keyboard shortcuts onto a GUI ad-hoc. The keyboard manipulations must be memorable, intuitive, and convenient. If the typical expert user can execute an interface element with a mouse more quickly than he can remember the equivalent keyboard control, then the keyboard control becomes useless. The author explains that "users have extremely limited short term memory. Any interface elements that strain users' memory are problematic because, again, the user's flow will be interrupted [1]."

5.1.2 Speed

In addition to helping users stay in the flow, keyboard shortcuts are fast. I postulated at the beginning of this section that dragging a mouse to a button is slower than invoking a keyboard shortcut, but you do not have to take my word for it. A 2005 study measured the relative speeds of different methods for executing interface elements, and the results are consistent with the theory: "These findings confirm that the keyboard shortcut method is substantially faster than the icon methods and that the icon method is substantially faster than the menu method. [15]"

Nevertheless, the authors are careful to point out that keyboard commands are not a panacea for controlling software. They recognize that pointing and clicking convey some interaction more effectively than an equivalent keyboard command. For example, "selecting a range of cells using keyboard commands can be very difficult … and the use

of a mouse is certainly more efficient [15]." An efficient user will strike an effective balance between keyboard and mouse usage to maximize his productivity, and the best GUIs will accommodate both input modes appropriately.

5.2 Expectations

5.2.1 User Expectations

Concurring with the recommendations of the flow study, the speed study notes that a well-designed interface "should be (a) easy for novices to learn, (b) efficient for experts to use, and (c) provide the means for users to make the transition from the easy-to-learn but inefficient methods of novices to the more difficult-to-learn and efficient methods of experts [15]." In fact, users expect good software to conform to this standard. Most new users have neither the time nor the patience to read a manual to learn the system. They expect to be able to jump right in, starting with the basics and incrementally learning new features. If the user never discovers or is never taught a particular feature, that feature is more likely to remain dormant than looked up in a manual.

On the other hand, some new users are power users. These novices are users who crave flow and efficiency so badly that they will thoroughly explore a new piece of software and its documentation to find flow enablers. For example, some derivatives of the Mozilla web browser have "a hidden 'incremental search' feature that allows users to search within a page and follow links, all from the keyboard. This is an advanced and 'scary' feature to some – but many of us that have put the energy into learning it have found that it has dramatically improved our web browsing efficiency [1]." Unfortunately, the users who enable features such as Mozilla's Find-As-You-Type feature are likely to be in the minority.

5.2.2 Developer Expectations

Since a developer has expert knowledge about the software he develops, he is in the best position to become a power user, employing advanced features that he may have implemented himself. For this reason, developers must constantly remind themselves that they are not typical users and must work hard to support typical users, from novice to expert and everything in between.

To this end, developers labor to support the transition to keyboard shortcuts and advanced features. Frequently used techniques include "Did You Know…" documentation snippets that appear when the program loads and showing the equivalent keyboard menu accelerators next to menu items. There is also substantial incentive for developers to encourage the transition to advanced features; the more efficiently a person can use a piece of software, the more the user will enjoy the software and want to continue using it and purchase newer versions.

Unfortunately, the study found "that although the keyboard shortcut method is the most efficient, it is not frequently employed. It is particularly notable that even highly experienced users rarely employ keyboard shortcuts… Therefore, even though the graphical user interface appears to support the transition from less efficient to more efficient methods, most users fail to make the transition [15]."

5.3 Learnability

Understanding the learnability of keyboard shortcuts is critical to making sense of why even many highly experienced, expert users do not use them to improve their efficiency and sustain their flow.

5.3.1 Visibility

In order for a user working alone to discover a new feature, the feature must be visible. Of course, there is a continuum of visibility. The least visible feature might be described in obtuse language deep within a disorganized thousand-page manual. Nevertheless, it is still there, leaving the possibility open for some intrepid user to stumble upon it. A more visible feature might be one that requires the user to explore a preferences dialog box and experiment with different settings. At the other end of the spectrum, popups that "show the corresponding keyboard shortcut when users position the mouse over an icon on an icon toolbar" make those shortcuts extremely visible [15]. Toolbar icons are arguably the most visible, because the user can always see them, and their position and behavior are fixed and memorable.

In one respect, the keyboard shortcuts displayed alongside menu items or in popups above toolbar icons are positioned appropriately because they are visible to the user when the user is preparing to execute that interface element, and there is a clear association between the shortcut and the element it invokes. In another respect, they could also be considered the worst place to display the shortcuts to the user. When the user is finally presented with the keyboard shortcut, he has already invested the time to move the mouse over the toolbar or scan through a long list of menu items. By the time the user sees the shortcut, it is no longer faster than clicking the mouse button to finish executing the interface element. If the user is not motivated to rehearse the keyboard shortcut after seeing it, he will probably not remember the shortcut the next time he wants to perform the same action. Again, the user will have to use the mouse to find the menu or icon, and the cycle repeats itself.

5.3.2 Social Factors

The speed study concluded, "People often adopt inefficient methods either because they do not know about efficient methods or else choose not to use/learn them. We suspect that both factors contributed to the relative lack of use of keyboard shortcuts [15]." For all the good that visibility does, it is insufficient to encourage people to use the keyboard. Another study suggests that the best way to help people transition to using keyboard shortcuts is through social interaction [19].

Among those surveyed who were identified as non-keyboard shortcut users, the results show "that the most endorsed statement was 'I would start using keyboard shortcuts if I had someone to train me to use them.' While the least endorsed statement was 'I would start using KBS if I thought they would save me time [19].'" Other questions revealed that most people learned to use keyboard shortcuts in social settings, "such as working with and watching other people who used KBS [19]." The survey results led the researchers to conclude, "An optimal training environment for the instruction of the efficient use of computer applications may be to have a group of co-workers in an interactive training setting. If the co-workers are trained together, they may then be able to act as support for each other when they return to work and implement what they have learned [19]."

Unfortunately, both providing and attending software training sessions is time consuming and expensive. For most users and most software products, the idea of getting a large group of new users to learn together is logistically impossible. Thankfully, however, the opposite is true of many educational software products such as StarLogo TNG. Designed for use in a classroom of new student users, with the promise of curriculum to help instructors prepare training sessions and tutorials for their students, educational software is ideally suited for hooking students on efficient, advanced features. Just as in a workplace environment, once a student discovers a feature with a high "cool" factor, the use of that feature will spread like wildfire through the classroom. There is incentive for

the student who found it to share the knowledge because it gives the student an opportunity to show off the student's technical skills, and there is an incentive for the student's classmates to adopt the knowledge so that they, too, can advance their own technical skills. Ultimately, keyboard shortcuts get widespread use once all the cool kids are doing it.

6 StarLogo TNG

Since StarLogo TNG is a rich graphical programming environment with no underlying textual language and limited prior use of the keyboard, it was the ideal vehicle for me to apply my theory that the well-designed application of keyboard input could multiply its usefulness to advanced users by many times. The following section introduces StarLogo TNG, starting with its history and goals and concluding with an in-depth look at its graphical programming language, StarLogoBlocks.

6.1 History

6.1.1 StarLogo

Before StarLogo "The Next Generation," there were several versions of StarLogo that preceded it. StarLogo began as a textual dialect of Logo implemented to run on a parallel processor computer. As technology evolved, MacStarLogo simulated the parallelism of the original StarLogo on an Apple Macintosh personal computer. Later, with the advent of Java, StarLogo 2 was created to provide a more versatile, cross-platform edition of StarLogo (Figure 1). StarLogo 2.2 is the most recent stable edition of StarLogo and may be downloaded for free from the official web site [12].

Unlike many derivatives of Logo, which are neither computationally intensive nor suitable for modeling complexity, StarLogo presents unique challenges to the developers.

According to the StarLogo 2 lead designers, “Building an agent-based modeling environment like StarLogo is not a trivial task. It involves balancing the pedagogical needs of students with the efficiency requirements of running thousands of agents at the same time [4].” In between attending workshops to train secondary school educators to incorporate StarLogo into their classrooms, they regularly receive questions, comments, and criticism from graduate students and scientific researchers.

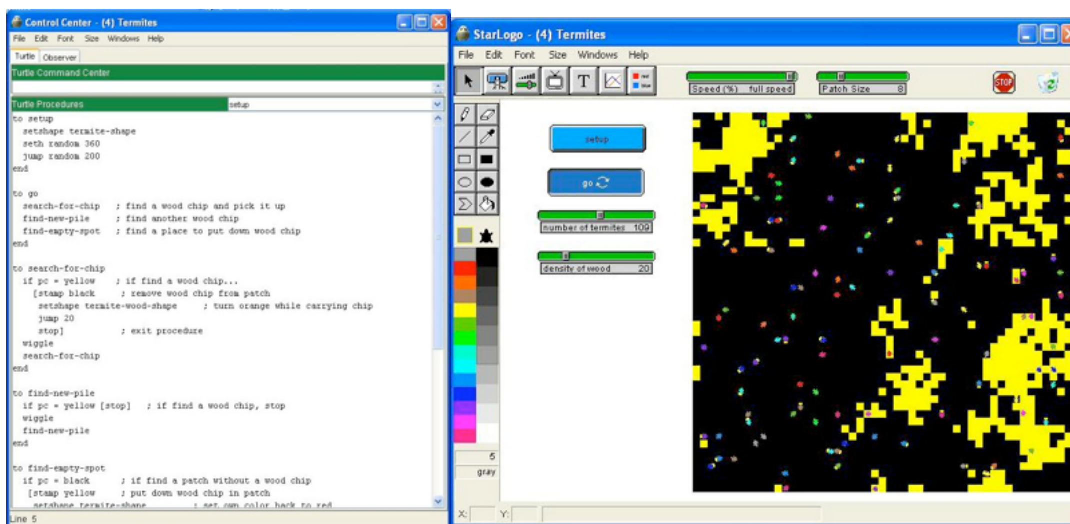

Figure 1: StarLogo 2.2 - Textual code and 2D rendering of agent-based model

At least in the classroom, StarLogo has proven to support the catalytic effect of peers supporting each other’s use of the program through social interaction. The following story exemplifies this phenomenon:

> One boy was fiddling with the slider that controlled the numbers of turtles that were on the screen, which by default, ranged from 1 to 50. At some point, he double-clicked on the slider to see what it would do. He was then presented with a dialog box that controlled the minimum and maximum values for that slider. Being a fifth grade boy, he immediately replaced the seemingly small value of 50 turtles with a new maximum of

> 1,000 turtles. He tried out the new value and quickly proclaimed his finding as "cool" since the new patterns were much different than the old ones with 50 turtles… While this innovation was interesting, it might have taken quite some time for others in the room to make similar changes if each one of them had to independently discover the same mechanism. But the accessibility of StarLogo, and the social atmosphere that it facilitates in the classroom permits and encourages the sharing of information. Within minutes of the boy's discovery of the way to change the slider, nearly half the class had changed their sliders in a similar way [4].

This evidence makes it likely that introducing keyboard usage to StarLogo TNG will enjoy the same benefits when it makes it way into more classrooms, replacing StarLogo 2 as the current version.

6.1.2 LogoBlocks

The other ancestor of StarLogo TNG is LogoBlocks, the graphical programming environment designed to make it easier to create programs for the Programmable Brick [2]. The Programmable Brick ran on a language called BrickLogo also derived from Logo, albeit far simpler than StarLogo, as it did not support procedures or some of the other advanced programming constructs described earlier. Begel's implementation of LogoBlocks sported many innovations in the field of graphical programming environments, and a number of graphical languages besides StarLogoBlocks found inspiration in LogoBlocks, including Bongo, Flogo, Mindstorms, Pet Park Blocks, and Tangible Programming Bricks.

6.1.3 StarLogo TNG

Initial design and development work began on StarLogo TNG in 2002 within the MIT Teacher Education Program. I joined the development team as an undergraduate researcher in the fall of 2002. StarLogo TNG would be a complete rewrite of StarLogo with an eye towards high-performance graphics and a rich, immersive programming and modeling environment (Figure 2). My initial work focused on the new StarLogo virtual machine, this time written in native C code instead of Java. Later, I shifted focus to enhancing the Java-based user interface and developing the block language, joining several other undergraduate researchers. Other students were responsible for crafting the three-dimensional, OpenGL renderer now known as Spaceland.

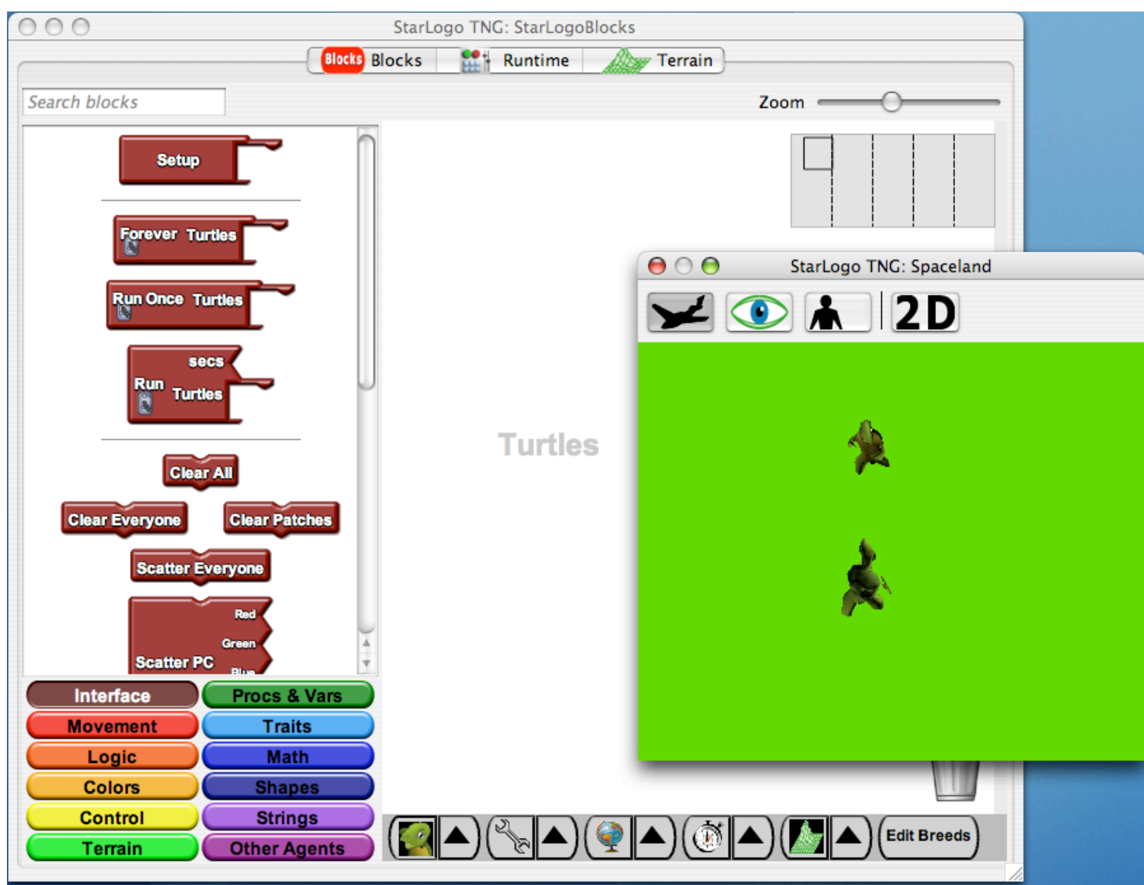

Figure 2: StarLogo TNG - Graphical language and 3D renderer

6.2 Goals

Integrating StarLogoBlocks with a new three-dimensional model renderer, StarLogo TNG espouses the following goals [13]:

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

To elaborate, StarLogo TNG represents a shift away from pure modeling software to a programming environment that makes modeling video games, complex systems in their own right, as easy as modeling biological systems. Appealing to the tastes of students who grew up with Sony PlayStations and Nintendo machines, "We believe that programming should be reintroduced to students, and that this can be done by focusing on video game construction, a compelling subject area for many students [5]."

To satisfy the first goal of making programming easier, the StarLogo language has been overhauled to enrich and simplify the set of primitive commands available to the programmer, as well as to push the graphical programming environment farther than any that has come before.

6.3 Users

StarLogo TNG's target audience is as diverse as its predecessor, StarLogo 2. Students, teachers, and researchers alike will soon use the faster and more capable StarLogo TNG to study and create models. Furthermore, some of the users will be new to StarLogo and programming altogether, while others will have expert experience with the textual language and 2D paradigm of StarLogo 2.

6.3.1 Students

StarLogo TNG is designed first and foremost for secondary students, many of whom have never programmed before but would benefit greatly from exposure to the field of programming. For previous versions of StarLogo 2, students were more frequently consumers of models rather than creators. "While many teachers we work with are successful at developing their own models and using them in the classroom, we have had relatively less success getting teachers to facilitate model construction with their students [14]." The textual language presented a learning curve that was too steep for most students to overcome.

6.3.2 Teachers

Teachers serve as the facilitators, mentors, trainers, co-debuggers, and co-modelers for their students. Since StarLogo TNG is meant to be useful in science classes and not just programming classes, we have to account for the fact that secondary science teachers often do not have formal training as computer programmers either. Therefore, StarLogo TNG must provide an easy programming environment for them as well. So far, we have observed success for both students and their teachers. "Using StarLogo TNG, students and teachers can rapidly develop and understand new programs and create their own 3D world in which to run them [14]."

6.3.3 Researchers

Third, though we will continue to support researchers who prefer the familiar StarLogo 2, which is undergoing preparation to be re-released as an open source project, our expectation is that many researchers will prefer the dynamic, rich, intuitive, and fast environment afforded by StarLogo TNG. Granted, complexity researchers may not be

interested in integrating a joystick-controlled avatar to make games, but they will find a wealth of tools available to them to create scientific models.

6.3.4 StarLogo 2 Users

Finally, StarLogo TNG must be able to support users who transition from StarLogo 2. All of them will appreciate the familiar commands from StarLogo that both versions have in common. Some users will love the new graphical programming environment, while others will not like the dramatic change, at least at first. Spaceland can be configured to give a two-dimensional, top-down view similar to what StarLogo 2 users are accustomed to, and many of the StarLogo 2 sample projects have been ported to StarLogo TNG, along with enhancements that highlight many of the great new features of the environment.

6.4 Spaceland

As the graphical renderer for StarLogo TNG, Spaceland provides the computationally intensive three-dimensional eye candy that contemporary students have come to demand from software intended to be visually appealing. The StarLogo agents live and interact in this world, which can be customized to resemble anything from an enchanted forest to a tropical fish tank (Figure 3).

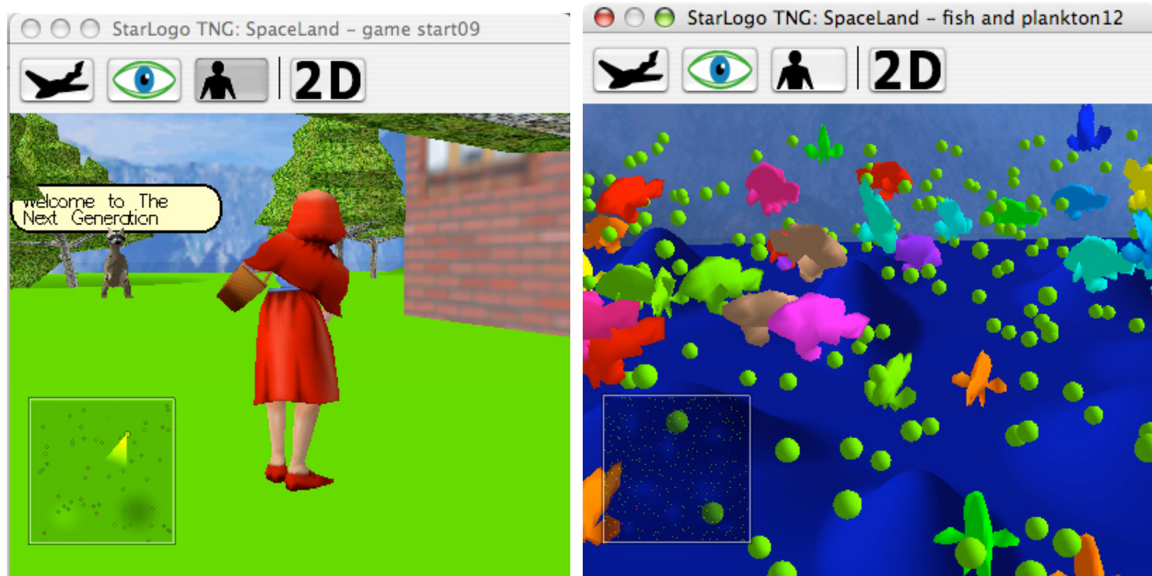

Figure 3: Rich 3D environments in StarLogo TNG [10]

6.5 StarLogoBlocks

6.5.1 Overview

Like other graphical programming environments described so far, StarLogoBlocks uses puzzle piece-like blocks to depict individual commands and the visual syntax they require (Figure 4). Thanks to the power of visual syntax, users can only create syntactically correct programs.

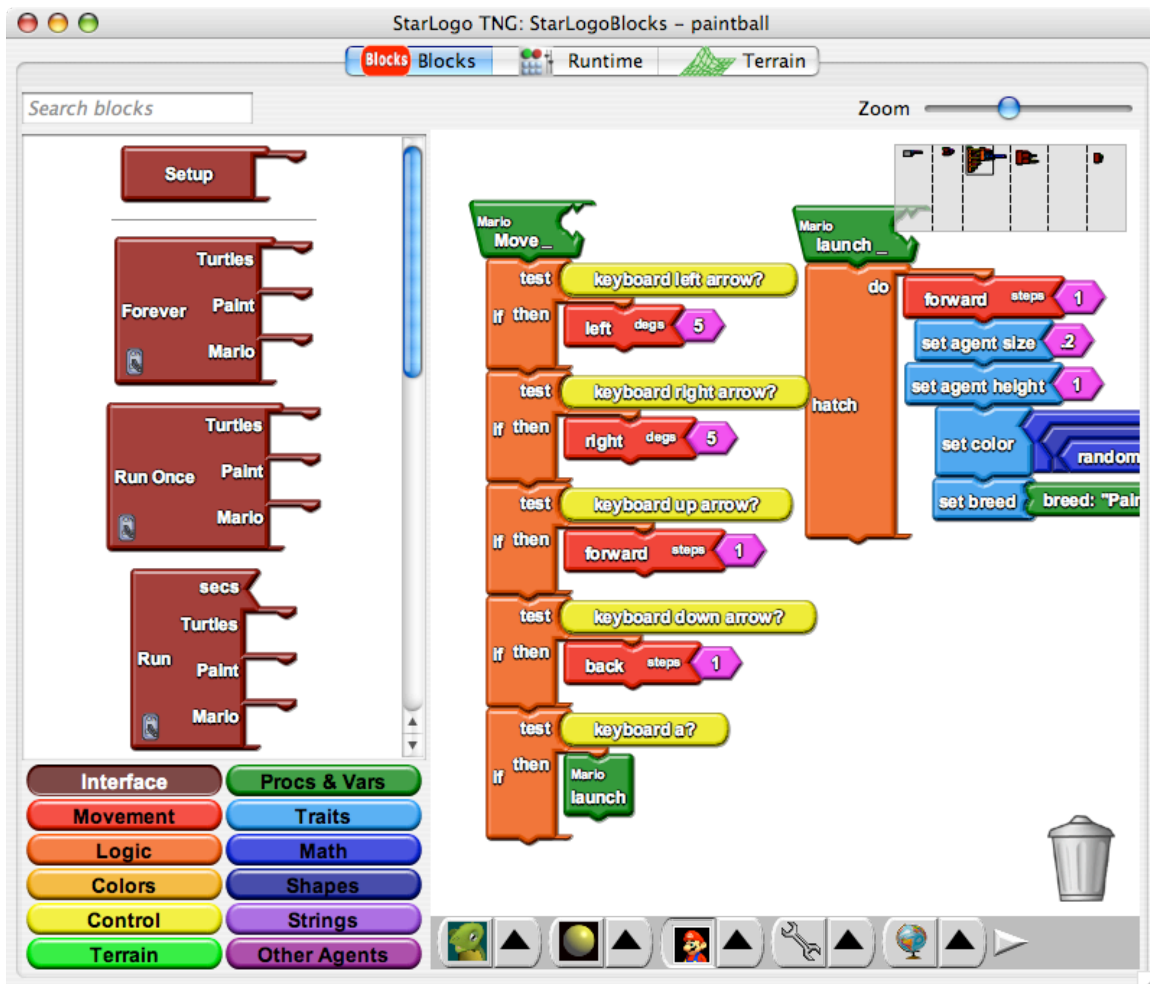

Figure 4: Overview of StarLogoBlocks

Unlike some graphical programming languages, such as Tinkertoy in which some iconic forms represent arbitrarily large sequences of code, "StarLogoBlocks is an instruction-flow language, where each step in the control flow of the program is represented by a block [5]." Users build programs by dragging blocks into the programming workspace from a block factory organized by categories and color-coded in a way reminiscent of syntax highlighting in textual IDEs (Figure 5).

Figure 5: Properties are magenta, and commands are colored by category.

Command blocks link together in a sequence from top to bottom. When a command requires some input, the block has labeled sockets, and the shape of the socket dictates the type of value that may be legally inserted. If the command reports a value, the shape of the plug on the left side of the block dictates the type of the returned value that may be inserted into other commands (Figure 6). Blocks such as “If” and “Repeat” that alter the linear control flow of the block code have special stretching sockets that accept sub-lists of commands (Figure 7). Stacks of blocks may be topped with a procedure declaration block, which allows block stacks to be called and reused elsewhere, as well as allowing for recursion, whereby procedures call themselves (Figure 8).

Figure 6: Triangular port shape only accepts number blocks.

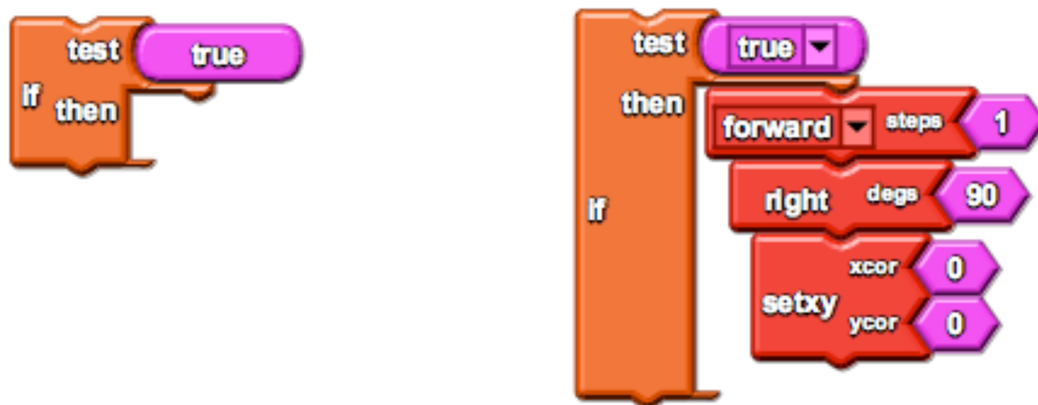

Figure 7: The "If" block stretches to accommodate the blocks inside.

Figure 8: User-defined parameterized procedure and dynamically generated call block

6.5.2 Managing Complexity

Despite its roots in LogoBlocks, StarLogoBlocks presented much greater design challenges "due to the relative complexity of the StarLogo environment. LogoBlocks programs draw from a language of dozens of commands, are typically only 10-20 lines long, have a maximum of two variables, no procedure arguments or return values, and no breeds. StarLogo programs draw from a language with hundreds of commands and can often be a hundred lines or more long… Driven by this challenge, we created a richer blocks environment with new features specifically designed to manage the complexity and size of StarLogo code [5]."

Block Editing

StarLogoBlocks supports many advanced block-editing techniques. Some were adapted from functionality common to textual IDEs, while others take advantage of the unique properties of graphical languages:

- *Undo:*

  Undo and Redo history give users a chance to fix mistakes or revert to a previous state quickly.
- *Block Families:*

  Block Families define groups of related commands such as forward/backward, right/left, and blue/red/white/green/yellow/etc., and blocks that are members of a Block Family have a combo box around the command label that allows the user to swap in a related command from the same Family without having to drag out a new block, disconnect the old one, and drop the new one in (Figure 8).
- *Insertion:*

  If the programmer forgets a command in a stack of blocks, instead of going through the trouble of disconnecting the blocks, adding the missing block, and reconnecting them, the insertion feature allows the programmer to insert a

new command in place, saving a lot of time-consuming clicking-and-dragging.

- *Dynamic Renaming:*

  Similar to Find-and-Replace and other refactoring features common in text editors, if the programmer renames any user-named block such as a procedure, variable, or breed, all of the dependent blocks such as call blocks and variable getters automatically rename themselves to reflect the new name in the declaration.

- *Blockdoc Tool Tips:*

  When the user moves the mouse over a block, either in the factory or the workspace, a tool tip popup appears with "Blockdoc," or Block Documentation, for that block (Figure 10).

Figure 9: Easy substitution for blocks with a family

Figure 10: Blockdoc tool tips help users remember what each block does.

Collapsible Blocks

A common complaint about graphical programming languages is that they take up too much space on the screen. The extra space is mostly taken up by the plug and socket shapes that describe the visual syntax, eliminating the need to type symbols such as square brackets and parentheses. Additionally, when blocks stretch to accommodate other blocks, the resulting block can be very large. However, the space taken up by a stretched block is similar to the indentation levels of textual languages to indicate nested regions of control flow. The difference here is that the stretched blocks delineate code regions more easily than white space.

Nevertheless, it is still true that there is only room for so many blocks on the screen, just as there is only room for so much text on a page. Just as textual IDEs allow code regions such as procedure definitions to be collapsed to reduce the amount of space they take up and eliminate unnecessary distraction from surrounding code that the user wants to focus on, StarLogoBlocks also supports "collapsible" procedures. With a click, procedures collapse or expand to hide or reveal their contents, reducing the number of visible blocks on the screen at any given time (Figure 11).

Figure 11: Procedures collapse to utilize space more efficiently.

Animation

"One of the most important innovations is to incorporate dynamic animated responses to user actions. We use this animation to indicate what kinds of user gestures are proper and improper while the user is performing them [5]." For example, as a user drags a

large stack of blocks toward the command list of an “If” block, the “If” block stretches to accommodate the stack. The animation, which occurs as the user drags the stack towards the “If” block, serves three purposes:

- The real-time feedback confirms to the user that his action will be permitted.
- The stretched block avoids layout issues problematic for other graphical programming languages. LogoBlocks [2] and Tangible Programming Bricks [17] required extra “padding blocks” to prevent blocks from overlapping in unreadable or physically impossible ways. Tinkertoy rendered icons in a large, fanned-out shape connected with tube-like wires that quickly became hard to process for large programs [8].
- The dynamism looks cool, making the environment more visually appealing and fun for the user, young students in particular.

StarLogoBlocks uses animation to provide feedback for other language features as well. For example, “When a user picks up a number block to insert into a list of values (which in Logo may contain values of any type), all block sockets in the list will morph from an amorphous ‘polymorphic’ shape into a triangular shape of a number, to indicate that a number block may be placed in that socket. We plan to continue adding new kinds of animations to help prevent users from making programming errors in the system [4].” Figure 12 shows the equality test block. Since you can test Boolean, numbers, or strings for equality, the block has polymorphic ports that change depending on what type of value the user picks up and tries to insert.

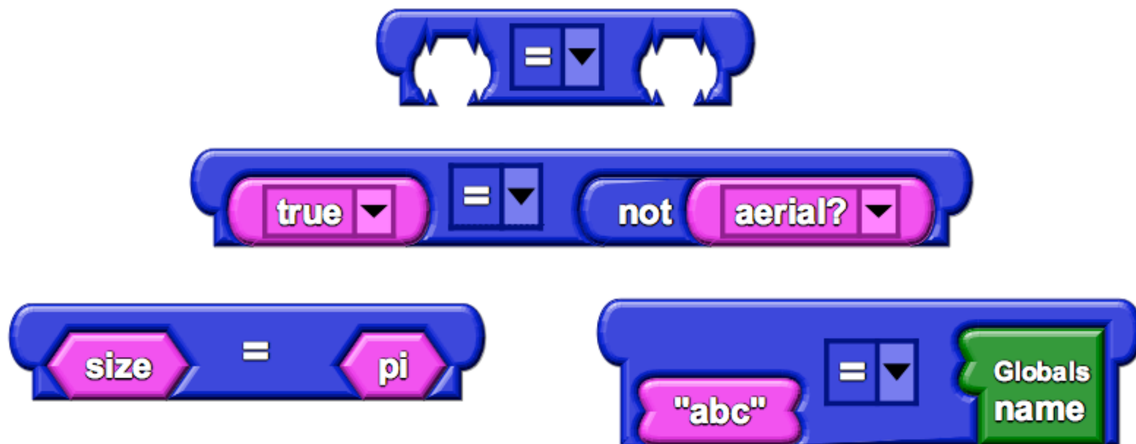

Figure 12: The equality block morphs to contain Boolean, number, and string values.

Organization and Navigation

In order to help the user keep the dozens or hundreds of blocks comprising a program organized, StarLogoBlocks provides several innovative features for promoting block organization and facilitating navigating through the workspace:

- *Pages:*

  The workspace is visually divided into several resizable logical divisions called pages. There is a page for each StarLogo "breed" of agents, a page for global variables and procedures, a page for blocks concerning individual "patches" in the Spaceland terrain, and a page for runtime blocks that describe the user interface for the model. Each page has a drawer that contains blocks specific to the Page to avoid cluttering the standard block factory (Figure 13). Pages are similar to the tabs in a textual IDE to switch among multiple open files from a single project.

- *Zoom:*

  With the zoom slider, the user can adjust his perspective of the entire block workspace easily "to look closely at a procedure they are writing, or expand their view to see an overall picture of the project (Figure 14 and Figure 15) [5]."

- *Minimap:*

  The "Minimap" is a miniature representation of the entire workspace that is analogous to the outline view provided by textual IDEs (Figure 16). Users can jump to any region of the workspace by clicking on it in the Minimap. Additionally, users may drop blocks onto a point in the Minimap to quickly move blocks anywhere within the workspace.

Figure 13: The Breed Bar with the "Turtles" drawer open

Figure 14: Zoom in to focus on a particular stack of blocks.

Figure 15: Zoom out to get an overview of a project.

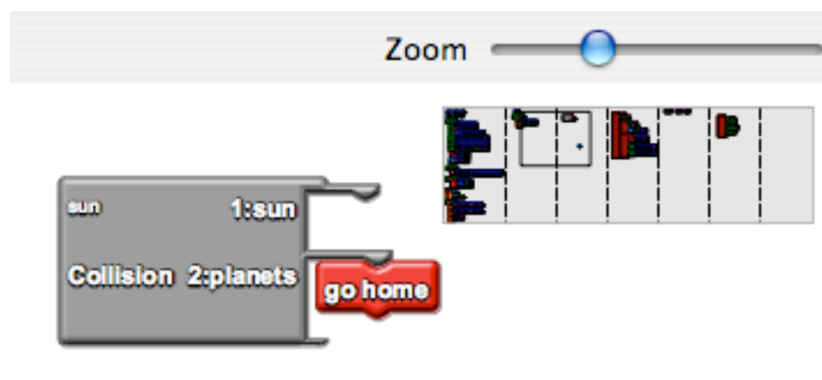

Figure 16: The Minimap provides easy navigation of the block workspace.

6.5.3 Reactions

Preliminary user testing and a public beta release available for download from the web site have given us an opportunity to get feedback about the leap to a graphical programming environment in StarLogo TNG [13]. In particular, since "StarLogo's complex text-based language has always skewed it towards high school students and older," we wanted to know whether younger students could easily pick up StarLogo programming in block form [5].

Among students and teachers in particular, the reaction has been quite positive. "This particular case was with two tenth grade girls (Alice and Beth) who had never programmed before [5]:"

> They strongly preferred the StarLogoBlocks programming paradigm to the text based paradigm of the existing StarLogo. Specifically, they pointed out the way in which you could follow the flow of programs visually, and that you didn't need to worry about syntax.
>
> Alice: It is easier to see the commands too because instead of typing in random things that you don't what they really mean this is like a puzzle piece and you can kind of put it together.
>
> Beth: You can tell if you are doing it right if the puzzle pieces fit too. Because before I was like questioning myself if I was doing it right like bracket or space.

Although the students in the classroom were instantly engaged with constructing programs out of the lively blocks, StarLogoBlocks was not love at first sight for everyone. One review written by an adult expert on programming languages said, "I suspect that this representation of statements as Lego-like pieces with coded connections will work very nicely for children. I can also tell you that programming by dragging blocks from palettes to a design screen with a mouse can turn into a real pain in the shoulder for an adult of a certain age [9]." Another adult user familiar with StarLogo 2 said, "I still think the requirement to build TNG programs graphically, rather than supporting the option of writing (and printing) program code as text, is a mistake; nonetheless, TNG is showing a lot of promise… I don't consider TNG a letdown - though I did, for a while (mostly because of the inability to write models as text) [6]."

Although we have no intention of supporting the ability to print or read block code as text, my hope is that the inclusion of more keyboard shortcuts and the Typeblocking feature will assuage their desire to use a keyboard to type the block code.

7 The Mythical Algebra Block

The Myth of the Algebra Block relates the story of how our team tried to handle the consequences of barring a textual representation of the block code from StarLogo TNG.

7.1 Algebraic Expressions

Prior to releasing the first public beta of StarLogo TNG, we spent a great deal of time playing and working with StarLogoBlocks constructing our own models and sample projects. We really liked it, and we felt very little remorse about excluding a textual language from our design. Private user testing helped confirm our beliefs, as "nearly everyone who has seen the blocks has commented on how much easier it is to see the flow of the programs, and that it relieves the stress of having to remember all of the syntax [5]."

Nevertheless, as the designers revealed in their introductory paper, "We have, however, been asked if it will be possible to 'drop down' to the text level after setting things up. Our answer is 'no'. It is our goal to make it possible to construct sophisticated programs using this paradigm, and we are not treating the blocks as a starting place. Algebraic expressions have been problematic, in that it takes several clicks and drags to write expressions. As a result, we have revised the layout of algebraic expressions to appear less procedural, and will eventually add an algebra-specific mode that will allow basic mathematical expressions to be entered by keyboard and laid out automatically in blocks [5]."

Here, we have conceded the point that algebraic expressions in particular have been problematic because we perceive that it is too much work to click and drag blocks to form mathematical expressions. This point may seem odd, given that one could argue constructing any code requires too much clicking and dragging, as some users have noted. So the first question is, why do we make a special distinction for algebraic expressions? The answer to which is discussed in the next section. The second question is, what is this "algebra-specific mode," and how will it work? The answer is found in the following section within the Myth of the Algebra Block.

7.2 Calculator Syndrome

While experienced programmers may feel frustrated having to click and drag any code at all, most of our novice student users like directly manipulating the blocks with a mouse. However, the one exception, that nearly all users find frustrating, is the composition of algebraic expressions, and I attribute that response to something I call the "Calculator Syndrome."

Even novice students who have never programmed before are very familiar with calculators. Whether it is a basic four-function calculator or a TI-89 programmable graphing calculator, everyone is comfortable with typing out number and operator keys to computer an arithmetic expression. The overwhelming feeling of a blank page does not apply to arithmetic, because the possibilities of input are already limited to ten digits and a few operators. Users do not need to have the blocks or the syntax related to arithmetic expressions laid out for them in a palette.

For most expert calculator users (i.e. nearly everyone), arithmetic blocks feel like an abacus, slow and cumbersome. In blocks, "2 + 12" equals three clicks, three drags, and three keystrokes. First the user clicks on the "Math category." Then the user drags out

the “1” number block, clicks on the block and types “2.” Then the user drags out the “+” block and connects it to the “2” block. Then the user drags out another “1” block, connects it to the right side of the “+” block, clicks on the block and types “12.” Whew, and that was just a simple expression with only one operator. Optimally, users would prefer a mode in which “2 + 12” equals four keystrokes, the same number as button presses on a calculator.

7.3 The Myth of the Algebra Block

Accepting the dilemma of the Calculator Syndrome, we were face with the challenge of providing the ability to type arithmetic expressions without supporting an underlying textual language. The first idea was to provide an algebra block, which would allow the user to type out an arithmetic expression on it. Thus, instead of three blocks “2,” “+,” and “12” all connected, only one block would read “2 + 12.” Unfortunately, the algebra block posed a lot of problems:

- *Inconsistent:*

  Upon seeing the algebra block, some users would be quick to ask why it is all right to allow typing arithmetic expressions without allowing users to type code too. Why not have a “command block” that allowed you to write familiar StarLogo text such as “fd 10 rt 90?”

- *Error Prone:*

  What happens if users type invalid expressions such as “2 + + 2” or “(3 x 5)) + 6?” One could argue that users comfortable enough with math to use the algebra block instead of the standard number blocks are unlikely to make errors such as those, or that they should accept the consequences of syntax errors. Nevertheless, we did not want to open the door to allowing syntactically incorrect expressions. Furthermore, if the algebra block was meant to be fully-featured, then it would be necessary to support other functions that exist as blocks such as absolute value, random, and max, as well

as supporting typing the names of variables, which were allowed to contain spaces and symbols such as arithmetic expressions if the user wanted them to.

- *Incompatible:*

  Finally, and perhaps most importantly, this design for the algebra block is incompatible with the existing math blocks. There was no clearly good way to convert an algebra block to math blocks reliably or vice versa. Therefore, the presence of an algebra block in a model may make the model more difficult to use by a novice, and the presence of math blocks may make the model too frustrating to edit for an expert.

Realizing that allowing users to write textual code, even limited to math on a single block, was a potential disaster for users and a slippery slope for the design. At this point, I suggested an algebra block that does not contain any text. Rather, it enabled a mode such that when it was active, if a user typed the number "3," then a block with the number "3" would appear connected to the algebra block. The user could continue typing an arithmetic expression, watching as the math blocks formed before his eyes. This solution allowed users to type arithmetic expressions as they wanted without succumbing to incorporating textual expressions into the environment. Furthermore, it was completely compatible with math blocks, because the result of typing into an algebra block was the formation of a complete stack of interconnected math blocks, and the visual syntax on the math blocks would protect the user from errors.

Eventually, I came to the realization that this principle of typing out blocks could be applied to any block, not just math blocks, and the design for Typeblocking, as well as the premise of this thesis, was born. Consequently, the fabled algebra block was never implemented, consigned to the dustbin of ideas. Although Typeblocking was not ready for the release of Preview 2, it is slated for inclusion in Preview 3, which is due to be released early this summer.

8 Typeblocking

Typeblocking, as opposed to typewriting, refers to the concept of using the keyboard to construct block code instead of written text. In order for this feature to be useful and feel intuitive, I designed a number of features inspired by the metaphor of a textual IDE.

8.1 Block Cursor

In text editors, the user knows where the next typed character will appear on the screen because a blinking cursor points out the location. When the user types a letter, the cursor conveniently and automatically moves to the next logical position for a character. At the end of a line, the cursor wraps to the next. Deleting a character causes the cursor to move back, and the arrow keys allow the user to move the cursor to any position within the page to perform an insert or overwrite operation.

For Typeblocking to work, the user needs to know where the next block is going to appear, so I created a block cursor. The block cursor is a sort of focus manager. Clicking on a block gives the block focus, and blue highlighting that indicates focus represents the cursor. When a particular block has focus, the next block typed will get connected to the starting block, and then the focus shifts to the new block, effectively moving the cursor to the next logical position in the block stack. The user can quickly undo a typed block by pressing the "delete" key, and the user can move the cursor around the stack of blocks with the arrow keys.[2]

If the user wants to start typing a new stack of blocks, he can simply click on any empty region of the workspace, and the cursor jumps to that point such that any block may be typed and will appear at that point in the workspace without being connected to anything.

---

[2] The "delete" key on an Apple keyboard is the equivalent of "backspace" for Windows and Linux PCs.

If a new block expects a new value or name when it is typed, such as a number block, string block, or procedure or variable declaration, the block automatically enters text edit mode when it appears. If the user wants to change the name later, the "enter" key toggles editing text when the block cursor is hovering over an editable block. Alternatively, if the block with focus is not editable but belongs to a Block Family instead, then pressing "enter" causes the drop-down combo box to open, and the user can select a new member of the Block Family with the arrow keys.

8.2 Workspace Navigation

8.2.1 Search

The ability to search is an indispensable feature of even the simplest text editor, allowing us to find any text in an arbitrarily long document instantly without having to scan the text ourselves. Most graphical programming languages are simple enough that search functionality would not be necessary. However, StarLogoBlocks has many commands in over a dozen categories, and it may be hard for users to find a block without having to spend time searching through each category. Furthermore, StarLogo TNG projects may contain dozens or hundreds of blocks littered throughout the workspace, and it may be useful to find instances of particular blocks without having to perform a visual scan of the workspace manually.

For these reasons, I added an incremental search bar to the StarLogoBlocks window. As the user types part of the name of a block into the search bar, categories containing matching blocks light up with yellow text, and every instance of the block on the workspace gets highlighted on both the main workspace and the Minimap, allowing users to identify the locations of each copy of the block quickly (Figure 17). Users can also jump to the search bar with the keyboard shortcut "Command-F," a common shortcut for search that stands for "Find."

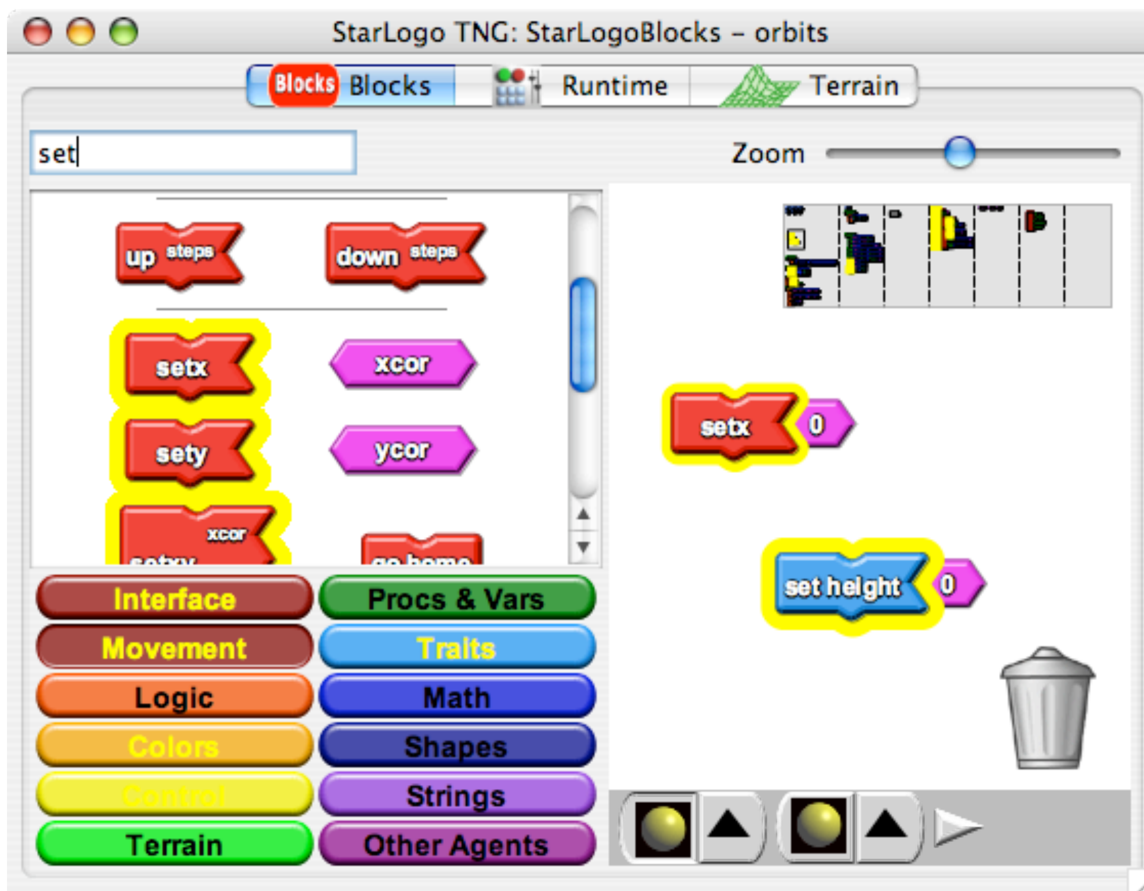

Figure 17: Search reveals blocks and categories that match the query.

In addition to the search bar, I added other keyboard shortcuts to navigate the workspace based on the context of existing block code. When any block has focus, pressing "Command-R" or choosing "Find Related Blocks" from the "Edit" menu performs a search for related blocks. For example, when a variable getter or a procedure call block has focus, pressing "Command-R" highlights the variable or procedure declaration block associated with the starting block, along with other instances of the call block or variable setters and getters. This feature is analogous to the Find Declaration feature common in textual IDEs, and it is particularly handy if the user would like to rename a variable but cannot find the declaration block where renaming is permitted. Similarly, this feature also incorporates the Call Hierarchy feature from textual IDEs. When the block cursor is hovering over a variable or procedure declaration, pressing "Command-R" highlights all

of the variable getters and setters or procedure call blocks associated with the declaration on the workspace, allowing the user to find all references to a particular variable or procedure more easily.

After performing a search, either by manually entering a query or by using the "Find Related Blocks" feature, users can press "Command-G" and "Command-Shift-G" to jump to the next or previous block in the search results respectively. The new block receives focus from the Block Cursor, and the view of the workspace animates smoothly to center the block on the screen to help the user follow the transition.

8.2.2 Zoom

To control the level of zoom within StarLogoBlocks, I borrowed the keyboard shortcut scheme used in many web browsers and Mozilla Firefox in particular. "Command-(plus)" zooms in one notch, "Command-(minus)" zooms out one notch, and "Command-0" returns the workspace to the default level of zoom.

8.3 Context Awareness

One of the requirements of Typeblocking is that it must preserve the visual syntax of the blocks and produce error-free code. Just as a user cannot drop a Boolean block where a number block belongs, the user should not be able to type "true" for the "true" block where a number belongs. In order to achieve this goal, Typeblocking must be aware of its context. When the block cursor is on a "forward" block, there are only two available sockets. The number socket to the right of the block accepts numbers, and the command socket underneath the block accepts other commands. Thus, if the user types a number, the number block should appear connected to the number socket. If the user types a command, the command should appear connected below the "forward" block. Finally, if

the user types a block that is neither a number nor a command, then no block should appear.

If a socket is already filled with a block, then it first tries an insert operation as in the case of a command socket. If an insert would be syntactically incorrect, such as trying to insert a number next to an existing number block in a number socket, then the existing block is overwritten. Though overwrite could potentially destroy some of the user's data, it is easily undone with the undo command.

Some blocks such as "Scatter PC" accept multiple number blocks, such that if a "Scatter PC" block has focus, and the user types a number, the socket where the number block should appear is ambiguous (Figure 18). For this case, I chose a simple, intuitive heuristic to determine where the block should appear. The number block fills the first available empty socket reading from top to bottom. If all of the number sockets are full, then it will overwrite the top number block. If the user needs to change a different number, then he can use the arrow keys to move the cursor to that number block and press "delete" to delete it or "enter" to edit it.

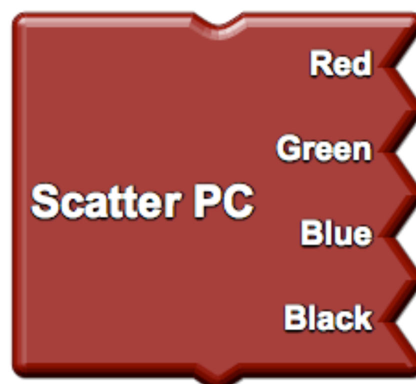

Figure 18: Scatter PC accepts multiple blocks of the same type.

8.4 Auto-Completion

The best textual IDEs implement some form of auto-completion. Combining context awareness with knowledge about the structure of the programming language in use as well as the existing code already written, the IDE will provide a popup box containing a list of valid completions based on where in the code the user is typing and what characters the user has typed so far. As the user continues to spell out the code, possibly using the list box as a reference or reminder of what to type, the list shortens dynamically to include only valid completions after typing (or lengthens after deleting) each character. At any time, the programmer may use the arrow keys to select an item from the list and press the "tab" key or "enter" key to have the editor automatically complete the code selected.

For Typeblocking to be truly useful, users should get the same reminders of valid blocks that they would get if they were perusing the block factory or writing code in a textual IDE. So, when the user begins typing the name of the block, a list appears near where the block will appear that shows all of the valid completions given the position of the cursor, the known StarLogo commands, and the user-defined names of procedures and variables already created.

8.5 Animation

Finally, Typeblocking incorporates animated feedback to help the user understand what is happening as he types out blocks. In text editors, animation is less important because the sudden appearance of a single character that does not affect the layout of the characters around it is not jarring to the user, whereas animating the appearance of the character would probably cause a distraction.

StarLogoBlocks is a different story altogether. Blocks can be very large and take up a significant amount of space on the workspace. Furthermore, the size of the surrounding blocks may change when larger blocks are connected to them. If the new block suddenly snapped into place, the lack of animation could be confusing, and the confusion unto itself is a distraction. These are the instances where careful use of animation contributes to sustaining flow: "Another approach that can potentially be used to reduce users' sense of interruption is to use animation between screen states. While animation in general can be used in ways that are very disruptive, we have found that animation can be helpful if used to help users understand how the interface changes [1]." Specifically, "The potential for this kind of animated transition is that it can reduce the cognitive overhead of understanding of the relationship between two screen states, thus enabling uses to stay focused on the task [1]."

I had several potential designs for animating the appearance of a new block. The first took advantage of existing transparency features of the block graphics. By gradually increasing the alpha channel of the block's visual representation, the block would appear to fade into view, changing quickly from something ethereal to a more solid block. The second idea utilized the vector graphics used for the blocks that allow them to be zoomed easily without distortion. I imagined that the block would at first appear very small and quickly scale up to match the size of the surrounding blocks. Another idea was to have the blocks quickly fly in from the side of the screen, coming to a stop when they arrived at their final destination. The block would appear near the factory and move, dragged by an invisible force, until it snapped into place where it belonged.

Ultimately, I chose the fly-in approach for a couple reasons. First, it was most consistent with the other mode for creating blocks—namely, dragging. If a block flew out from its starting position in the block factory to arrive at its final position on the workspace, the animation would look as though StarLogoBlocks responded to the user's key press by dragging the block out of the factory for the user. Second, it seemed to have the coolest

effect. Realizing that the best way to encourage a user to adopt a particular feature is to make it seem cool, the visual effect of blocks flying onto the workspace as the user types seemed to be the most visually appealing.

In order to ensure the animation always takes the same amount of time regardless of the speed of the user's computer, I used path and pacing functions in conjunction with a thread-safe timer to send updates to the GUI's render queue [18]. For the path function, I chose a simple linear transformation that renders the block along a straight-line path from the starting position to the ending position. The pacing function, given the current time and the total amount of time allocated to the animation, determines how far along the path the block should be rendered at each point in time. I chose a sigmoid, slow-in/slow-out pacing function that gives the block the appearance of revving up as it leaves the factory and slowing down as it reaches its final position. For each frame, the following pseudo-code computes the block's next position:

```
// Apply a sigmoid pacing transformation:
s = (atan(current_time * PI / TOTAL_TIME - PI/2) + 1) / 2
// Apply a linear path transformation:
next_x = (1-s) * initial_x + s * final_x;
next_y = (1-s) * initial_y + s * final_y;
```

Unfortunately, introducing animation comes at a cost. According to the flow study, "The tradeoff is that animations fundamentally require time. So a question is whether the time spent on such animations is well spent [1]." We believe that the time is indeed well spent, but it remains to be seen whether our users will appreciate the effect when it is introduced in Preview 3.

8.6 Typing Modes

A common type of error for users is called a "mode error." Mode errors result when the same action produces different effects under different circumstances, and the user either

momentarily forgets which circumstances he is in, or the interface presents an ambiguous state that makes it unclear which circumstances apply.

For example, consider the shape of a typical text editor cursor. The cursor is a straight blinking bar for normal text, but when the user activates *italics* mode, the cursor appears slanted, just like italicized text. However, when the user enables **bold** mode, the cursor does not become bold. It looks exactly the way it does when bold style is not enabled. Usually, the easiest way for a user to determine which mode is active is to type a character and see whether it comes out bold. If the interface provided a clear and obvious indicator of which mode the user is in, such as by making the cursor appear bold, then the user could avoid many such mode errors.

StarLogoBlocks has the same potential problem, because keyboard input can mean several different things to StarLogo TNG. In one mode, the keyboard keys control the camera in Spaceland. The arrow keys rotate and translate the scene, and several other keys represent shortcuts to toggle various views and modes in Spaceland. A second mode already exists for StarLogo TNG, which allows models to detect and respond to keyboard input, such that pressing any letter, number, arrow key, or spacebar could affect the running model. Even with just two modes, many users have expressed confusion trying to understand how to differentiate and toggle between the two modes [16].

Now, Typeblocking introduces a third mode for keyboard input to StarLogo TNG. The keyboard shortcuts for searching and zooming, for example, do not causes conflicts because none of the Spaceland shortcuts overlap, and StarLogo TNG currently does not support detecting and responding to Command, Option, or Shift key presses within models. However, the ability to type blocks by spelling out their labels could easily lead to conflicts with both model keyboard input mode and camera control mode. Thus, it becomes even more important to make it clear which mode is enabled and when.

There are two kinds of modes: permanent and spring-loaded [18]. A spring-loaded mode is a transient mode typically enabled by holding down a key or mouse button. For example, typing a letter key in a text editor normally results in a lowercase letter getting printed to the screen; however, when the user holds down the "Shift" key, the CAPS mode is spring-loaded, such that keys typed while "Shift" is held down results in capital letters getting displayed. Toggling CAPS mode with the "Shift" key is different from toggling the mode with the "caps lock" key, which permanently changes the mode until the key is pressed again. Spring-loaded modes are ideal because they require a more conscious effort on the part of the user to enable them, which makes them less likely to lead to mode errors, whereas the set-and-forget nature of the "caps lock" key makes mode errors more likely.

Unfortunately, none of the modes described above would be suitable as spring-loaded modes. There are many use cases where it is convenient to be able to enter many key commands in rapid succession for controlling models, the camera, or blocks, and it would be inconvenient to have to hold down a modifier such as "Shift" the entire time. Since each mode is a permanent mode, there are two ways the interface can improve their usability. First, the modes should be mutually exclusive. It is rare that a user would benefit from being able to use the keyboard to type blocks, move the camera, and control a model all at the same time. Second, the modes should be clearly visible and easily toggled via toolbar toggle buttons, checkmark menu items, and keyboard shortcuts. I added the keyboard shortcut "Command-K" cycles through each of the three modes.

9 Future Work

With Typeblocking and other keyboard shortcuts slated for inclusion in public beta Preview 3, the next step is to gather user feedback on the usefulness of these features and iterate on the design. In addition to improving upon the features already implemented, these features lay the groundwork for other potential improvements to StarLogoBlocks.

For example, it might be useful to allow users to define custom keyboard shortcuts and macros.

9.1 Alternative Input Modalities

This thesis focused on keyboard input to facilitate fast code entry and efficient use for expert users in a way that does not alienate novice users, but there are many other input modalities besides mice and keyboards that are worth exploring. A few of them are described in this section.

9.1.1 Speech Recognition

As technology improves, software solutions for speech recognition are also improving at a steady pace. For writing text, products such as IBM ViaVoice and Dragon Naturally Speaking provide usable speech recognition for composing arbitrary text. Additionally, many companies employ increasingly intelligent voice response systems that utilize speaker-independent voice recognition for a limited but robust grammar to give callers more control over their interactive sessions. Voice recognition also enables hands-free use of cell phones and other portable devices.

Graphical programming environments such as StarLogoBlocks may be ideal for coding with speech recognition because the relatively simple languages result in relatively simple grammars for the speech recognition engine to process a user's utterances. Begel's Ph.D. thesis explored the use of speech recognition to program in textual languages, and I think there is a great opportunity for one of the lead designers of StarLogo to apply his results to graphical programming environments and StarLogoBlocks in particular [3].

9.1.2 Gesture Recognition

Another class of input is known as gesture recognition, and it comes in several flavors. The simplest and most common variety is responding to touch-screen input. As Tablet PCs and Ultra-Mobile PCs (UMPCs) become more widespread, users may find that a stylus or finger is a much more comfortable and natural tool for composing block code than either a mouse or keyboard.

Another form of gesture recognition that takes place on a touch-screen or with a mouse is the interpretation of symbols traced out with a pointing device. For example, tracing a square on the screen in a counter-clockwise motion is a gesture shortcut to trigger the Undo operation, whereas tracing a square in a clockwise motion would trigger the Redo operation. On touch-screens in particular, gestures may be faster to execute than a menu item or pressing buttons in an on-screen keyboard.

A specialized form of touch-screen gesture recognition is handwriting recognition, where the gestures are meant to resemble natural handwriting that trigger the equivalent key press for the letter drawn. Graphical programming languages could be “Writeblocked,” causing blocks to appear as a result of spelling out the labels of the blocks with gestures.

Taken to the extreme, some immersive systems such as the upcoming Nintendo Wii utilize a three-dimensional motion-sensing device to provide input to the system. A clever designer may be able to dream up interesting UI interactions for block programming using a similar device.

9.2 Accessibility

As a corollary to exploring alternative input modalities, graphical programming environments could benefit from the application of techniques to make the software

accessible to users with sensory or motor impairments. Commercial operating systems have built in support for enabling accessibility features in textual applications. Users with motor impairments can slow down the keyboard repeat rate, decrease the threshold for detecting a double click, enable “caps lock” style modes for other modifier keys that are typically spring-loaded, and more. For users with hearing impairments, users can turn up the volume or enable screen flashes to replace the system alert sound. Vision impaired users can enable automatic zooming of whatever is under the mouse pointer, and they can employ text-to-speech (TTS) screen readers to help them understand what is displayed on the screen. Finally, sufferers of repetitive stress injuries (RSIs) benefit greatly from speech recognition technology supplanting their use of the keyboard.

9.3 Collaborative Programming

Now that most computers, especially classroom workstations, enjoy a fast connection to the Internet or a Local Area Network (LAN), there are more opportunities to use software supporting collaborative writing and collaborative programming. However, to my knowledge, there are no existing collaborative graphical programming environments, which could have as much potential as written ones, allowing pairs of students to work together on the same project from two different computers, supporting each other and helping each other learn without one of them having to take a backseat to the controls. One interesting twist to collaborative programming as it applies to educational software is that it might need to incorporate more protections against a student who may lack the maturity to respect the work of his partner and cooperate effectively.

10 Conclusion

Fundamentally, there is no single environment—textual or graphical or something in between—that will satisfy every user. Nevertheless, I have proposed a compromise that

has the potential to satisfy most novices and experts alike with a graphical programming language that adapts as many advanced features from mature textual IDEs as possible and supports a mode for creating graphical code using the keyboard. Unfortunately, textual languages have other benefits besides keyboard entry and fancy search features. Ultimately, block code is not a perfect substitute for text, and it does not help users who will always prefer to read, print, or share their programs as text.

In spite of the disadvantages of my solution, given the relative merits of textual and graphical languages, the evidence I have presented mostly favors graphical programming languages for educational environments. Since StarLogo TNG is primarily educational software, I believe that the exclusion of a textual language in favor of a robust graphical language is beneficial for the students. My proposal strives to unify the mouse and keyboard for graphical programming, performing text processing without the text. And if I am correct that the features I have described make StarLogo TNG more enjoyable for some users without detracting from the educational experience of those who appreciate the block programming experience, then I consider this endeavor to be a success.

References

1. B. Bederson, "Interfaces for Staying in the Flow," *Ubiquity*, 5(27), 2004.

2. A. Begel, "LogoBlocks: A Graphical Programming Language for Interacting with the World," advanced undergraduate project, Media Laboratory, Massachusetts Inst. of Technology, 1996.

3. A. Begel, "Spoken Language Support for Software Development,"doctoral dissertation, Graduate Division, Univ. of California, Berkeley, 2005.

4. A. Begel and E. Klopfer, "StarLogo: A Programmable Complex Systems Modeling Environment for Students and Teachers," *Artificial Life Models in Software*, A. Adammatzky and M. Komosinski, eds., Springer-Verlag: 187-209, 2005.

5. A. Begel and E. Klopfer, "StarLogo TNG: An Introduction to Game Development," *Journal of E-Learning*, in press, 2005.

6. N. Bennett, "New Mexico Supercomputing Challenge," Jan. 2006; https://mode.lanl.k12.nm.us/forum/viewtopic.php?p=93.

7. V. Colella, E. Klopfer, and M. Resnick, *Adventures in Modeling: Exploring Complex, Dynamic Systems with StarLogo*, Teachers College Press, 2001.

8. M. Edel, "The Tinkertoy Graphical Programming Environment," *IEEE Transactions on Software Engineering*, 14(8): 1110-1115, 1988.

9. M. Heller, "StarLogo TNG," *Byte.com*, 6 Mar. 2006; http://www.byte.com/documents/s=9957/byt1141336941474/0306_mh.htm.

10. C. Kelleher, D. Cosgrove, C. Forlines, J. Pratt, and R. Pausch, Alice2: Programming without Syntax Errors, *Proc. 15th ACM Symp. User Interface Software and Technology* (UIST 02), ACM Press, 2002.

11. C. Kelleher, R. Pausch, "Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers," *ACM Computing Surveys*, 37(2): 83-137, 2005.

12. E. Klopfer, "StarLogo on the Web," May 2006; http://education.mit.edu/starlogo/.

13. E. Klopfer, "StarLogo TNG: Welcome to the Next Generation," May 2006; http://education.mit.edu/starlogo-tng/.

14. E. Klopfer and S. Yoon, “Developing Games and Simulations for Today and Tomorrow’s Tech Savvy Youth,” *TechTrends*, 49(3): 33-41, 2005.

15. D. Lane, H. Napier, S. Peres, and A. Sándor, “Hidden Costs of Graphical User Interfaces: Failure to Make the Transition from Menus and Icon Toolbars to Keyboard Shortcuts,” *International Journal of Human-Computer Interaction*, 18(2): 133-144, 2005.

16. C. McCaffrey, “StarLogo TNG – Free Agent,” May 2006; http://education.mit.edu/starlogo-tng/blog/.

17. T. McNerney, “Tangible Programming Bricks: An approach to making programming accessible to everyone,”master’s thesis, School of Architecture and Planning, Massachusetts Inst. of Technology, 2000.

18. R. Miller, “6.831 User Interface Design and Implementation,” Dec. 2005; http://groups.csail.mit.edu/uid/6.831/.

19. S. Peres, F. Tamborello, M. Fleetwood, P. Chung, D. Paige-Smith, “Keyboard Shortcut Usage: The Roles of Social Factors and Computer Experience,” *Human Factors and Ergonomics Society 48th Annual Meeting*, 48: 803-807, 2004.

20. K. Wang, C. McCaffrey, D. Wendel, and E. Klopfer, “3D Game Design with Programming Blocks in StarLogo TNG,” *7th International Conference of the Learning Sciences* (ICLS 06), in press, 2006.