OpenBlocks: An Extendable Framework for Graphical Block Programming Systems

by

Ricarose Vallarta Roque

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2007

Author .....
Department of Electrical Engineering and Computer Science
May 25, 2007

Certified by .....
Eric Klopfer
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

OpenBlocks: An Extendable Framework for Graphical Block Programming Systems

by

Ricarose Vallarta Roque

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2007, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

Graphical programming systems have been built to lower the threshold to programming for beginners. However, because these systems were designed to make programming more accessible to novices, they were developed with narrower intentions for their users and applications. For example, in StarLogo TNG, a graphical block programming environment, users may create games and simulations, but they cannot use this same system to create programs that can automate their computer processes, like the text-based scripting system AppleScript. Application developers can create their own programming systems, but doing so can take a significant amount of time to design and implement. This thesis describes an extendable framework called OpenBlocks that enables application developers to build and iterate their own graphical block programming systems by specifying a single XML file. Application developers can focus more on the design of their systems instead of on the details of implementation. The design and implementation of OpenBlocks are described, along with a user study conducted to test its usability and extendability.

Thesis Supervisor: Eric Klopfer
Title: Associate Professor

Acknowledgments

I would first like to thank my advisor Eric Klopfer for his tireless support, endless patience, and for inviting me to join his team, where I have gained invaluable experience and met people who are more my friends than my teammates. His passion for the StarLogo TNG project is contagious.

Thanks to Daniel Wendel and Corey McCaffrey, who had both already graduated and moved onto a full-time jobs, still joined me for weekend hacking sessions and many design discussions through email and phone. They have both acted as mentors for me, helping me get started and guiding me through my time here in the group. Brett Warne made the code blocks as beautiful as they are today. Brett and Daniel made this such a fun experience for me as well. Eric Rosenbaum has been a valuable source of advice, acting as a sounding board for my various concerns and ideas in design. Thanks also to the crucial contributions of my other teammates: John Jackman, David Greenberg, Thomas Robinson, Mark Burroughs, and John Pope.

Thanks to the support of the EvoBeaker team, who's interest in integrating blocks into their application inspired the OpenBlocks framework. I am excited for their future students to learn and delve into block programming.

Thanks to Tammy Stern for always giving me the strength to keep going, even during my lowest of lows. I feel so lucky to have her as my friend and confidant. Brian Keegan proofread this thesis and his unending patience, love, and encouragement helped me through this stressful process by keeping me sane and grounded. I can't imagine my MIT experience without either of them.

And finally, I'd like to express my gratitude to my family for their sacrifices, support, and love. I owe them all the success and happiness in my life.

Contents

List of Figures

Chapter 1

Introduction

Learning to program can be a difficult task. In addition to learning confusing and non-intuitive syntax, beginners must learn how to structure their thinking and solutions and understand the execution of their programs [1]. However, once these obstacles are overcome, programming can be a very useful and beneficial skill. Advantages include but are not limited to [1]:

- Learning how to solve problems in a structured and logical way.
- Building software customized for personal use.
- Exploring subject areas such as studying biological simulations or creating interactive animations.

In education, computer programming can be "an ideal medium for students to express creativity, develop new ideas, learn how to communicate their ideas, and collaborate with others" [2].

Many graphical programming systems exist to break down the barriers to computer programming, helping to reduce the threshold for beginners to learn how to program and reap the benefits of programming. Instead of working exclusively with text and recalling language syntax and rules, many graphical programming systems have users manipulate and interact with visual objects or images to build their programming projects. For example, StarLogo TNG is a graphical programming environment for secondary students and teachers to study and create 3D simulations and games [3]. This system only requires users to connect puzzle-piece like objects called blocks of varying shapes and colors to build their programs,

as shown in Figure 1-1. Users drag blocks from a palette of blocks to the side called the block factory to the main work area or the block canvas. Blocks are labeled with the program's commands and users can only connect blocks of complementary shapes. Thus, the syntax of this language is visually apparent and encoded in the shapes of the blocks, relieving the work and stress of learning and recalling confusing syntax.

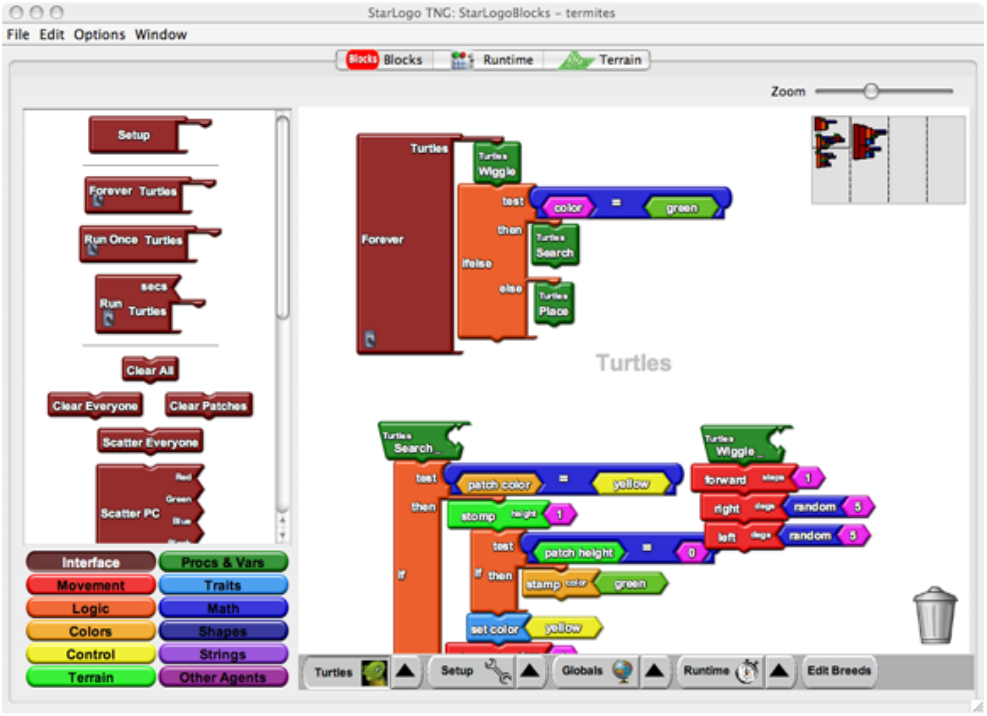


Figure 1-1: StarLogo TNG's block programming environment.

Some graphical programming systems have even eliminated text altogether in their languages and allow their users to manipulate images instead. In StageCast, users manipulate images of agents or avatars in a 2D-grid-like environment [4]. To specify how one of these agents should behave, users create before and after states for each agent. For example, if the user wants a girl to step on top of a vase, the user must specify the before state by creating an image with the girl next to the vase, and then, specify the after state by creating an image with the girl on top of the vase, as shown in Figure 1-2.

However, because graphical programming systems were built to make programming more accessible to novice users, they were designed with narrower intentions for their users and applications. This limited focus restricts the possibilities that users can do with most graphical programming systems. For example, in StarLogo TNG, users may create games

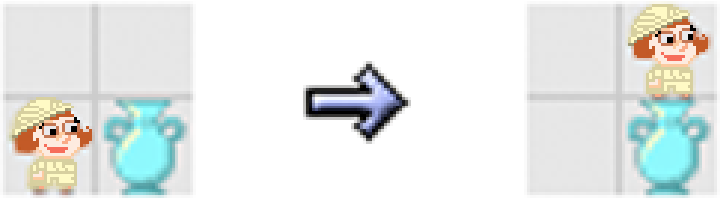


Figure 1-2: Before and after states of a girl agent stepping over a vase in StageCast

and simulations that they control through block programming, but they cannot use this same system to create programs that can automate their computer processes, like the text-based scripting system AppleScript [5]. And while users can attempt to use StarLogo TNG to perform matrix operations, it may not be the appropriate environment. The resulting project in TNG would be messy and complex compared to the more elegant implementation in numerical computing environments such as Matlab [6]. Finally, because commands in TNG are represented as text labels on block shapes, StarLogo TNG may be difficult for younger audiences who are still learning to read.

Thus, if application developers want to add a graphical programming component into their applications, they must build their own system to match their specific goals, end-user needs, and tasks. However, building a graphical programing system can take a significant amount of time to design and implement. When designing their systems, developers must choose a language that maps closely to the natural language of their end-users and has the right level of abstraction for their users to understand the language. Solving these two problems may require extensive design iterations. Implementing the system would also take a substantial amount of time and effort, and so can modifying the system after each design evaluation.

1.1 OpenBlocks Framework

This thesis introduces OpenBlocks, an extendable framework to create block programming systems like StarLogo TNG. By specifying a single XML file called the Language Definition file, application developers can have a running block programming system. Further iterations of the designed system can be done by modifying the same XML file, shortening the time to loop through each design iteration of the programming system.

OpenBlocks actually inherits many of the generic and user-tested block programming features of StarLogo TNG, such as the graphical blocks and the programming environment to contain and manipulate those blocks. Other basic programming system features have already been implemented, such as dragging and dropping blocks, connecting blocks to form stacks, undoing and redoing user actions, and saving and loading projects. OpenBlocks, however, does not have a pre-built block compiler or interpreter, as such a module depends on the block language.

OpenBlocks also implements many StarLogo TNG features that help users manage the complexity of their projects and tools to make them more efficient block programmers. For example, one criticism of graphical programming environments is how visual objects such as blocks can rapidly soak up screen real estate as projects grow. StarLogo TNG implemented a zooming feature to help users manage their work space. It also implemented a code folding feature, where a stack of blocks can "collapse" or hide its set of blocks, leaving only one block to represent the stack, as shown in Figure 1-3.

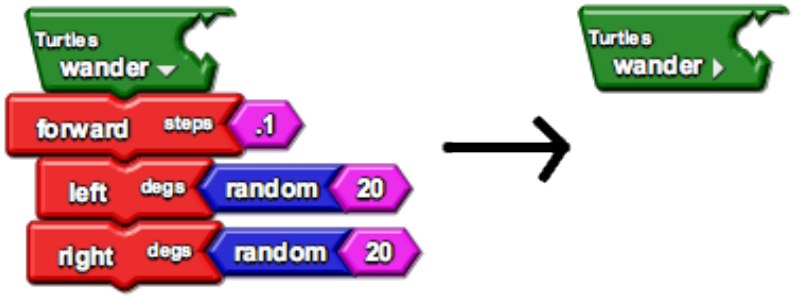


Figure 1-3: Code folding can help save space in a crowded workspace.

This framework also includes features that further improve upon StarLogo TNG's feature set. For example, OpenBlocks adds eight more block connector shapes to the six that StarLogo TNG had. OpenBlocks also enables developers to create their own custom block shapes.

User tests of StarLogo TNG revealed some usability problems that OpenBlocks addresses. In one set of tests, when we asked users to remove the blocks they had dropped onto the block canvas, many users immediately dragged the blocks back to the block factory to "bring back" or "return" blocks to where they came from. However, StarLogo TNG only allowed users to remove blocks by dragging them to a trash can icon at the bottom right corner of the canvas. OpenBlocks offers more ways to remove a block, one of them being

the option to drag blocks back to the block factory.

Finally, the user interface configuration of StarLogo TNG may not apply to all possible applications of OpenBlocks. Thus, this framework enables application developers to select which user interface elements to include in their programming environment. For example, in StarLogo TNG, the block factory is static. In OpenBlocks, however, application developers can configure their block factory to be a floating window over the main work area, as shown in Figure 1-4.

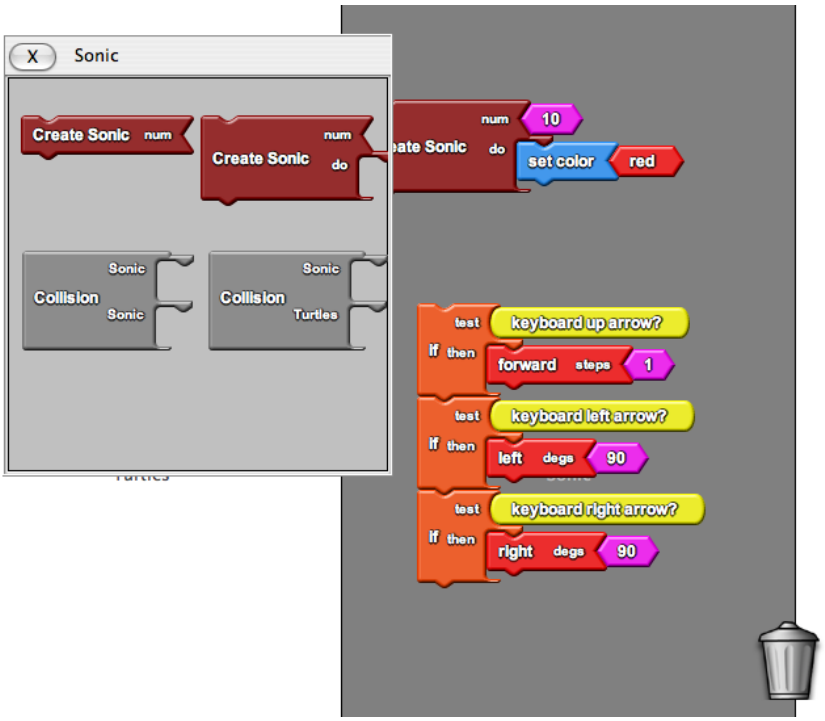


Figure 1-4: A block factory can be a floating window over the block canvas.

1.2 Thesis Summary

Chapter 2 offers an overview of existing programming systems that use textual, graphical, or a mix of both types of language representations. Chapter 3 discusses StarLogo TNG, from which OpenBlocks derives many of its features. Chapter 4 presents the design of OpenBlocks, starting with an enumeration of the goals and requirements of the system, and then describes the user interface design of OpenBlocks. Chapter 5 explains how to extend the OpenBlocks framework to design and develop a programming system. By performing a case study, Chapter 7 assesses the usability of the OpenBlocks framework from the perspec-

tive of application developers and language designers. Finally, Chapter 8 recapitulates the contributions of this thesis and discusses future extensions and research on this framework.

Chapter 2

Background

This chapter describes previous work on various programming environments. Section 2.1 describes textual programming environments and accesses some of their features and uses. Section 2.2 explores different types of graphical programming environments including block programming environments.

2.1 Text-Based Programming Environments

Text-based programming environments can be as simple as a text editor or sophisticated as a integrated development environment (IDE). IDEs may include some or all of the following: a source code editor, a compiler or interpreter, build tools, team synchronization tools, and a debugger. Tools within the IDE and its user interface features are meant to help developers program and manage their software projects. Some IDEs come equipped with GUI builders so developers can rapidly create the user interface for their applications. In general these features help developers cope with some of the difficulties of programming: learning and remembering syntax, understanding the execution of their code, debugging unexpected behaviors, and managing the complexity of their projects.

2.1.1 Assistive Features for Text-Based Programming

Text or source code editors are equipped with several features to help developers to become more efficient, to catch and prevent syntax errors, and to manage the complexity of their code. These assistive features do not necessarily lower the barrier to entry for novice users, but act as guides and tools for current developers. Some of these common assistive features

are:

- Syntax highlighting: Significant keywords are rendered in different colors and fonts, as shown in Figure 2-1. This feature helps to improve the readability and context of code and help catch syntax errors.

```
public void changeGenusTo(String genusNam
    System.out.println("changing genus");
    this.genusName = genusName;
    label = BlockGenus.getGenusWithName(g
    //return new Block(genusName);
}
```

Figure 2-1: Keywords are rendered in different colors.

- Auto-format: This feature automatically formats source code to standard styles. Some editors allow developers to specify the type of formatting to enforce.

- Outline view for classes, methods, fields: This feature can show the overall structure of project.

- Re-factoring tools: When renaming variables or redefining method signatures, refactoring tools will trace calls to these fields and methods and change them accordingly.

- Code completion: The text editor makes suggestions to the developer based on the current text input. In Microsoft's InteliSense [7], a popup becomes visible showing the possible completions to the current text input, as shown in Figure 2-2.

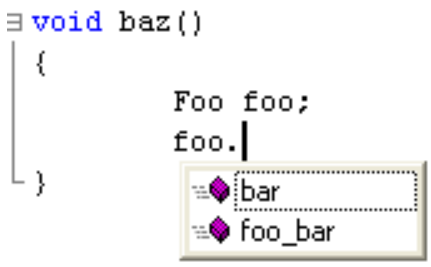


Figure 2-2: The Intellisense in Visual C++ popup shows the developer the possible completions of the current text input.

- Code-folding: Developers can selectively hide or show defined regions of their code. Such folding helps to manage large projects, as developers can hide large sections of code and only show relevant regions instead.

- Code documentation and commenting: Text-based languages include special characters or character sequences that enable developers to write comments alongside their code. The corresponding compiler or interpreter of the language will ignore strings that follow these special characters. In addition to ignoring the characters '//' as comments, Eclipse will automatically generate web pages featuring the Java documentation encoded in sections enclosed in '/** */' [8].

2.1.2 Debuggers

Debugging mechanisms are also available in some IDEs to help developers better understand the execution of their code. Eclipse modifies the layout of its programming environment to display a "debugging perspective," where developers can step through their source code, examine the state of objects, and explore the stack trace at specified breakpoints [8].

2.1.3 GUI Builders

Some IDEs provide Graphical User Interface (GUI) builders that enable developers to rapidly build their application's user interface or the entire application through direct manipulation of GUI elements. In Microsoft's Visual Basic [9], programmers can manipulate forms and form objects such as text fields, buttons, and labels. Programmers specify attributes and actions for each of these objects and may write lines of code to add more functionality. Visual Basic has the compiler within the GUI building environment and so application developers may immediately have a running application after manipulating their GUI elements without writing large sections of code.

2.2 Graphical Programming Environments

Graphical programming languages and their associated programming environments grew from motivations to lower the barriers to programming. Text-based programming languages, while very powerful, can still be difficult to learn for beginners. Not only must they learn to form solutions in a structured way and understand how a computer executes their code, they must also learn confusing and inflexible syntax and commands [1].

2.2.1 Node-based Visual Programming

In node-based programming, users manipulate and link *nodes* that each perform specific operations and return output based on the node's inputs. The links between the nodes represent the flow of data from one node to another. The resulting structure looks like a directed graph that provides users with a visual overview of the data and program flow. And by inspecting the data received and returned at each node, users are also provided with the before and after states of their data at that node. For example, Quartz Composer is a node-based programming language that can process and render graphical data [10]. If a user wants to render an image with a gaussian blur, the user links the node representing the image to a node that performs Guassian blurs as shown in Figure 2-3. Users can also group a collection of nodes within Quartz Composer to build a macro node that represents the overall operation of the nodes it comprises.

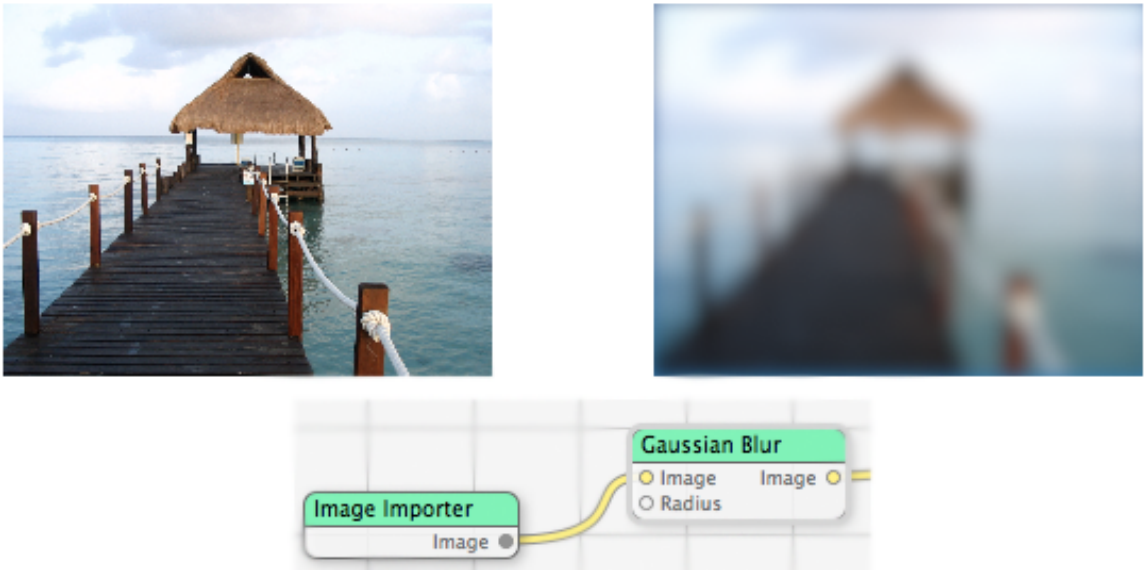


Figure 2-3: By connecting the Image Importer node to the Gaussian Blur node, a Gaussian blur transformation is applied to the image.

Node-based programming languages are also useful in visualizing parallel processes, as programs do not flow from a single point, but instead can begin from several points, making that data proceed through an asynchronous network of nodes.

However, node-based programming languages do not scale well as projects grow and become more complex. Nodes can take up screen real estate. Links can become messy and hard to follow. Figure 2-4 a more complex project in Quartz Composer that renders the

Photo screen saver on Macs.

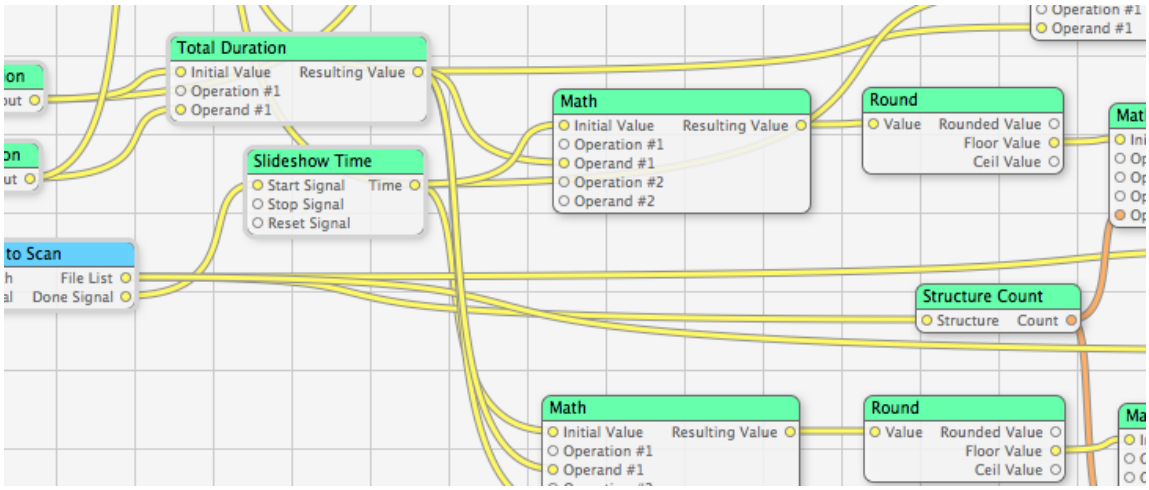


Figure 2-4: A more complex project in Quartz Composer.

2.2.2 Graphical Rewrite Rules

Instead of forcing their users to translate their solutions into "code," these systems allow the users to dictate the behavior agents or objects within a rigid world through graphical rewrite rules: specifying before and after states of their agents and objects. Figure 2-5 shows a sample rule from StageCast, a graphical rewrite rule environment for kids, that specifies that an agent should go on top of the blue vase if if the agent is initially to the left of the vase [4]. By directly manipulating these graphical rules, these programming systems make it easy to create animations, movies, or even simulations that are purely visual.

Figure 2-5: In this rule, the girl should hop over the blue vase if she is to the left of it. The before image has the girl on the left side of the vase and the after image has the girl on top of the vase.

StageCast also has some built in debugging features to help users step through their code. During the execution of a simulation, users may select a sprite and step through the execution of that agent's rules, watching as some rules are skipped or executed depending on the current conditions and environment of the chosen agent. Users may also specify

breakpoints ahead of time between rules to pause the execution at those breakpoints. Figure 2-6 shows a sample graphical rule list with a user specified breakpoint.

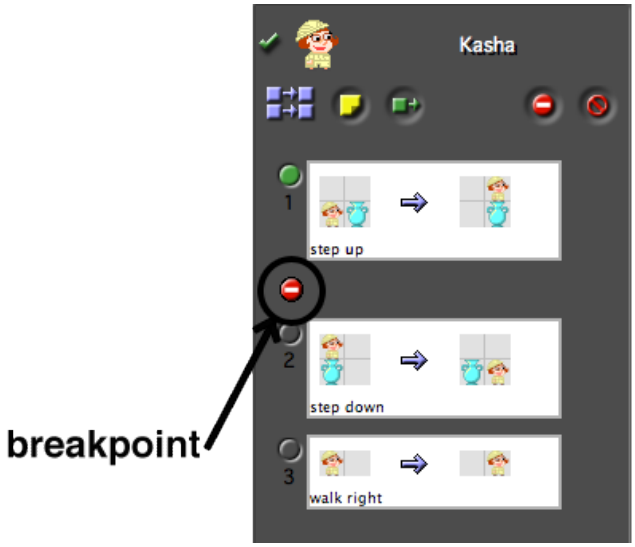


Figure 2-6: StageCast lists the rules that an agent follows. Rule 1 is being executed as indicated by the green circle next to the rule. A user may step through an agent's rule list and place breakpoints between rules, as indicated by the red circle with the white line across it.

In AgentSheets, another graphical rewrite rule environment, users can create games and simulations. This system was also designed as a framework to build other visual programming languages that were more domain oriented [11]. Application developers could program behaviors of agents and add additional functionality through AgentSheet's programming language called AgentTalk. These programmed agents then become the language components of the new visual programming language that end-users will interact with. Using AgentSheets as a framework, several visual languages were built such as an application to support the design and simulation of voice dialog interfaces.

These graphical rewrite rules, however, do not scale well when trying to express more conditions for a particular agent. For example, the set rules for a sprite like a lion to walk around a maze made with different objects can grow rapidly as the user tries to specify rules for all types of combinations. In addition, graphical rewrite rules can not be generalized and reused. A rule to turn left at a particular wall must be created for all sprites within the system.

2.2.3 Block Programming

In graphical block programming, users manipulate and connect puzzle-piece like objects to build their programs. The shape of these blocks dictates the syntax of the language: only blocks with complementary shapes can connect. Thus, users are not forced to learn and recall confusing syntax, helping to reduce the learning curve. In addition, allowing only complementary blocks to connect prevents users from making syntax errors. Figure 2-7 shows some block code from LogoBlocks, a block programming environment where each block represents a command to control Programmable Bricks. This system influenced many other languages[1] such as Bongo, Flogo, Mindstroms, Tangible Programming Bricks, Scratch, and StarLogo TNG.

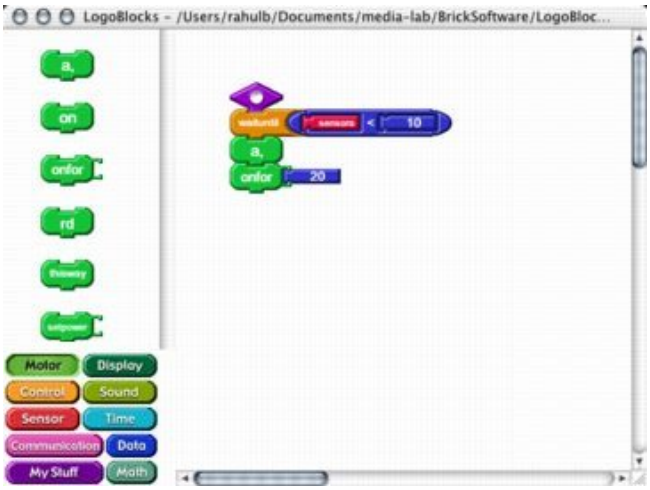


Figure 2-7: The LogoBlocks programming environment.

The shapes and colors of blocks also provide a visual overview of code, allowing users to more easily browse and follow the pathways of their code[12].

As projects grow, however, these graphical blocks like the nodes in node-based programming can take up a lot of screen real estate. In addition, repeatedly dragging and dropping blocks can be tedious for building block configurations like algebraic expressions. Dragging and dropping every operand and operator in the expression $2 + 3 - 5$ can take some work. Figure 2-8 shows a more complex project of a strategy game developed in StarLogo TNG, another graphical programming environment for secondary students and teachers to create 3D simulations and games.

From StarLogo TNG, we designed the OpenBlocks framework, as OpenBlocks inherits

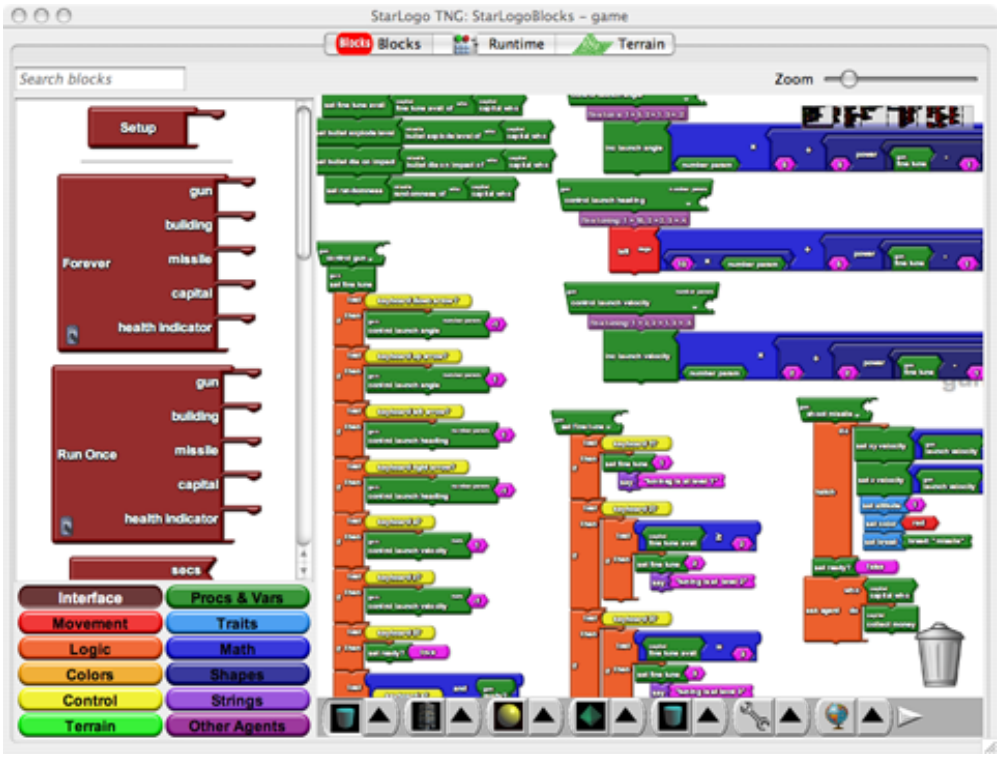


Figure 2-8: A more complex project in StarLogo TNG

many of its generic block programming features. StarLogo TNG addressed the space problem of blocks by allowing users to manipulate the zoom level or overall size of the blocks. Also, to make code or project building more efficient, StarLogo TNG has a feature called TypeBlocking that automatically bring blocks from the block factory to the main work area by entering the block names [13].

The next chapter describes the features of StarLogo TNG further as well as some of its shortcomings.

Chapter 3

Assessing StarLogo TNG

This chapter describes and assesses the block programming environment StarLogo TNG from which OpenBlocks was derived.

3.1 Background

StarLogo "The Next Generation" (TNG) is based upon the text-based programming environment StarLogo. StarLogo was designed to create and study models of decentralized systems, where interesting patterns would arise from interactions between individual objects or agents [14]. For example, a student can program a car to follow rules such as moving at the speed limit and stopping if the car in front of it has stopped or if it hits a 'stop' sign. When running several cars with these rules, a student can observe how traffic patterns can form from these individual behaviors.

StarLogo TNG extended StarLogo to use the block programming model of LogoBlocks (discussed in section 2.2.3) to leverage the benefits of block programming such as visually apparent language and syntax. Figure 3-1 shows a procedure called *wiggle*, which performs random movement, implemented in both StarLogo and StarLogo TNG code. Like LogoBlocks, users drag blocks from a categorized palette of blocks on the side onto a main work area called the block canvas. Each block in StarLogo TNG represents a command or a value. Users can build a program by stacking blocks from top to bottom. These programs control turtles or other 3D agents in an environment called Spaceland.

The shapes and colors of blocks can also provide a visual overview of code, allowing users to more easily follow the pathways of their code [12]. Figure 3-2 illustrates the browsability

and organization of block code within StarLogo TNG, especially juxtaposed to the textual programming environment of StarLogo. The code from both environments create the same biological simulation of how termites construct their mounds. Looking at the StarLogo code, users may discern sections of text within the environment, but users can not easily see if these sections represent procedures or setup code. Looking at the block code, users can also locate sections of block code, but they can also determine the content of these block collections or stacks by examining the colors and shapes of the blocks. For example, *procedure* blocks in StarLogo TNG have a slanted left edge, are dark green in color, and are always the top block of its stack. With these graphical qualities, users can immediately resolve which stacks of blocks belong to a *procedure* block stack.

Figure 3-1: The StarLogo and StarLogo TNG code are both expressing the same commands for the procedure *wiggle*.

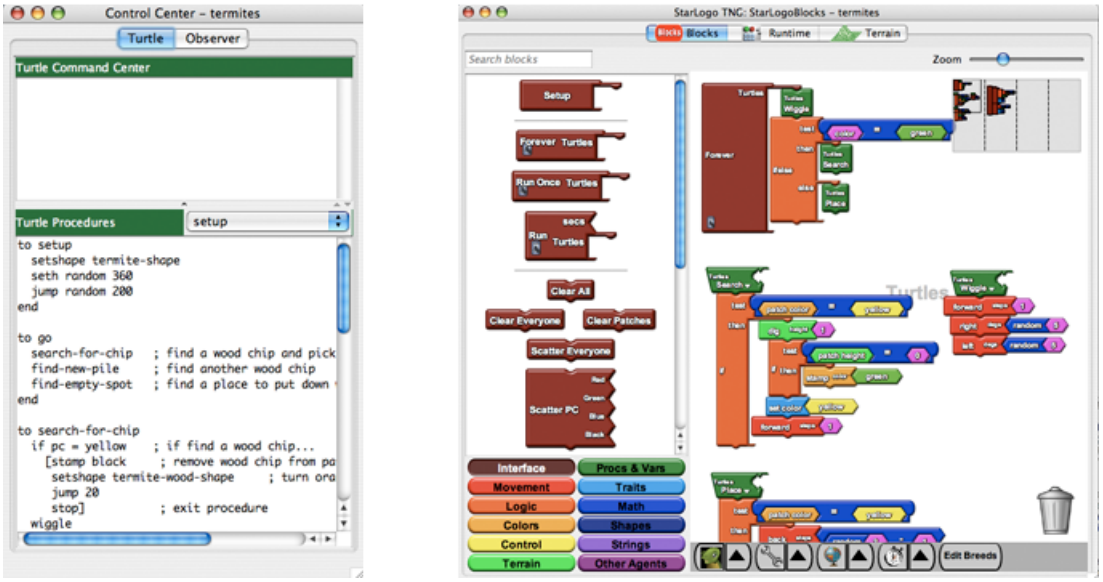


Figure 3-2: Users can drag blocks from the palette of blocks on the left side to the main work area called the block canvas.

StarLogo and StarLogo TNG target similar audiences, namely middle school, high school, and college students and their teachers. In addition to supporting the creation of simulations, StarLogo TNG was also designed to facilitate game construction to entice younger audiences into programming [3]. The computer programming experiences of these audiences vary, from none to a significant amount of experience. The user interface design of StarLogo TNG aimed at novice programmers, but StarLogo TNG also implemented features to help users become more efficient and manage the complexity of their projects as these users became more advanced programmers.

3.2 Programming Features of StarLogo TNG

StarLogo TNG adopted the block programming model of LogoBlocks and further refined the programming model by adding features to address some of the problems of block programming environments discussed in section 2.2.3 and to help users manage the size and complexity of their projects. OpenBlocks inherits some of these new features that StarLogo TNG implemented, such as the following:

- Typeblocking: More advanced programmers can type the names of blocks onto the canvas and the block instance requested will "fly" out from the block factory [13]. This feature is an alternative to repeatedly dragging and dropping blocks from the palette to the workspace.

- Auto-complete: As programmers invoke blocks by using typeblocking, a popup appears suggesting blocks to the programmer that can connect to the currently selected block, based on the current input, as shown in Figure 3-3.

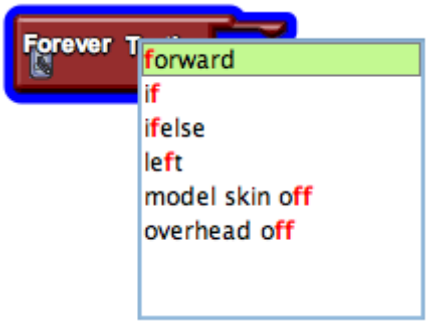


Figure 3-3: The popup suggests blocks to the user based on the current input.

- Block search: As the number of blocks in the main palette and canvas grow, users can search for a block and the desired blocks highlight themselves for the user to locate, as shown in Figure 3-4.

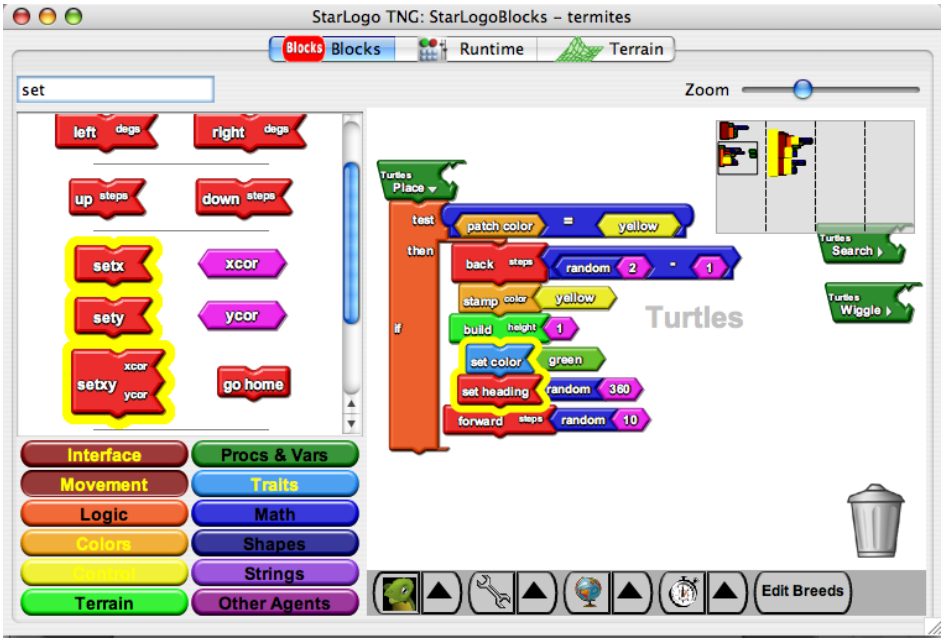


Figure 3-4: As users enter text in the top left search bar, blocks with labels that contain the current query are outlined in yellow.

- Syntax highlighting: Colors of the block denote what collection of blocks or commands a block belongs to.

- Code folding: For blocks that enclose or represent an entire block stack such as procedures, the block stack can be "folded" such that the only the representative top block is visible, as shown in Figure 3-5.

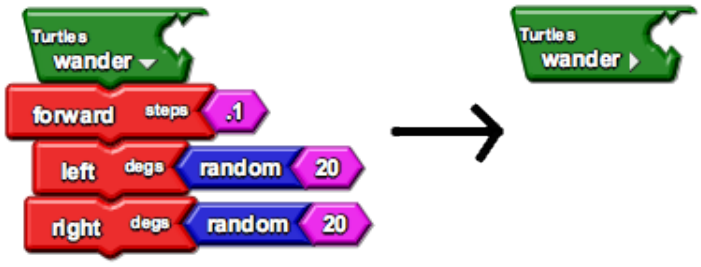


Figure 3-5: The minimap in the top right corner shows users a tiny birds-eye view of their block canvas.

- Minimap: A small panel shows a tiny representation of the entire canvas of blocks to give users an instant birds-eye view of their programming project, as shown in Figure 3-6. Users can also use the minimap to navigate around the entire block canvas.

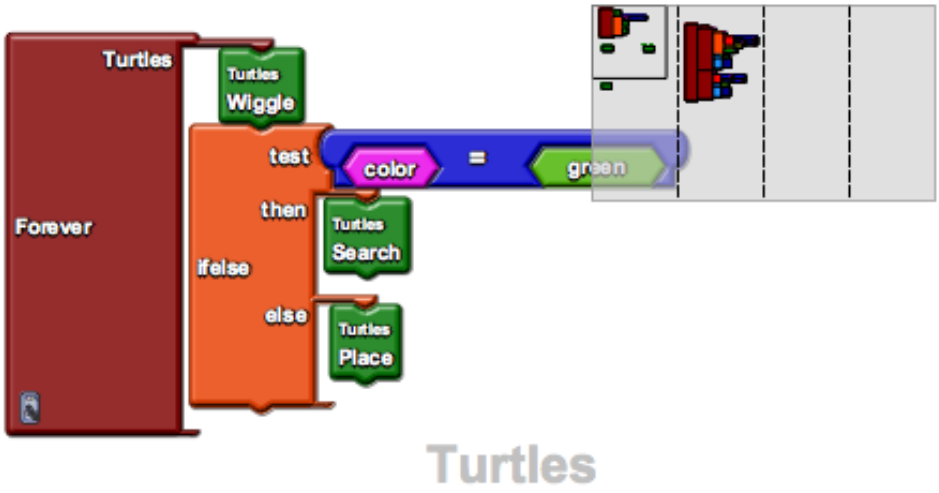


Figure 3-6: As users enter text in the top left search bar, blocks with labels that contain the current query are outlined in yellow.

- Zooming: Users can manipulate the zoom level of their entire block canvas and choose a zoom level to work in. By controlling the zoom level, users also control amount of visible space they have on their block canvas.

- Pages: The main work area or block canvas is a large horizontal space that is divided into 'pages' to help users organize their code.

- Block drawers: Block drawers are dynamic block palettes or containers that can show and hide itself as well as change its content of blocks. Unlike the block factory to the left of the block canvas, the dynamic nature of these drawers helps to save space on the block canvas.

- Block Families: Blocks with very similar properties, usually with only one difference, are grouped into *block families*. Blocks within families have a combo box around the text label of the block. Users can change these blocks by selecting another related block from the same family, saving them the time of dragging out a new block, disconnecting the old one, and connecting the new block. Figure 3-7 shows the *forward*

and *back* block family, two commands that can move agents forward and backward in Spaceland.

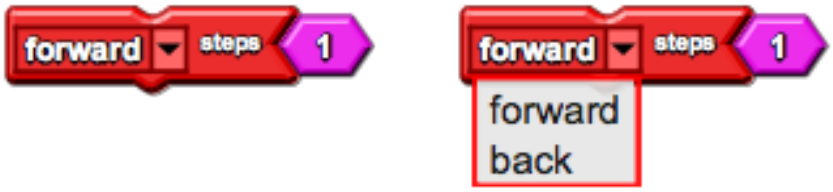


Figure 3-7: A combo box appears over the 'forward' label of the block. Users can change this block to a *back* block, which is in the same family as *forward*.

StarLogo TNG also added more features to the graphical blocks themselves that OpenBlocks inherited as well:

- Polymorphic block connectors: Connecters can have a variable block shape and once a block connects at that connector, the connector shape will assume the shape of the opposite connector, as shown in Figure 3-8.

Figure 3-8: Polymorphic connectors take on the type of the opposite block connector attached to it.

- Expanding sockets: When a block is connected to another block with expanding sockets, the block will add another socket to itself.

3.3 Evaluations

StarLogo TNG has undergone both focused user tests and field tests in classrooms. Users in all of these tests had very little or no computer programming experience.

3.3.1 Field Tests

In field tests [2, 15], students were given some instruction on block programming, such as dragging and dropping blocks and connecting them to form programs to control agents in Spaceland. In field and pilot tests, StarLogo TNG appealed to both girls and boys. Students followed the flow of programs more easily by using the shapes and colors of the blocks as guides. In addition, because of the restriction on connecting blocks of complementary shapes, students spent more time building their projects and debugging only logical errors within their code instead of syntax errors. Finally, while one of the pilot tests occurred on a weekly basis, the visual nature of the blocks helped students recall the meanings and commands of blocks they worked with the previous week.

3.3.2 User Tests

In the more focused user tests, we gathered detailed observations regarding the usability of its programming features. In these user tests, we tested one user at a time and provided each of them with a set of tasks to perform. Users were given only a description of what block programming is, instead of an explanation of how to program using the blocks. For example, we did not demonstrate how to build a simple program. Some of these user tasks were as simple as dragging and dropping specific blocks, to test how users performed such basic tasks. Other tasks required users to build large block stacks to test how each user managed their growing projects.

In the first phase of tasks, we asked users to perform simple tasks such as dragging and dropping blocks from the block factory to the canvas, removing blocks from the canvas, connecting blocks to form a stack. The following enumerates recommendations based on the usability problems we discovered:

- **More drag and drop affordance needed in blocks.** Dragging and dropping was not obvious to all the users. Changing the mouse cursor to an open hand icon when the user mouses over a block can suggest to the user that this object can be dragged.

- **Add more ways to remove a block.** Users wanted to "bring back" or "return" the blocks when we asked them to remove them. Others clicked on blocks and pressed delete, while some right-clicked on blocks and waited for a context menu to appear. Not all users noticed the trash can icon in the bottom of the canvas immediately. Multiple ways should be provided to perform basic and common tasks.

- **Implement consistent and visual feedback when connecting blocks.** When asked to build block stacks, users were confused by one kind of connector shape shown in Figure 3-9 called the command hook. Users would connect blocks with complementary shapes like the *forward* block to the left side of the *forever* block's opening, instead of connecting to the nub at the end of the command hook. Enabling block connection to happen at a wider radius or at the left side connection space can prevent this problem. Also, providing the user with more visual feedback, such as highlighting possible connectors, can act as a guide for the user.

Figure 3-9: Users would try to connect blocks on the left side of the opening for the *forever* block instead of the nub at the end of the command hook.

- **Do not overload block connector shapes.** Some blocks in StarLogo TNG language should connect based on their block shapes, but do not when users attempt to connect them, which contradicts one of the basic contracts of block programming. This contradiction occurs because a block connector shape has an overloaded meaning in StarLogo TNG, allowing only blocks that both match in shape and syntactical meaning to connect. For example, the *forward* block in Figure 3-10 seems like it can connect with the *setup* block, but only blocks that perform setup operation like *clear all* can connect to it. Such blocks caused much confusion with users. While there are a limited number of connector shapes, block connector overloading should be avoided as much as possible.

- **All block factories should be visually apparent.** Some block drawers only appeared if the user clicked on the button to activate the drawer. These hidden drawers frustrated some users, when they needed to locate blocks that only resided in

Figure 3-10: Users would try to connect blocks that match the connector shape at the *Setup* block but find they can not for syntactical reasons even though they visually can.

these drawers. Only after users began randomly clicking buttons did some users find the drawers and the blocks within them. While StarLogo TNG has a block searching tool located above the block factory, users could not use this tool immediately since they were not familiar with all the blocks in the language and could not recall what the names were of particular blocks yet. Instead of separating block drawers from the main facotry of blocks, users can find these drawers and their blocks more easily if they were more integrated or physically closer to the main factory of blocks.

As their projects grew in size, many users developed their own strategies to manage the complexity of their projects, while some felt overwhelmed by the size of their projects.

- **Provide feature to automatically clean up canvas blocks.** Users would spend time manually organizing the stacks of blocks in the canvas themselves, moving stacks of blocks to make room for new ones. An automatic arrange feature could help users clean up their workspace.

- **Implement multiple block selection.** As users cleaned up or organized their canvas space, users would move blocks or stacks individually. Multiple selection of stacks can save them time and effort.

- **Available canvas space needs to be more apparent to users.** Some users would also only limit themselves to the visible space of the block canvas. They were not aware that it was much larger than it appeared to be. In order to access other parts of the workspace, users must drag the block canvas or use the minimap, which most users did not notice. Applying scroll bars to the block canvas can be one indication to the user that there is available space.

- **Optimize basic functionality.** As programming projects grew, users ran into performance issues that made simple tasks such as dragging and dropping blocks and scrolling through the workspace difficult.

The next chapter discusses the design of OpenBlocks and how OpenBlocks integrated the generic features of StarLogo TNG and some of the improvements in the usability and performance of its inherited features.

Chapter 4

Design of OpenBlocks

This chapter presents the design of the OpenBlocks framework, which adopts many block programming system features from Starlogo TNG described in the previous chapter. It also introduces improvements in the user interface design based on the evaluation performed on Starlogo TNG discussed in section 3.3.2.

4.1 Requirements and Goals

From past research work on and evaluations of Starlogo TNG and other programming environments, the following sections highlight our target audiences and list the requirements and goals considered in the design of OpenBlocks.

4.1.1 Target Audience

Because OpenBlocks is a programming environment development framework, its primary audience consists of application developers and language designers. However, in designing OpenBlocks we also took the secondary audience into account: end-user programmers that would be using the systems built by our primary audiences.

Application developers. Application developers have significant programming experience and may have degrees in Computer Science and other related fields. However, they may not have programming experience in Java or Swing. Application developers may or may not be involved with the design of the language.

Language designers. Language designers must have significant knowledge of the intended domain, audiences, and user tasks that their programming system will be built for. They may have less programming experience than application developers, but they must have an understanding of the process and difficulties of programming as they are developing a programming language.

Programming system users. The extended system implemented by application developers and language designers are built for these set of users. This audience is very broad, spanning all ages, especially those with little or no programming experience. The types of environments and task these users have varies as well: personal, educational, or professional.

4.1.2 Requirements

We developed a set of requirements that appeals to the needs of the programming system designers and to the needs of the programming system users who may have little or no programming experience. For application developers and language designers, extendability and the ease of extending OpenBlocks are important factors.

- **Essential programming environment features are provided.** These essential features include the graphical blocks, a workspace to manipulate those blocks, and drawers to store those blocks. Back-end features include mechanisms to save and load projects and undo and redo user actions.
- **Muliptle ways to configure specific UI elements.** The system should provide more than one way to configure particular user interface elements. For example, application developers should be able to choose whether their block drawers, which contain blocks, either float above the block canvas or remain in one static location.
- **Customizable programming environment.** The system should provide developers' ways to customize the programming environment such as turning certain UI elements on or off, positioning elements in multiple locations, and changing the look and feel of the overall environment.
- **Easy to extend.** Application developers and language designers should only need to work with as few components as possible to create a very simple block programming environment.

- **Easy to understand.** The features and components of the system should be easy to understand for all users. The software components should make sense to application developers. In other words, they should not have to dig through a lot of documentation and source code to get started. The features of the system should be understandable to both application developers and language designers so they can work with its entire feature set. And finally, the user interface elements and actions of the system should be intuitive for the end-user programmers.

- **Comprehensive Documentation.** Multiple sources of clear and detailed documentation should be provided in the form of tutorials, examples, specifications, and a public API for OpenBlocks framework.

- **Optimized functionality.** Common functionality such as dragging and dropping blocks within the environment should be optimized and well-tested.

4.2 User Interface of OpenBlocks

The OpenBlocks interface includes many of the basic features of a block programming environment, such as graphical blocks, block drawers that contain these blocks, and a canvas where block programs are built. In addition to these features, OpenBlocks enables application developers and language designers the ability to customize the UI of their programming system, selecting which features to support and deciding particular behaviors for some of them. These features have been user tested and evaluated to increase the effectiveness and usability of them.

4.2.1 Block Design

The graphical block design in StarLogo TNG inspired the graphical block design in OpenBlocks. We took from StarLogo TNG the primary look and feel of its blocks such as the block beveling and connector shapes. From these base set of features from StarLogo TNG, we modified some and added more features to enhance the look and usability of the blocks as described in the following sections. Figure 4-1 shows a sample set of blocks from the StarLogo TNG language implemented using OpenBlocks.

Block and Connector Shapes

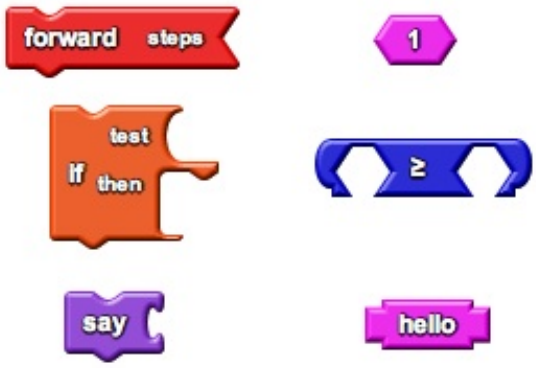


Figure 4-1: A sample set of code blocks from StarLogo TNG

Block shapes are similar to puzzle piece shapes. They are connected like puzzle pieces, by matching the blocks at their connector shapes. It is at these connector shapes where the programming language syntax is specified, as each connector shape will only connect with its complementary connector shape. OpenBlocks currently provides 14 distinct connector shapes as opposed to StarLogo TNG's six. Figure 4-2 shows four of the 14 connectors shapes.

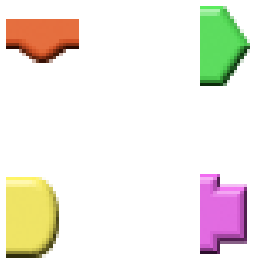


Figure 4-2: Four of the fourteen types of connector shapes. The top left conenctor is a special connector, that resides at the top and bottom of blocks. They allow blocks to be stacked on top of one another.

The Look of Blocks

The blocks are meant to be manipulated directly by the user through clicks, drags, and drops. To encourage users to directly interact with blocks with their mouse, StarLogo TNG rendered blocks using a beveling effect. To further emphasize that these blocks can be dragged and dropped, OpenBlocks implemented an open hand cursor that appears when

the user mouses over a block and a closed hand cursor that appears when the user picks up the block. To visually differentiate the user selected block from other blocks in the workspace, OpenBlocks renders a drop shadow behind the block.

Connecting Blocks

The OpenBlocks framework improves visual feedback to inform users which blocks they can or cannot connect. When a user selects a block to drag, connectors from other blocks, that the selected block can attach to, begin to highlight. In addition to highlighting connecters, OpenBlocks includes the "block stretching" feature in StarLogo TNG, such that when the selected block comes closer to the target block, the target block will "stretch" to accommodate the selected block. However, when a user selects a block and drags it to another block it cannot connect to, the selected block will be "repelled" by the target block, such that, when the user drops the block, it will be automatically moved away.

4.2.2 Workspace: Programming Environment

The programming environment or the Workspace is divided by multiple user interface components called *widgets*. Each widget serves a different function with regards to the blocks. Figure 4-3 shows a sample workspace with its different components. Like in StarLogo TNG, the primary space where users develop their code is called the *block canvas*. The canvas itself can be divided into *pages* to provide further organization.

Multiple Block Drawers

Blocks are initially stored in Block Drawers. These drawers can be separated from each other in the environment, but this layout is not recommended. Many users during the StarLogo TNG user tests did not notice or use drawers that were separated from the main set of factory drawers. Thus, users had difficulty locating blocks that were contained in these drawers. The different types of drawers that developers and designers may choose from are as follows:

- Factory: This drawer contains blocks that produce multiple instances of themselves. Also, when a block is dragged and dropped over a factory drawer, the block is removed, since users are bringing the block "back" to where it came from. In the StarLogo TNG

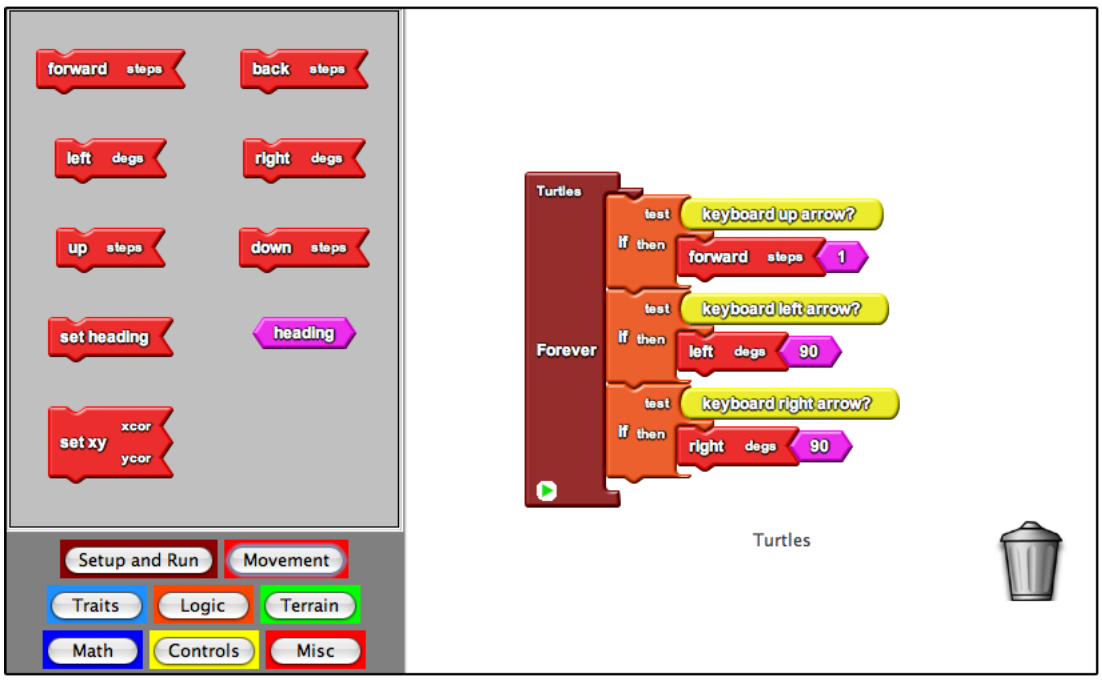


Figure 4-3: A sample workspace implemented using OpenBlocks.

user tests, many of the users attempted to remove blocks by returning blocks to the factory, but this technique was not supported by TNG.

- Page: This drawer is associated with a page that resides in the canvas and can contain blocks that are unique to that page. It can consist of blocks that produce multiple or singles instances of itself.

- Custom: Factory and Page drawers and their block content are specified by the programming system designers in the Language Definition file, which is discussed in section 5.2.1. Users, however, can create their own drawers and determine its block content using Custom drawers. Users can drag blocks into and out of this drawer and then save their custom drawer for later use.

Block drawers can either be static or dynamic. Figure 4-4 illustrates the two types of drawer states. Static drawers remain in their specified location throughout an entire programming session. Dynamic drawers float above the block canvas and can be moved around within the workspace. Users can also toggle the visibility of these drawers.

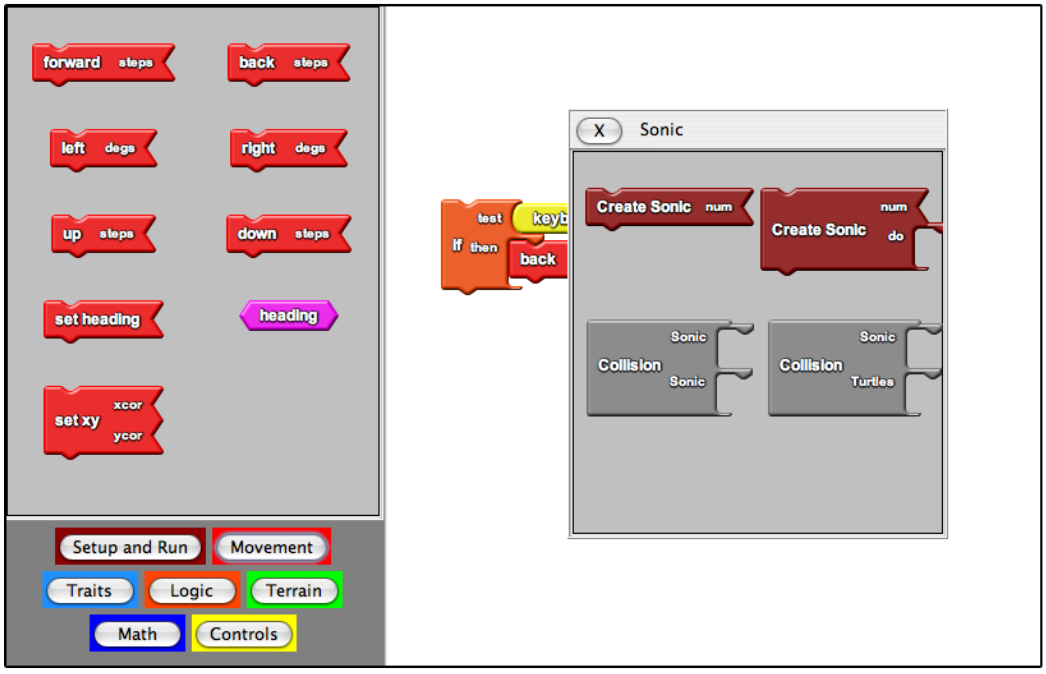


Figure 4-4: Static and dynamic drawers.

Managing Project Complexity

To manage the size and complexity of block projects, OpenBlocks assumes many of the tools StarLogo TNG implemented such as zooming, code folding, and the minimap. These tools are described in more detail in section 3.2.

This framework also adds an automatic clean up feature that arranges block stacks within the canvas. The current implementation, however, naively arranges block stacks by the y-coordinate of their top block. Future implementations would provide multiple options to organize block stacks.

To help users document their code, OpenBlocks adds a new way to comment on blocks using floating "post-it" panels that are linked to specific blocks, as shown in Figure 4-5. These comments can be created by right-clicking on a block and selecting 'Comment Block' from the context menu that pops up. More ways to create a comment should be added in the future, as OpenBlocks only supports the right-clicking option. The comment panel can be hidden or removed, and it moves wherever the blocks moves.

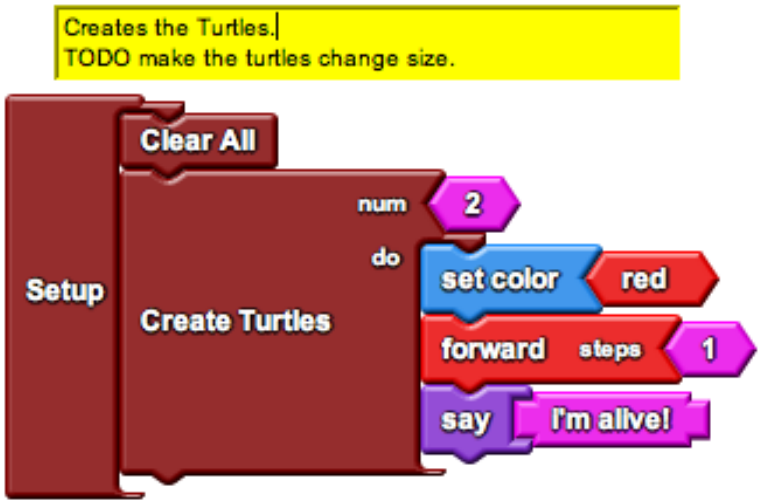


Figure 4-5: Comments appear as post it notes and are linked to blocks.

4.3 Software Optimizations

In developing the software of the OpenBlocks framework, we optimized as much common functionality that we could anticipate, such as dragging and dropping blocks. Since we cannot anticipate every usage scenario for OpenBlocks, future applications and evaluations of this framework will determine any further optimizations. The set of optimizations described below originated from evaluations of StarLogo TNG:

- Dragging and dropping of blocks over the block canvas: This operation slowed drastically as projects grew because the dragged block would be compared against all blocks within the canvas. Now OpenBlocks checks over a bounded region instead of the whole canvas.

- Scrolling and moving the canvas: Whenever the canvas view changed, all blocks on the canvas were redrawn, which involved recalculating their shapes and rearranging their components such as their labels. Now instead of redrawing all the blocks, it repaints buffered images, espeically since the blocks do not change in appearance as the user scrolls.

- Rendering of blocks: Blocks are only redrawn if their appearance needs to change. Otherwise, a buffered image of the block is repainted.

- Saving and loading projects: Between the saving and loading different projects, only

the differences are loaded instead of re-loading the entire block language and default set of workspace widgets.

4.4 Format of Save Files

OpenBlocks has saving and loading mechanisms built in that application developers can immediately use. The format of the saved files is similar to the format of the language definition file. The contents of saved files, however, only contain modified data. For example, if a user modified a block label, that new label will be saved. However, if a block's label never changed, then that block label will be omitted from the save file.

Saving only the modified information will be useful in internationalization, particularly when sharing programming projects across different languages. For example, if a user creates a project in Spanish, saves it, and then loads it in another environment in English, the blocks within the English programming environment will remain in English, except for the blocks that were modified by the Spanish user.

4.5 Limitations

This section enumerates some of the limitations of this framework.

- **At most one return type allowed per block.** Like Starlogo TNG, OpenBlocks limits blocks to have at most one return type. Implementing multiple return types introduces complexity. For example, with multiple outputs for a block, a user can potentially connect multiple independent stacks to each output of that block. Also, if a block language implements procedures, language designers must determine the interface for their users to return multiple values. Finally, what can be done by using a block with multiple return types can be done using multiple blocks with single return types.

- **Compiler/Interpretor for language must be built from scratch.** The framework does not have a built-in compiler to parse and interpret the language created. As each language is different, the compiler or interpretor, which are responsible for understanding and placing meaning to blocks and their configurations, must be implemented by the application developer. However, implementing the compiler or in-

terpretor from scratch enables greater flexibility and specialization for the language of the programming system.

- **Framework developed in Java and Swing.** These technologies were chosen to enable platform independence. In addition, we wanted to exploit the already implemented and tested features of Swing to develop the GUI. The development of this framework in Java and Swing does not limit developers to these technologies, as Java can still work and interact with other modules implemented in different languages. Starlogo TNG, for example, has a virtual machine written in C and an 3D environment written in OpenGL.

- **Limited look and feel of the programming system.** Currently, in order to change the look and feel of the programming system, application developers must dig through the source code of OpenBlocks. An interface to more easily modify the look and feel is proposed in Chapter 7, which discusses future work.

Chapter 5

Extending OpenBlocks

This chapter describes how to extend the OpenBlocks framework to create a block programming system. We recommend that language designers spend the first phase of their design process thinking about the design of the language itself without taking the features of OpenBlocks into account. Language designers must make sure that OpenBlocks can satisfy the requirements of their language instead of fitting their language within the requirements of OpenBlocks. A user-center design approach is recommended when designing the language and programming environment [16].

Once language designers have determined that OpenBlocks can be used, section 5.1 provides additional guidelines for the language design so that designers can take advantage of OpenBlocks' full feature set and how to represent that language graphically. Once the language has been specified, section 5.2 describes how to build the programming system to support the language.

5.1 Designing a Programming System

Probably the most difficult step in building a programming system is designing the language. Language designers are faced with the following problems:

- Understanding the target users to anticipate their needs and expectations.
- Aiming for a language that closely maps to end-users' natural language, so that users can easily "read" and "write" in the programming language.
- Achieving the right level of abstraction, so that users can understand the language.

- Having all the necessary components to express solutions, so that users can translate their ideas to code.

Fortunately, OpenBlocks makes it easy to generate a running programming system by simply specifying a single file called the Language Definition file, which is described in section 5.2.1. Little programming knowledge is required to specify this file as it is done in XML.

5.1.1 Language Guidelines

The following are some guidelines to follow to further refine a block language. These guidelines were gathered from experiences in developing StarLogo TNG and EvoBeaker, the case study described in Chapter 6, and background research of past programming systems.

- **Prevent users from making syntax errors.** The shapes of blocks represents a contract that only blocks with complementary shapes can connect and syntactically make sense. This contract can be breached if the system allows users to connect blocks that fit visually, but are not correct within the bounds of the language. For example, if a *move* block expects a number block, make sure that the blocks that can connect to *move*make sense. In other words, it doesn't make sense to *move* true values.

- **Do not overload connector shape type.** Each connector shape should map to a unique data type and vice versa. StarLogo TNG has three data types: number, string, and boolean. Each data type has its own connector shape. Giving a connector shape more than one associated data type can cause the user to make syntax mistakes.

- **Keep block color consistent.** Block color can be a guide for users to understand and later recall what a block does or means. In StarLogo TNG, all blocks related to movement were colored red. Users then would associate any red block to mean some form of movement. Having all the blocks of related color residing in one drawer also helps users locate blocks better.

- **Use sockets labels as needed.** Sockets labels add more meaning to a block connected at another block's socket. A *forward* block accepts any *number* block. By labeling the socket of the *forward* blocks "steps," users can determine that this number will determine how far an agent moves.

5.1.2 Usability Guidelines

The following are some usability guidelines to help design both the block programming language and environment. These guidelines were gathered from designs and usability tests of StarLogo TNG and EvoBeaker.

- **Constrain the environment for novice users.** Allowing novice users too much freedom to change the interface of the programming environment can lead them into a configuration that would not be conducive for learning and understanding this new system. For example, factory drawers are best kept as static for novice users. If they were able to hide factory drawers, the entire language now becomes invisible to them until they bring the drawers up again.

- **Enable more expert users to configure environment.** Expert users, unlike novices, have an understanding of the different tools and features of an environment. And with experience they can perform the basic tasks much faster. Allowing them to configure their environment allows them to set up their workspace to match their new needs as more experienced programmers. For example, some expert users may want typeblocking enabled by default.

- **Provide constant visible feedback for user actions.** OpenBlocks tries to provide as much visual feedback to the user, but it cannot anticipate what all user or program actions must display feedback. For example, if a block or set of blocks have a compile error, OpenBlocks will not know this since the compiler is not within the framework. Thus, application developers must notify the framework with the appropriate feedback to display.

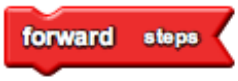
- **Large subsets of blocks may overwhelm novice users, consider a small subset of blocks.** Adding blocks to the framework is easy with the Language Definition file, and so, the number of blocks in the system can increase rapidly. However, in user tests, we found that too many blocks overwhelmed novice users and made them unsure of which blocks were more useful and important from the rest.

5.2 Building the Programming System

Once the language and the programming environment are designed, application developers can then specify the Language Definition File, which is described in section 5.2.1. Afterwards application developers can immediately run the WorkspaceController class described in section 5.2.2 to see a visual representation of their design and specifications.

5.2.1 Language Definition File

Developers can use the Language Definition file and its specified XML schema to describe the layout of the programming environment as well as the components and properties of every single block. Figure 5-1 shows a sample XML specification for a *forward* block in StarLogo TNG. Certain block properties may be specified as immutable or mutable, such as block labels. Some blocks may have labels that do not change over time, whereas others may have labels that can be edited by the user. Changes to the block language can be made quickly by changing values of properties within the XML Language Definition file.




```
<BlockGenus name="fd" kind="command" initlabel="forward" color="255 0 0">
    <description>
        <text>Agents move steps' forward.</text>
    </description>
    <BlockConnectors>
        <BlockConnector label="steps" connector-kind="socket"
            connector-type="number">
            <DefaultArg genus-name="number" label="1"></DefaultArg>
        </BlockConnector>
    </BlockConnectors>
</BlockGenus>
```

Figure 5-1: A sample XML specification of a *forward* block.

In addition to specifying the language and programming environment, the file also specifies the following:

- Mapping between block connector type (i.e. number, string) to block connector shape (i.e. round, square).
- Location and block content of block drawers.

- Default set of pages.

5.2.2 WorkspaceController

The WorkspaceController (WC) is an abstract Java class that acts as an interface between the programming environment, the Workspace, and the application enclosing the OpenBlocks framework. To immediately get a programming environment started once the Language Definition file has been specified, application developers only need to run from the main method of this class. The WC will pass the Language Definition file to the OpenBlocks library and generate the language and programming environment.

In addition to loading and generating components from the language definition file, the WC provides the following functionality for application developers to use:

- Saving and loading in the OpenBlocks save format.
- Reseting the project state.
- Relaying undo/redo commands to the Workspace.
- Setting the Workspace zoom level.

Application developers may either extend this class or modify its source code to customize and further control the flow of particular processes such as saving and loading.

5.3 Other Extendable Components

While the following features enable more customization of the programming system processes, they require the application developer to write and modify some source code.

5.3.1 Customized Block Shapes

Code block shapes are rectangular with connectors shapes along its sides. The top and bottom connectors slightly modify the tops and bottoms of blocks, that is if a block has them.

By implementing the CustomBlockShape class, application developers can specify the four corners of a block shape: its top left, top right, bottom left, and bottom right corners. OpenBlocks will then interpolate between the four corners and render the rest of the block components accordingly. Figure 5-2 shows a custom block used in StarLogo TNG.

Figure 5-2: A custom block used in StarLogo TNG. This *procedure* block has its bottom left corner offset from its top left.

5.3.2 Link Rules

The basic rules of connecting between blocks are through their connector shapes. One connector shape can only connect with its complementary connector shape. However, if language designers would like to add a level of complexity to their block linking rules, application developers can create extra rules by implementing the LinkRule class. Developers can implement methods within this LinkRule interface so that, given two blocks and their connectors, developers can apply more complex rules that rely on other properties of blocks besides their connector shapes. The rules that the developer specifies are checked every time a user moves a block within the canvas. Thus, the implementation of these added rules should be efficient.

While adding more complex rules may seem like a common feature among block languages especially since only 14 connector types are provided in the framework, we decided to discourage application developers and language designers from adding more complex rules than necessary, as it puts more load on the user to remember or understand these additional rules. It is confusing for users to see that the connectors shapes match, form an idea to connect them, and finally attempt to connect them, but discover that the blocks cannot connect. Thus, adding more complex rules can waste users' time and cause confusion.

5.3.3 Listening to Workspace Events

If the developer would like to perform more customization or respond to particular user actions within the workspace, developers can implement WorkspaceListeners that listen for significant user actions. For every event, workspace listeners receive WorkspaceEvent instances that contain information regarding the block, workspace widget, and other relevant objects and data. Some of the events that workspace listeners receive are:

- Individual block events: block added, removed, renamed
- Block connection events: block connected, disconnected

- Page events: page added, removed, renamed

5.3.4 Other Public Classes within Library

The OpenBlocks Library is packaged as a Java jar file. Application developers may import public classes of the library such as the following:

- **BlockGenus:** A block genus describes the properties that define a common set of blocks. For example, the *forward* block in the StarLogo TNG language has a BlockGenus to describe the common properties that all *forward* block instances inherit. All the data in this class is immutable.

- **Block:** Block holds all the mutable data regarding each instance of a graphical block. Developers can access and modify this mutable data such as the block label (if the block is editable). Developers can also set their own properties within each block.

- **Workspace:** This class provides access to the current state of the Workspace, or the programming environment. For example, developers can access all the blocks currently on the block canvas.

- **BlockDrawer:** This class acts as a container for block instances. Developers can access the contents of a BlockDrawer and add and remove blocks within it.

These public classes will be especially useful for developers that build a block language compiler or interpreter to parse through the current block stacks within the block canvas.

Chapter 6

EvoBeaker: A Case Study

This chapter describes our evaluation of the usability of the OpenBlocks Framework by performing a case study on EvoBeaker, an educational software tool to teach middle school students micro- and macro-evolutionary biology [17].

6.1 Documentation and Support

Before the case study with EvoBeaker, OpenBlocks had the following documentation and support materials:

- **An API for the OpenBlocks Java library**: This API includes a brief description of all public classes within OpenBlocks and their public method signatures.
- **An XML specification of Language Deinition file**: This document explains how to build a language definition file and describes all the necessary elements to specify.
- **Sample Language Definition file**: This file includes example specifications for different blocks and a simple workspace layout.

6.2 Testing Framework with EvoBeaker

Creating the OpenBlocks framework was partly inspired by collaboration with EvoBeaker, an educational simulation software for middle school students in Maine. EvoBeaker features experiments where students can explore biological and ecological models such as the dynamics between predators and prey. However, students could only work within the pre-defined

parameters of each experiment. Modifying these experiments would require students to understand and parse through complicated XML files that described these experiments.

The EvoBeaker development team became attracted to the visual nature of StarLogo TNG's block programming system and decided to incorporate a similar system into their application for students to edit their experiments by block programming. However, the source code of StarLogo TNG was very specific and dependent on the StarLogo TNG language and could not be easily be reused by another system. Thus, we developed the OpenBlocks framework, which inherited many of the features from StarLogo TNG, for their developers to extend.

6.2.1 EvoBeaker Collaboration

The EvoBeaker team will use OpenBlocks to allow their students to create new rules and behavior for models within each of their experiments. For example, a student may be examining the foraging strategies of rabbits in one experiment. Currently, the rabbits forage for food only within a small radius from their burrows. The student wants to modify the foraging behavior of rabbits to wander beyond their burrows. The following is a sample usage scenario for the student to edit rabbits' foraging behavior:

1. Student is working on experiment studying foraging habits of rabbits in the main workspace of EvoBeaker.

2. Student wants to change the behavior of the rabbits.

3. Student clicks on a "special button" that launches the block programming environment. The environment already contains the blocks that define the behavior of rabbits.

4. Student edits rabbits' behavior by manipulating the blocks, such that the rabbits forage farther from their burrows.

5. Student saves edited blocks by pressing another "special button" that sends the block code back to the experiment and returns to the original EvoBeaker screen. The experiment will now run with the edited behavior.

The EvoBeaker application was developed in Cocoa and Objective-C. Before using the Java OpenBlocks framework, we built an interface between the Cocoa and Java components

using the Java Native Interface (JNI). The two components can exchange string data and messages through this JNI communication channel.

EvoBeaker began working with the OpenBlocks framework in January 2007. The framework included the documentation materials described in section 6.1. In the months following, I collaborated with the EvoBeaker team to set up their language file and programming environment. Their team did not use the documentation to get started. Instead, I met their team personally to explain the system, how to use the Language Definition file, how to build a simple system, and to explain the file format of save files. Figure 6-1 exhibits a set of blocks currently defined by EvoBeaker, which represent the different rules or behaviors of a rabbit for an experiment to study their foraging habits.

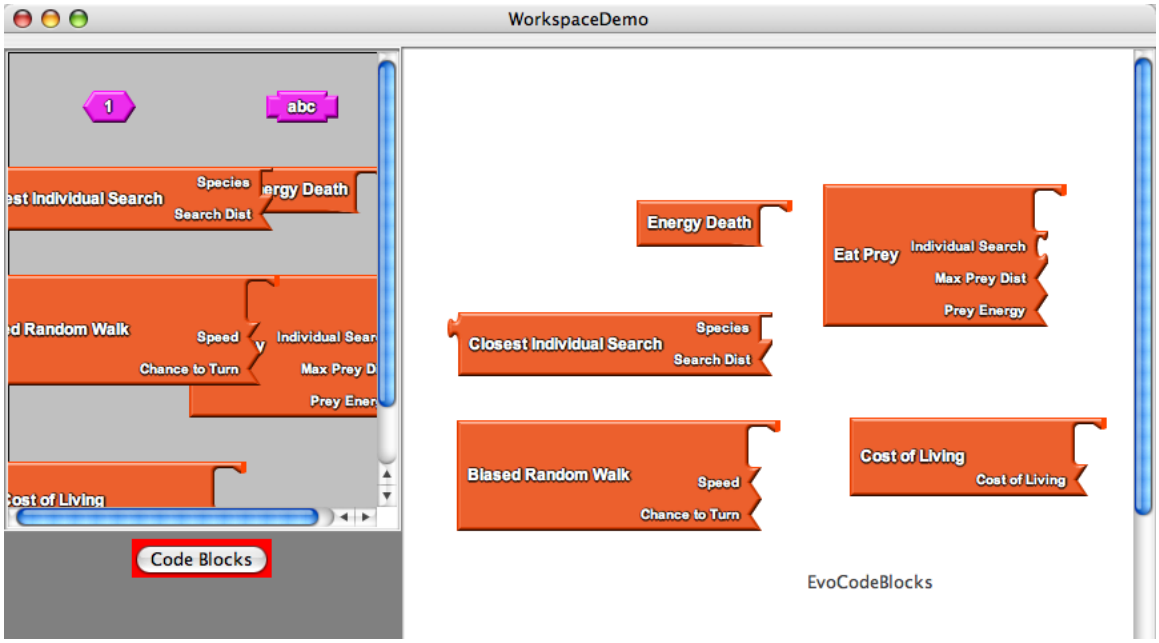


Figure 6-1: EvoBeaker's initial set of blocks are derived from an experiment that studies the foraging strategies of rabbits. Each block on the canvas represents the different behaviors of a rabbit.

6.2.2 Evaluation of EvoBeaker's OpenBlocks Experience

The assessment described in this section only evaluates the first iteration of EvoBeaker's programming system. After three months, they have created a simple language and programming environment layout, with one static block factory and a block canvas with no pages. In the following months, they will begin evaluating their language, designing their programming environment, and testing their system with students.

The following is an enumeration of our observations:

- The language definition file was easy to use. They learned how to use the file, however, because I personally met with them to explain the details of it.
- The provided sample definition was a good reference to help them get started.
- The public API of OpenBlocks Java library was sufficient in getting an overview of the system.
- EvoBeaker found the 14 provided block connector shapes to be very limiting. They requested more shapes or a way to overload the meaning of the current set of block connectors. We discouraged both features. Adding more block connector shapes would mean creating more complicated shapes that could be less distinct from the other shapes. Regarding the latter feature, if block connecters were overloaded, the block language would lose the ability to specify their syntax visually within the shapes of blocks. The resulting language would only confuse the user.
- The EvoBeaker team had difficulty understanding the relationships and differences between the different programming environment components: the different types of drawers, pages, the block canvas, and the workspace. A document visually noting the differences and describing each component would have been helpful.
- The EvoBeaker team needed more interfaces to easily access, modify data or create OpenBlocks Java objects like Blocks during the runtime of their programming system. The framework however does not prevent developers from doing such operations currently. They can perform such operations through less obvious ways: they must understand the public API and write their own Java classes to directly interact with the framework.

6.2.3 Additions and Modifications to Framework

From this initial evaluation of EvoBeaker's experience with OpenBlocks, we plan to add the following support materials and modifications to the framework:

- Design and implement more interfaces for applications developers to better integrate the framework to existing applications, to acquire and modify current programming

system information, and to create instances of Blocks and other significant workspace objects during runtime.

- More sample language definitions files of varying languages. This diverse set of files can help illustrate how specifications differ between languages.
- A simple abstract WorkspaceController class to load the specified language to a visual representation within a programming environment.
- Sample java classes that use the more complicated extensions to OpenBlocks, like LinkRules and WorkspaceListener.
- Tutorials to create a simple language and programming environment.

Chapter 7

Conclusion

This chapter describes some ideas to further extend this framework and recapitulates the contributions of this thesis work.

7.1 Future Work

This section enumerates further research and development of OpenBlocks.

7.1.1 Further Development of OpenBlocks

We tried to anticipate as many basic features that language designers would need to create their programming system. OpenBlocks currently includes the minimal set of features to create a usable block programming environment.

Facilitating Interaction with Framework

As noted in the EvoBeaker case study in section 6.2, there were no obvious ways to exchange information to and from the framework. The EvoBeaker developer also found no clear ways to modify and create objects once the programming system was launched from the Language Definition file. Currently, the only way to get information regarding current system information is to request the save string from the *WorkspaceController*. Further collaboration with EvoBeaker as they continue to use OpenBlocks will be useful in developing the appropriate interfaces to implement for external modules to better interact with the framework.

A More Customizable Look and Feel

Application developers and language designers can decide how their workspace elements are laid out, but they cannot specify the look and feel of their programming environment. Look and feel includes the overall visual design, such as colors, font-faces, and textures, and the interaction design of the interface, such as how buttons respond based on mouse input from the user. For example, designers may want to change the look and feel of their scroll bars such that they appear more metallic and change to a lighter shade when the user mouses over them. Currently, the look and feel of OpenBlocks is determined by Java's native look and feel. To change the look and feel, developers must either replace the look and feel specifications in the Swing properties file or set the look and feel in the source code themselves.

A potential solution using a properties file would allow developers to control the look and feel, similar to how the Language Definition file controls the block language and programming environment. This properties file would allow the specification of features such as:

- Font-faces and sizes of labels in blocks and buttons.
- Color schemes for the programming environment, such as the background color within block drawers and the color of the block canvas.
- Button appearance when users mouse over, press, or release buttons.
- Default block size when users initially open the environment.
- Texture of the blocks.
- Ability to enable or disable block beveling.

Multiple Open Projects

Currently, the framework only allows one active programming project. However, as users build more projects, having multiple projects open at once is a more efficient way for users to compare and exchange code between them.

Printing Block Code

From field tests of StarLogo TNG, many teachers requested the ability to print block code to have a hard copy of their students' work and to manually grade projects. When designing this feature, future developers may consider printing out the code textually or graphically. By printing the code textually, users save space and ink on paper. However, by just printing out the text labels, users lose the graphical information of blocks such as their locations within the block canvas, the shapes of their block connectors, and the visual verification of the syntax through block connections. Designers of this feature can let users decide for themselves by providing options to print textually or graphically.

Debugging Support and Features

We found in StarLogo TNG that as users become more advanced and develop more complex projects, their projects become large and unwieldy. While the programming system should be preventing users from making mistakes, language designers cannot anticipate logical bugs that users make. Logical bugs result in unintended or unexpected behavior. With the current set of features in OpenBlocks, users may need to resort to manually stepping through their projects or examining the resulting behavior or their code.

Some possible solutions to alleviate the tedious and timely work of debugging include include building a stepper interface within the programming environment that highlights the code blocks currently executing. Another interface can be created to show project state or variables. Note, however, that these proposed solutions are dependent on the language design. For example, project state and the types of variables available (if the language has any) will differ between languages. Any solution will need to be generic enough to cover some of the common usages.

7.1.2 Further Evaluation and User Testing

Studying EvoBeaker's experience with OpenBlocks showed some of the strengths and weakness of the framework. However, additional research and assessment is needed to refine OpenBlocks.

Testing New User Interface Features

OpenBlocks implemented many features to address the usability problems discovered in StarLogo TNG, such as block connector highlighting to offer more visual feedback. User tests should be performed to evaluate these new features and how they compare to StarLogo TNG under the same set of user tasks.

Applying OpenBlocks in More Diverse Contexts

Currently, both StarLogo TNG and EvoBeaker are utilizing the OpenBlocks system. Because both projects are educational and agent-based simulation environments, there is significant overlap in their user base and goals. Thus, our evaluation of OpenBlocks has been limited in these educational contexts.

However, OpenBlocks is intended for a diverse set of uses. As OpenBlocks becomes adopted in other contexts such as professional contexts, further evaluation and testing of these set of users should be done.

Developing an OpenBlocks Design Process

In addition to studying OpenBlocks usage in more diverse contexts, a more comprehensive study of how language designers and application developers create and build their programming systems using OpenBlocks can help provide insight to what worked and what did not for some languages. These studies can also give us insight to the limits of OpenBlocks' feature sets, identifying places for improvement in the framework, while also providing earlier warning to future application developers and language designers what the OpenBlocks framework is not capable of. Such prevention could help designers decide earlier if the framework is appropriate for their intentions.

This further evaluation can also focus on developing a design process that application developers and language designers can follow to help in the process of developing their programming systems using OpenBlocks. Some goals to develop this design process include:

- Improving understanding of the intended domain and anticipating and matching end-user needs and tasks.
- Encouraging application developers and language designers to design around their

problem, goals, and set of users instead of designing around the features of OpenBlocks.

- Facilitating the translation of their programming system design to features of OpenBlocks and ensuring that this translation makes sense to them and to their users.

In addition to developing a design process, further evaluation can help to develop the language and usability guidelines presented in section 5.1.

7.1.3 An Open Source Library

After performing the evaluations suggested in the previous section, OpenBlocks should be released as an open source library, which was one the original goals of this thesis work. Before the release, substantial code reviews should be conducted to revise any programming shortcuts or inconsistencies and clarify documentation. Also, tutorials for future developers should be developed to easily extend the library.

7.2 Closing Remarks

OpenBlocks is an extendable framework designed to make it easy for application designers and developers to create their own graphical block programming system. Constructing their graphical language and laying out their programming environment only requires defining a single XML file. And any further modifications of their programming system can be done by changing that same XML file, increasing the speed of design iteration.

Because OpenBlocks derived from Starlogo TNG, applications of OpenBlocks will leverage the many benefits of graphical block programming systems such as a visually apparent language and syntax. While the inherited features of StarLogo TNG help end-users manage the complexity of their code such as a zoomable block canvas and other features mentioned in section 3.2, OpenBlocks provides further improvements in usability over its predecessor such as displaying more visual feedback when users connect blocks.

The OpenBlocks framework is also customizable, so application designers and developers can create a programming environment that better matches their application's purposes and needs of their end-users. However, we could not anticipate all purposes and needs, as the the case study conducted with EvoBeaker revealed that we needed to create more interfaces

for external components to interact with the framework. Future work on this framework will require evaluation of OpenBlocks usage in more diverse contexts. Such a study can help in the ongoing development of OpenBlocks as an extendable and useful framework and in the creation of a design process for future application designers and developers to appropriately leverage the benefits of this framework.

Finally, because OpenBlocks allows for rapid development and iteration of a working block programming system, application designers and developers can concentrate on the design, rather than the implementation, of their programming system. Such ease and speed can help developers create a programming language at the right level of abstraction and expression for their end-users and a programming environment that properly supports the use of that language. The OpenBlocks framework not only makes graphical block programming systems more attainable for application designers and developers, but it also facilitates the creation of applications that can further make programming more accessible to larger audiences.

Bibliography

[1] C. Kelleher and R. Pausch, "Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers," *ACM Comput. Surv.*, vol. 37, no. 2, pp. 83–137, 2005.

[2] A. Begel and E. Klopfer, "Starlogo tng: An Introduction to Game Development," *Journal of E-Learning*, 2005.

[3] "Starlogo TNG," 2007. [Online]. Available: http://education.mit.edu/starlogo-tng

[4] "Stagecast," 2007. [Online]. Available: http://www.stagecast.com/index.html

[5] "Apple - Mac OS X - Applescript," 2007. [Online]. Available: http://www.apple.com/macosx/

[6] "The Mathworks - Matlab - The Language of Technical Computing," 2007. [Online]. Available: http://www.mathworks.com/products/matlab/

[7] "Using Intellisense," 2007. [Online]. Available: http://msdn2.microsoft.com/en-us/library/hcw1s69b(VS.71).aspx

[8] "Eclipse," 2007. [Online]. Available: http://www.eclipse.org

[9] "Visual Basic Developer Center," 2007. [Online]. Available: http://msdn2.microsoft.com/en-us/vbasic/default.aspx

[10] "Quartz Composer," 2007. [Online]. Available: http://developer.apple.com/documentation/GraphicsImaging/Conceptual/QuartzComposer

[11] A. Repenning and S. T., "Agentsheets: A Medium for Creating Domain-Oriented Visual Languages," *Computer*, vol. 28, no. 3, pp. 17–25, 1995.

[12] A. Begel, "Logoblocks: A Graphical Programming Language for Interacting with the World," Master's thesis, Massachusetts Institute of Technology, 1996.

[13] C. McCaffrey, "Starlogo TNG: The Convergence of Graphical Programming and Text Processing," Master's thesis, Massachusetts Institute of Technology, 2006.

[14] M. Resnick, "Starlogo: An Environment for Decentralized Modeling and Decentralized Thinking," in *CHI '96: Conference Companion on Human Factors in Computing Systems*. New York, NY, USA: ACM Press, 1996, pp. 11–12.

[15] K. Wang, C. McCaffrey, D. Wendel, and E. Klopfer, "3D Game Design with Programming Blocks in Starlogo TNG," in *ICLS '06: Proceedings of the 7th International Conference on Learning Sciences*. International Society of the Learning Sciences, 2006, pp. 1008–1009.

[16] J. F. Pane, "Designing a Programming System for Children with a Focus on Usability," in *CHI '98: CHI 98 conference summary on Human factors in Computing Systems*. New York, NY, USA: ACM Press, 1998, pp. 62–63.

[17] "Evobeaker," 2007. [Online]. Available: http://www.simbio.com