



VCU

Virginia Commonwealth University
VCU Scholars Compass

Theses and Dissertations

Graduate School

2023

Development of Tangible Code Blocks for the Blind and Visually Impaired

Hyun Woo Kim
VCU

Follow this and additional works at: <https://scholarscompass.vcu.edu/etd>



Part of the [Graphics and Human Computer Interfaces Commons](#)

© The Author

Downloaded from

<https://scholarscompass.vcu.edu/etd/7314>

This Dissertation is brought to you for free and open access by the Graduate School at VCU Scholars Compass. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of VCU Scholars Compass. For more information, please contact libcompass@vcu.edu.

©Hyun Woo Kim, 2023
All Rights Reserved

Development of Tangible Code Blocks for the Blind and Visually Impaired

A dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Biomedical Engineering at Virginia Commonwealth University.

By
Hyun Woo Kim
Bachelor of Science, Virginia Commonwealth University 2017

Director: Dianne T.V Pawluk, Ph.D

Virginia Commonwealth University
Richmond, Virginia
February, 2022

ACKNOWLEDGEMENTS

A lot of effort was put into steps I took to complete my journey as a PhD candidate, and a lot of people have contributed to this research. I would like to thank everyone who supported me during the journey, and foremost I would like to thank my advisor Dr. Pawluk, who taught me how to conduct research and ask important research questions. She is a great advisor, professor, and educator that I could rely on. Thank you Dr. Palwuk, I would have not done this without you.

I would like to thank Bryson Goolsby, another PhD candidate who worked with me on this research. He is a great researcher and a great friend. We helped each other and supported each other all the time. Sometimes giving ideas to each other when either of us were stuck on certain parts of the research. It was my pleasure to collaborate with you, this work could have not been possible without you, so thank you.

I would like to express my gratitude towards all the committee members. Dr. Palwuk, Dr. Krusienski, Dr. Wetzel, Dr. Chu, and Dr. De Arment. Thank you so much for being part of my dissertation committee, every advice you guys gave helped me get through this journey.

I have many name to mention, Benjamin Nguyen, Steven Bae and his family, Megan Shin, Jae Lee, Esther Jeong, Kristen Kim and her family, Jiwon Seo, Mia Choi, Minsu Ju and his family, Ellie Miyun Ju, Sookyoung Kim, Julia Hwang, Joy Lee, JiHyun Lee, Ha Young Ha Eun sisters, Tien Comlekoglu, Nathan Le, DongHo Shin, Dr. Park, Sunny Kim, Brian Kim, Skye Ali and her family, Hope Kim and her family, Seonjoo Hwang, Andy Park, Paul Park, Youjoo Lee, Amy Kim, Joanne Kim, and many other friends who supported and gave words of encouragement all the time. I would like to thank everyone again for your love and support.

Finally, my family members. My mother, father, sister, my uncles, aunts, cousins, and my Grandmother who passed away two years prior, who I missed the most. I love you guys! I would like to thank my uncle, the only family member who stayed in the United States. He invited me every holiday, fed me, supported me, and gave me a place to stay! Thank you too, Chris and Michelle, my great cousins, for always welcoming me. I miss drinking wine with you guys already.

I will never forget my journey as a PhD candidate. Thank you.

This work is based upon work supported by the National Science Foundation under grant no: 17-422-42. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Table of Contents

Acknowledgements.....	ii
Table of Contents.....	iii
List of Figures.....	vii
List of Tables.....	ix
Abstracts.....	xx
Chapter 1. Introduction	1
Chapter 2. Related Literature	7
2.1 Educational use of block based programs	7
2.2 Programming environments for BVI students	9
2.3 Haptic and tactile perception	13
Chapter 3. Tangible Code Block Design, Experimental and Analysis Details	15
3.1 Overview of Scratch	15
3.2 Project Overview	17
3.3 Tangible Code Block Design	18
3.3.1 Design Scope	18
3.3.2 Design Approach	19
3.3.3 Basic Block Design	20
3.3.4 Valid Operator Syntax	23
3.3.5 Nesting	26

3.4. Experimental and Analysis Details	28
3.4.1 Code Concepts Used	28
3.4.2 Experimental Set-Up	29
3.4.3 Participants	30
3.4.4 Experimental Design	30
3.4.5 Measured Metrics	32
3.4.6 Experimental Procedure	33
3.4.7 Statistical Analysis	40
3.5 Results and Discussion	40
3.5.1 Building Simple Expressions	40
3.5.2 Debugging Simple Expressions	43
3.5.3 User Survey After Tasks involving Simple Expressions	45
3.5.4 Building Complex Expressions	47
3.5.5 Debugging Complex Expressions	48
3.5.7 Limitations	52
3.6 Conclusions	53
 Chapter 4. Code Block Design-Further Considerations for Single Line Code Commands..	55
4.1 Introduction	55
4.1.1 Previous Work	57
4.2 Objectives	60
4.3 Study 1 and 2: Using Magnets for Connections	61
4.3.1 Code Blocks: General Design	61

4.3.2 Code Blocks: Studies 1 and 2	62
4.3.3 Participants	64
4.3.4 Experimental Design	65
4.3.5 Experimental Procedure	65
4.3.6 Statistical Analysis	65
4.4 Results: Study 1 and 2	66
4.4.1 Study 1	66
4.4.2 Study 2	68
4.5 Discussion: Study 1 and 2	70
4.6 Study 3: Using Stoppers with Magnets	72
4.6.1 Code Blocks: Study 3	72
4.6.2 Participants	74
4.6.3 Experimental Design	74
4.6.4 Experimental Procedure and Statistical Analysis:	
75	
4.7 Results: Study 3	75
4.8 Discussion: Study 3	77
4.9 General Discussion	78
4.10 Extensions to Other Applications	80
4.11 Conclusions	81
Chapter 5. Code Block Design: Non-Operator Blocks	83
Chapter 6. Assessment of Overall System.	86

6.1 Introduction	86
6.2 Participants	86
6.3. Classroom Set-Up	87
6.4 Curriculum	93
6.5 Data Collection	95
6.6 Assessment of Audiovisual Data and Exit Interview	96
6.7 Results	99
6.7.1 Student 1: Performance.	101
6.7.2 Student 2: Performance	123
6.8 Discussion	132
6.9 Limitations	135
6.10 Conclusion	138
Chapter 7. Conclusions	140
7.1 Future work.	143
References	146
Appendix A	150
Appendix B	156

List of Figures

Figure 0: Overview of System.	5
Figure 1. Scratch UI.	16
Figure 2. Scratch block that connects vertically.. . . .	17
Figure 3. Scratch relational operator.	19
Figure 4. Example of Scratch less than block.	19
Figure 5. Example variable blocks.. . . .	21
Figure. 6. Example operator blocks.	22
Figure. 7 Example of a binary operator (+) and unary operator (NOT).	25
Figure. 8. Top: correct expression, bottom: incorrect expression.. . . .	45
Figure 9. Scratch Code.	55
Figure 10. Overview Diagram.	56
Figure 11. Code blocks showing syntax connections.. . . .	59
Figure 12. “Visual” Scratch set code block.	60
Figure 13. Standard sized variable blocks. 2: Numeric variable. 3: Boolean variable..	62
Figure 14. Logical not operator.	63
Figure 15. The left-hand side of the “set” blocks created for this experiment.	74
Figure 16. General view of the design and implementation of blocks with syntax exceptions.	74
Figure 17. If-then block (left) and example code structure if (v<w) then...(right).	84
Figure 18. Overview of the system	86
Figure 19. Code editor work surface	88
Figure 20. Overview of the drawer organization.	89
Figure 21. Examples of number blocks.	89

Figure 22. The robot used in this study.	90
Figure 23. Robot placed in the background with a coordinate system.	91
Figure 24. The final project of day 1 of the camp.	91
Figure 25. A background used for the first part of the second day.	91
Figure 26. The background for the final project, which is an obstacle course.	92
Figure 27. Example of sequential code (left) and nested code (right).	99
Figure 28. Example of a student identifying a block assembly by feeling and identifying the symbol on the first block with their left hand.	102
Figure 29. Example of a student identifying a block by feeling the unique shape of an event block along the upper edge with their left hand.	102
Figure 30. Orienting the block by feeling for the raised dot located in the top left corner with left thumb	104
Figure 31. Example of feeling symbols of blocks on the bin.	106
Figure 32. The student checked the location of a jut using her left hand	110
Figure 33. Example of placing block successfully	112
Figure 34. Example of block getting stuck on the telescope and student unable to contract the blocks	113
Figure 35. Example of Fails to contract telescope.	114
Figure 36. Connecting block.	115
Figure 37. Disconnecting the blocks (left) and Replacing the Block (right)	116
Figure 38. A student searching for the block in workspace by feeling symbol trial and error...	117

List of Tables

Table 1. User Experience Survey	34
Table 2. Marginal means for models of metrics with standard error	41
Table 3. Marginal means for models of metrics with standard error	43
Table 4. Marginal means for models of metrics with standard error	46
Table 5. Marginal means for models of metrics with standard error	47
Table 6. Marginal means for models of metrics with standard error	49
Table 7. Marginal means for models of metrics with standard error	51
Table 8. Block configurations for conditions for Study 8.	64
Table 9. Study 1: Mean results re magnetic strength and block size (N = 160)	67
Table 10. Study 1: Tests of Model Effects for Ease of Use (N = 160)	68
Table 11. Study 2: Mean results re magnetic strength and block weight (N = 160)	69
Table 12. Study 2: Tests of Model Effects for Ease of Use (N = 160)	70
Table 13. Block configurations for conditions for Study 3.	73
Table 14. Study 3: Mean results re design type and stopper length (N = 80)	76
Table 15. Study 3: Tests of Model Effects for Ease of Use (N = 80)	76
Table 16. Interview Questions	98
Table 17. Student 1: Block identification methods and performance of each day of the camp...	103
Table 18. Student 1: Symbols identified incorrectly and asked the instructor for identification.	104
Table 19. Student 1: Methods and performance of orienting blocks	106
Table 20. Student 1: Method and performance of finding blocks in organization bins.	108
Table 21. Student 1: Performance of bringing blocks from the bin to workspace	109
Table 22. Student 1: Methods of identifying connections	111

Table 23. Student 1: Methods of detecting connections	111
Table 24. Student 1: Performance in adjusting telescoping for the to be inserted block/block assembly	114
Table 25. Student 1: Performance in connecting block	115
Table 26. Student 1: Performance in disconnecting and replacing block	116
Table 27. Student 1: Methods and performance of finding blocks on workspace	118
Table 28. Behaviors of students that used to identify the pattern.	119
Table 29. Sequence of behavior by student 1 during Day 1 and Day 2.	119
Table 30. Student 1: Sequences of behavior when “place” is the second behavior	120
Table 31. Student 1: Sequences of behavior when “bring to table” is the first behavior	120
Table 32. Student 1: Emotions displayed during each day of the camp	122
Table 33. Student 2: Block identification methods and performance of each day of the camp ..	124
Table 34. Student 2: Symbols identified incorrectly and asked the instructor for identification.	124
Table 35. Student 2: Methods and performance of orienting blocks..	124
Table 36. Student 2: Method and performance of finding blocks in organization bin	125
Table 37. Student 2: Performance of bringing blocks from the bin to workspace	125
Table 38. Student 2: Methods of identifying connections	126
Table 39. Student 2: Methods of detecting connections	126
Table 40. Student 2: Performance in placing blocks on telescope	127
Table 41. Student 2: Performances in connecting block	127
Table 42. Student 2: Performance in disconnecting and replacing block	127
Table 43. Student 2: Methods and performance of finding block on workspace	129
Table 44. Sequence of behavior by student 2 during Day 1 and Day 2.	129
Table 45. Student 2: Sequence of behavior when “place” is a second behavior	130

Table 46. Student 2: Sequence of behavior when “bring to table” is the first behavior	130
Table 47. Student 2: Emotions displayed during each day of the camp	131

Abstract

Development of Tangible Blocks for Blind and Visually Impaired

By Hyun Woo Kim, B.S.

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in in Biomedical Engineering at Virginia Commonwealth University

Virginia Commonwealth University, 2021

Major Director: Dianne T. V. Pawluk, Associate Professor, Department of Biomedical Engineering

The fields of Science, Technology, Engineering, and Mathematics (STEM) have been growing at an accelerating rate in recent times. Knowing how to program has become one key skill for entering all of these STEM fields. However, many students find programming difficult. The block based programming language, Scratch, was specifically designed to lower hurdles to learning how to program for sighted students. Unfortunately, although very effective and widely used in K12 classrooms, Scratch, similar to other block based languages, is inaccessible to students who are blind and visually impaired (BVI). This thesis is part of a larger project to make the Scratch environment accessible to BVI students. The focus of this thesis is on creating a tangible code block design that: 1) is accessible to BVIs, 2) retains the reduced need to struggle with syntax of Scratch, 3) allows code construction through action, 4) and co-construction with other BVI and sighted students, and 5) can create moderately sized programs at low cost.

The first several parts of this thesis consider the design and assessment process for the code blocks, which went through two iterations. The four major components of the first design iteration were: 1) the use of passive blocks, with use of 2) the local edge shape connectivity between blocks defining the program syntax, 3) telescoping tubing to allow nested expressions

when valid, and 4) haptically legible commands for both Braille and non-Braille users. The first iteration of the block design was compared to a text based method in building and correcting operator expressions that included both simple and nested expressions of the arithmetic, relational and logical operators. BVI participants produced correct code significantly more when doing the tasks with the code blocks than with the text method. Although the text method was faster, it did not account for any additional time that would be needed to identify and change incorrect code before a program could be run.

One weakness of the first iteration was that it was difficult for BVI participants to easily determine connectivity between validly connecting code blocks. The second design iteration considered the effect of embedding different degrees of magnetic attraction within the local shape connection to improve identification of the connectivity. It also considered how to represent some commands that had additional restrictions to those found with most of the other code block types. In particular, we considered the use of different “stopper” designs to prevent numeric literals from being placed in the left slot of a “set” command, which could only accept a variable. Results from a set of studies evaluating the ability of BVI participants to identify the connectivity between blocks found that the magnetic attraction within the connection significantly improved accuracy and ease of use, with the stronger magnetic connections preferred. They also found that a stopper design could be used for “exceptions”, with the longer stopper aligned with the local connection preferred.

The final part of the thesis examines the use of the code blocks by the targeted population (BVI students in middle school) in a classroom setting within the context of the entire nonvisual interface. To do this, two day code camps were conducted with BVI middle school students, and recorded on video and audio. Qualitative content analysis was used to verify that the students

interacted with the system as intended by the code block design. Results suggest that the students did interact with the code blocks as intended by the design, but minor improvements should be made to increase their ease of use. Participants did appear to have a positive experience with the code blocks and the system overall.

Chapter 1: Introduction

The importance of knowing how to program code has been growing not just in the field of computer science, but also in other STEM fields. However, due to the difficulty many people experience in learning to code, it is hard to get pre-college students interested in coding and there is a low attrition rate in college computer science classes (Mason, 2005). The importance of STEM education can be seen by the fact that nine out of ten companies with the highest market capitalization in the US fall under the STEM category. This impacts not only the employment opportunities of individuals but the economic strength of a country. The effect is likely to be even more in the future as the STEM fields have the fastest growth rate among all other occupations, with an estimated 13 percent growth rate from 2020 to 2030 (U.S bureau of labor statistics).

Students who are blind or visually impaired (BVI) face even more difficulties in learning how to program code than sighted students because of the barriers created by inaccessible development environments and lack of accessible educational tools (Ladner, 2015). Unfortunately, the current employment rate of BVIs who are 21 to 64 years old is already only at 45.4 percent (in 2018), which is significantly lower than the 80.0 percent employment for their sighted counterparts (ASC 2018). This difference will only increase in the future if BVIs are unable to participate in STEM fields due to these barriers. It is, therefore, essential to provide the opportunities and education necessary for BVI students to be involved in fields with promising prospects and lessen the gap in the employment rates that currently exists.

To increase the number of students interested in STEM fields, tools have been successfully developed to broaden the participation of sighted students in learning how to program computers. The Scratch programming environment, which focuses on lowering the

hurdles to learning how to program and sparking interest in programming by novice programmers has been very successfully implemented and deployed in schools, coding clubs and camps, and even colleges. The developers successfully incorporated three key concepts to achieve these goals in a straightforward environment using drag and drop block construction: (1) a method that significantly reduces the need for students to struggle with syntax; (2) learning through action and co-construction with others (from constructivist theory); and (3) enabling the ability to tinker. Reducing the need to struggle with syntax and enabling the ability to tinker also increases the liveness of the environment (the speed at which changes to the code or reflected in the output display), which is considered valuable in education. However, Scratch is also a highly visual language that requires students to drag and drop elements to construct programs, making it inaccessible to BVI students.

There have been many attempts to make block based programming accessible to BVI students. JBrick provides an alternate text based interface for programming Lego Mindstorms (Ludi and Jordan, 2015); however, this greatly increases the focus on syntax. Blocks4All uses nested audio menus and audio cueing (based on spatial location), although the number of commands are small, the program layout linear and users still need to “drag and drop” blocks (Milne and Ladner, 2018). Other researchers have used tangibles, physical objects, to represent program constructs. Tile or block based tangible program languages have been created that assemble passive program blocks together by connecting them together or placing them in orderly slots (Koushik et al., 2019; Sabuncuoglu, 2020); however, although these languages often use shape to distinguish different code commands or command types, they do nothing to reduce the need for syntax and typically only implement a small subset of program concepts. Two groups (Morrison et al., 2019,2020; Rong et al., 2020) have developed methods using active

programming blocks (i.e., ones with embedded electronics). Some design aspects, such as the use of dials and sliders to set parameters of commands, do reduce the need for users to deal with syntax, but not nearly to the degree of Scratch. Also, the embedded electronics are costly if a larger number of blocks (such as for a medium sized program) is needed.

A recent focus group with IT and special needs educators has also provided feedback as to how block based programs would be best represented for visually impaired students: the tangible blocks should be light and easy to grasp, easy to differentiate but not with too much detail; the use of a confined space for programming, with potentially space to store and organize blocks and not in use (Pires et al., 2020).

The approach by our lab is to develop a nonvisual interface for the Scratch environment that provides the same benefits to BVI students as their sighted peers, and allows BVI students to learn alongside and in collaboration with sighted students. The choice to build on Scratch is based on its thoughtful design, its popularity and the fact that it uses the Blockly library as a basis, as does many other block based programming languages (BlocklyTalky, ArduBlockly and OzoBlockly, (Deitrick et al, 2014)) popular in schools, computer camps and computer clubs. The concepts that are to be maintained in the nonvisual environment are: 1) the design of a straightforward environment (in this case, further taking into account the differences between haptics and vision), 2) a program construction method that significantly reduces the need for students to struggle with syntax (further taking into account differences between haptics and vision, and physical and virtual environments), 3) learning through action and co-construction with others (noting that most BVI students go to mainstream schools), and 4) tinkerability and liveliness. Two other requirements have been added: 1) the system must be low cost to be

affordable by classrooms and computer clubs, and 2) the system should be able to create programs of moderate size (appropriate for middle school classrooms, our target audience).

The purpose of the initial team project is to design a tangible interface for Scratch that will consist of: (1) a tangible code block editor, (2) a nonvisual stage for displaying program execution and (3) an automated method to translate the code blocks constructed into code into executable scenarios on the nonvisual stage. This project is shared between myself and Bryson Goolsby, with the division of work shown in Figure 0.

My component of the project primarily focuses on implementing two concepts that exist in visual Scratch. The first is a method to significantly reduce the need of students to struggle with syntax. This is of significant concern for novice coders, who become easily frustrated with the need to focus on syntax. In Scratch, only blocks that belong together can validly be connected. In fact, users do not encounter any error messages. The second is a method to allow nesting in expressions for parameters and operands, and expansion of the statement component of control blocks from a single to multiple statements. Scratch allows these components to grow while still being enclosed by the connectivity that correctly defines its syntax. The difficulty in translating these concepts to a tangible environment are: 1) the perceptibility of many of the visual cues are not effective when only haptics is used, and 2) several actions can occur in virtual environments that cannot occur in physical environments.

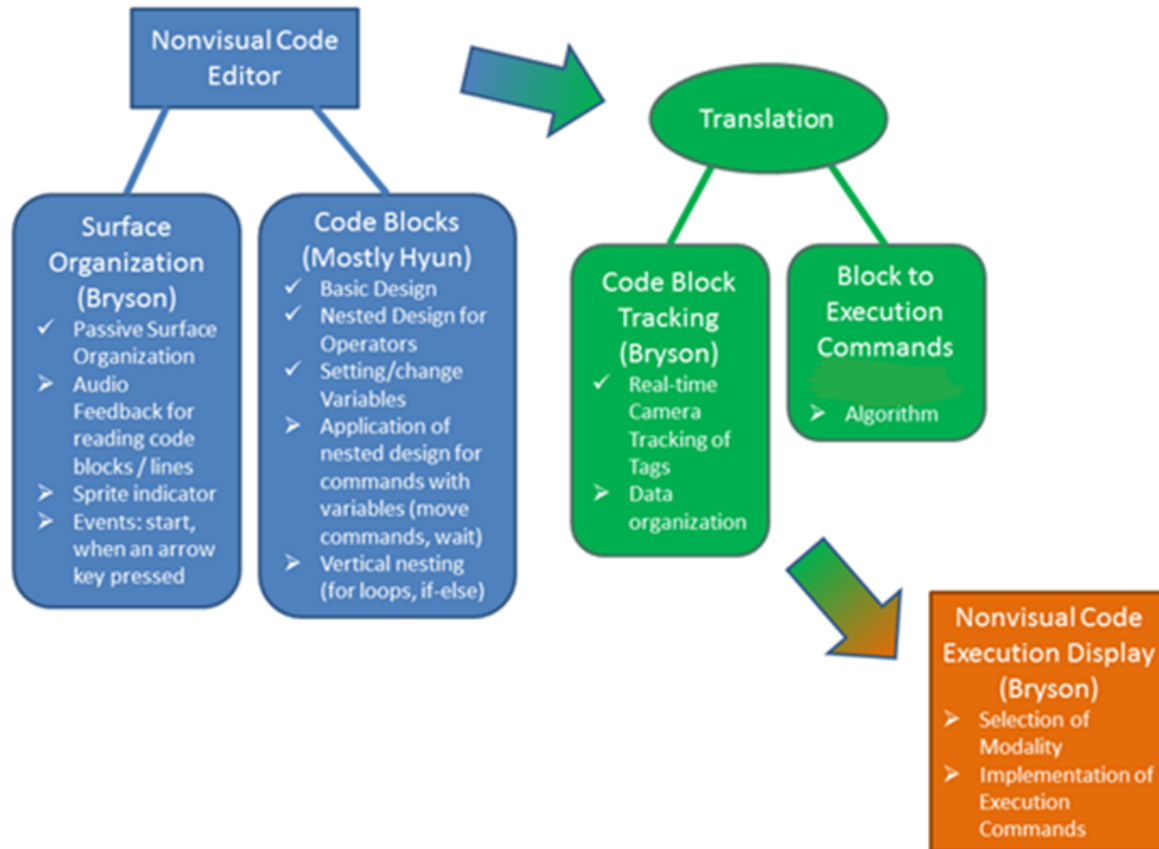


Figure 0: Overview of System

This thesis will first review related literature that includes a deeper dive into the current state of research developing and assessing programming environments for BVIs, as well as relevant information for nonvisual tangible block design, including basic perceptual research in touch and haptics. Then, the majority of the thesis will describe the iterative design of the code blocks for the nonvisual editor, including their assessment through human studies with BVIs. The aim was to develop design principles that reduce the focus on syntax for horizontally nested operator expressions that can then be extended to other program commands and vertically nested code. The design and assessment process was made using two design iterations. The first iteration focused on implementation of the initial design goals. The second iteration was used to eliminate weaknesses in the design that were observed during the first human subjects experiment, as well as consideration of a method that could incorporate exceptions to the

standard syntax into the code block designs. Next, the extension of these concepts to other programming commands and vertical nesting will be described. Finally, the use of the code blocks by the targeted population (BVI students in middle school) in a classroom setting within the context of the entire nonvisual interface will be examined to verify that the code blocks are used as intended and recommend any additional improvements.

Chapter 2. Related Literature

2.1 Educational use of block based programs

Many studies that consider the effectiveness of block-based programming for sighted students are framed by Jean Piaget's constructivism theory of cognitive development, which argues that individuals construct their understanding of the world through their experiences and interactions. According to Piaget, children's cognitive development depends heavily on their active engagement in testing and revising their understanding of the world or objects (Piaget, 1973). Seymour Papert's constructionism is also considered another key foundation for learning through the use of tangible objects. While constructionism shares similarities with constructivism in that learning is through action and experience, Piaget's theory emphasizes cognitive development, while Papert's emphasizes the actual learning process (Papert, 1993). Papert believed that students learn through sharing their constructions with other students and interacting with them. Additionally, constructionism places a strong focus on the use of technology, such as computers, as a tool to enhance learning. Despite the differences, both Piaget's constructivism and Papert's constructionism provide a strong theoretical foundation for the effectiveness of block-based programming and tangible block programming in education.

Many different institutions have developed and deployed Scratch-based outreach programs for middle school students. Many studies exist that show that Scratch increases interest in computing science and that students learn computing science content. For example, a study conducted with 5th and 6th grade students in Spain evaluated the use of Scratch in a classroom setting over the course of two academic years (Sáez-López, Román-González, & Vázquez-Cano, 2016). The students were taught computational concepts and practices, such as sequencing, looping, and user interface design, through 21 hour-long sessions. Results showed

significant improvements in students' understanding of programming concepts, logic, and computational practices. Additionally, observations indicated that students were engaged and motivated by the sessions. The study concluded that implementing programming education using visual block programming languages like Scratch would be beneficial during primary education.

Additional research studied the efficacy of block based programming combined with robotics. One such study, conducted by Koca and Cakir (2022), compared the use of Scratch, with its virtual output, to the use of the block based programming of LEGO-Mindstorm robots. The study was conducted for 2 hours a week for a one semester period with 5th and 6th grade students, which were divided into two groups: One group was taught Scratch and the other were taught LEGO-Mindstorm. After the semester was finished, the changes in students' reflective thinking skill towards problem solving, academic motivation, spatial visualization, and mental rotation from pre-test to post-test were analyzed. The result showed that the LEGO-Mindstorm group performed significantly better post-test than the pre-test on reflective thinking skill towards problem solving, spatial visualization, and mental rotation, whereas the Scratch group improved in terms of the spatial visualization and mental rotation, but not reflective thinking and academic motivation. Comparing two groups, the LEGO-Mindstorm group were significantly better on reflective thinking skills, and spatial visualization compared to the Scratch group. The study concludes that robotic coding education such as LEGO-Mindstorm not only helps students to improve their problem solving and reflective thinking skills, but also improves spatial visualization a lot more than Scratch. This study shows use of robotics improves the outcome of the computer programming education to the young students than just using digital block based programming.

2.2 Programming environments for BVI students

Unfortunately, visual computer programming environments are not accessible for BVI students. Block based programming languages rely on drag-and-drop methods to construct programs which are not conducive to screen readers. Text based programming languages exist in complicated GUI environments that are cumbersome to navigate with a screen reader and with visual debugging cues unavailable. The languages themselves can be difficult to navigate due to the use of many non-alphanumeric characters and it can be hard for a BVI programmer to understand the overall structure of the program (Loomis, 1981; Van Boven and Johnson, 1994). To overcome these problems, there has been a lot of research to make programming environments accessible to BVI students. Several approaches have been used: 1) make text based coding environments accessible, 2) turn block based coding environments into text based coding environments and make them accessible, 3) make block based coding environments accessible through the use of a screen reader combined with a touch screen, and 4) make a tangible environment with physical correlates to the virtual blocks in a block based language.

Quorum (Stefik et al. 2011) is probably the best example of making text based programming environments more accessible to BVI students. It is a text based coding language that greatly reduces the syntax that exists in traditional coding languages. Its user interface is simple compared to typical coding environments and is compatible with screen readers, which allows BVI students to more easily use the platform. However, it does require users to enter text and can only present the code in a linear fashion due to the screen reader output. Although it does simplify the syntax, users must still master the syntax to be able to run a program. It is also not as tinkerable and lively as some block languages. Another problem is that using Quorum

would require BVI students to use a language which is not likely to be used by their sighted classmates: separate but equal is not a solution.

As block based programming environments are now almost exclusively used to introduce students to computer programming and its concepts, there has been considerable effort to make these environments accessible to BVI students. Jbrick is an attempt to make LEGO Mindstorm's block based programming accessible by providing a text based language alternative (Ludi and Jordan, 2015). Its environment was designed to be more accessible to screen readers, allowing the speaking of the programmed text as well as various audio cues to provide feedback during environment navigation and robot execution. Unfortunately, in turning the block based programming into text based programming, it lost many of the advantages that block based programming has for programming novices.

Blocks4All (Milne and Ladner, 2018) uses the approach of adding a touchscreen with voice over and audio feedback to a virtual block based environment. The method maintains the drag and drop concept but with audio instructions and feedback, having the selection of the block and selection of the location separated (i.e., eliminating the actual drag), and collapsing the spatial dimensions to have users construct the code in a line along the bottom of the screen. However, it greatly reduces the number of program commands in the menu and simplifies their representations. This method is fine for small programs useful for teaching younger students, but would be difficult to use for more complex programs one would expect students to be taught in middle schools. This method also eliminates a lot of the implicit cuing (e.g., connection shapes) that occurs in block programming environments like Scratch, which greatly aids with program instruction.

The two main approaches to making tangible code blocks for block programming environments are to use either active blocks or passive blocks. With active blocks, each block contains embedded electronics to control the response of the block, respond to any input components (such as dials, sliders or switches), and provide any output (such as sound or light). Passive blocks have no embedded electronics and are, instead, visually tracked by a camera. The blocks usually have markers placed on them to facilitate tracking. Active blocks are more responsive individually, but passive blocks are more affordable and can, therefore, more easily be used to construct larger programs. Many different tangible blocks environments have been created but few have provided accessibility for BVI students, let alone be designed with them in mind. However, some of the ideas for tangible code blocks for sighted students, such as the use of dials and other input devices (Blinkstein, 2016), the indication of some functions by block shape, such as if then statements by a branching Y shape blocks (Horn and Jacob, 2007), and magnetics (Bdeir, 2009) are potentially useful for BVI students as well.

Both project Torino (Thieme et al., 2017) and CodeRhythm (Rong et al., 2020) created tangible block coding environments specifically designed for BVI students using active tangible blocks. In Project Torino, pods (i.e., blocks) are connected by plugs and wires to form programs, with knobs and dials on pods adjustable to set parameters. Each pod is designed to be tactually distinct through tactile, audio and visual feedback. “Play” pods will also output sound when the assembled code is run, although twisting parameter dials provide an audio response if the pod is plugged in. It is designed for liveliness and a reduced need to focus on syntax, but at the expense of a reduction in the concepts taught. The size of a program is also limited by both the cost of the electronics and the use of cables to connect blocks, which can make it difficult to follow code for

more complex programs. This is fine for young children, but more problematic for our target group of middle schoolers.

In Code Rhythm (Rong et al., 2020), blocks that represented notes of an octave could be magnetically connected together in a horizontal sequence. To this, internal cables in pre-made block assemblies were used to define switch statements and for loops. Preliminary feedback suggests the magnets were helpful, although it was unclear how easily distinguishable the tactile symbols for each of the block types were from each other. However, again, the limitation of the complexity of the concepts that can be implemented due to design choices and cost limits its utility to be used at a middle school level. Additionally, the block design does not embed the syntax connectivity, every block can connect to each other. This means users have to be aware of which blocks are connected to which block constantly to avoid errors.

StoryBlocks (Koushik et al, 2019) uses passive tile-like blocks with fiducial markers to assemble program-like stories. A camera, placed above the workspace, is used to translate the tags into executable code. The shape of the block corners indicate the type of block (3 types) and block symbols are based on the open source EmojiOne characters. Similar to Code Rhythm, internal cables between blocks were used to define an if-else command. The use of passive blocks makes it easier to potentially generate larger programs, while keeping the costs low. However, again the design limited the complexity of the concepts that can be implemented, including operator expressions and variables. There is a need for tangible programming blocks that are passive blocks, so it can be produced at lower cost, and also can generate larger and complex programs, to teach more complex coding concepts.

2.3 Haptic and tactile perception

The quickest manual exploratory procedures used to perceive objects haptically are static contact or an enclosure grasp (Lederman and Klatzky, 2009). The most salient properties in identifying unfamiliar objects are material properties, which can be picked up through a quick “haptic glance” with one of these procedures. In contrast, geometric properties need to be identified by contour following which is slow and memory intensive. When possible, designed tangible objects should be able to be covered in a single static contact with the fingers or an enclosed grasp.

In a target identification task (Lederman and Klatzky, 1993), material properties such as texture, thermal conductivity and compliance, as well as abrupt discontinuities, such as edge versus no edge, were able to be processed across fingers quickly and in parallel. More detailed geometry (even as simple as distinguishing a horizontal from a vertical line) was processed more slowly and serially across the fingers. This again suggests that material properties, along with abrupt discontinuities, will be most effective for distinguishing between different tangibles. Caution was noted, in that properties that are normally processed in parallel can begin to be processed in series if the discrimination between tangibles along the dimension becomes smaller. Properties along a dimension should be varied as much as possible to ensure this does not happen.

If raised reliefs are to be used because of the ease of interpretation of a symbol without training, simplicity is key (Edman, 1992). This needs to take into account the limits to tactile resolution and the more limited field of view of touch as compared to vision. The spatial resolution of touch is approximately 1mm (van Boven et al., 1994) and has been described as a low pass filter of vision (Loomis, 1981). For geometric information more complex than detecting

abrupt discontinuities, the field of view is only the size of a finger pad (Lederman and Klatzky, 1993).

Chapter 3. Tangible Code Block Design, Experimental and Analysis Details

3.1 Overview of Scratch

Scratch is an online block coding environment that is contained in one window. The most relevant parts for our project are the Scratch Code Editor (Figure 1, left) and the Stage (Figure 1, right). We are not considering, at this time, the tabs that contain elements to make costumes or sounds for the Sprites. The Code Editor is divided into three main sections: 1) a section where users can select the category of the block (Figure 1a), 2) a section that shows blocks that fall into the category (Figure 1b), and 3) a workspace where users can drag and drop blocks (Figure 1c) and create programs. The code blocks are colored based on the category that they are in. To create a program, blocks are dragged from their list and placed on the workspace. Blocks need to be connected to be valid, led by an event they respond to. To encourage tinkering: other blocks can be left on the workspace, but will not run; and blocks and parameters can be changed while the program is running.

The visual Stage does the animation of the executed program. The default Sprite is the Cat shown. Below it, one can follow the exact coordinates that a Sprite is at. Multiple Sprites can be used, but, at this time, we are only considering one Sprite. Execution of code happens immediately when the green flag is clicked, again to encourage liveliness of the program.

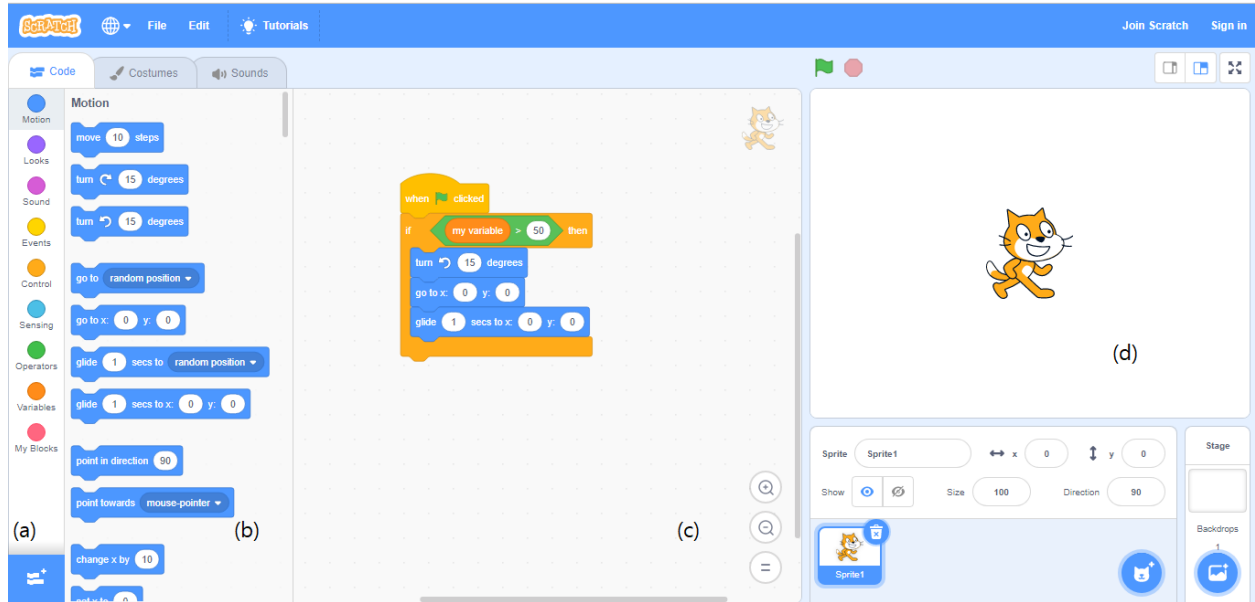


Figure 1. Scratch UI. On the Far left (a) is where user can select category of the block, (b) is where user can fine blocks that fall into such category, (c) is where user can drag and drop the blocks to create program, and (d) is where user can see the output of the code by observing the actions of the cat (Sprite).

The design of the Scratch blocks reduces the burden on syntax. For vertical connections, blocks can only be connected if the juts and notches match up (Figure 2). Note that in Figure 2a, a code block cannot be placed above an event block as an event must always start a piece of code. Code pieces without an event to start them will not run even if they are validly connected together. For parameters and operands, syntax is defined by either a hexagon (b, must insert only blocks that produce a Boolean value) or a circle (c, the blocks can produce any number). The hexagonal shape only accepts blocks whose left and right ends match the hexagon. Circular shapes can only accept blocks that have either circular or hexagonal ends; the shape will morph into a hexagon if a block expression with hexagonal ends is inserted. If the user tries to put a block type into an input shape that it does not fit, the editor actually “throws” the block into another part of the workspace.

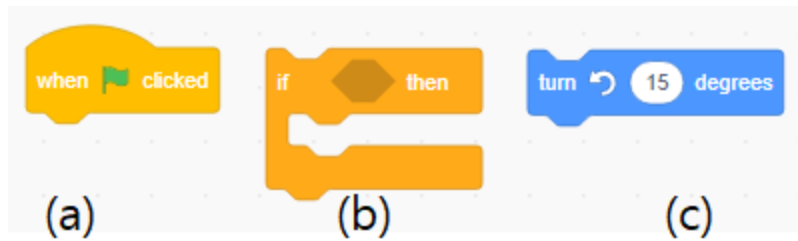


Figure 2. Scratch block that connects vertically. (a) is block type that can only be connected vertically on the bottom, (b) type can connect top, bottom, and nest blocks, and (c) is block type that can connect both on top and bottom.

It is important that all these characteristics to low the hurdles to programming and attract student interest in coding are retained for BVI students if they are to have the same opportunities as sighted students.

3.2 Project Overview

The purpose of this project is to design a tangible interface for Scratch that will consist of: (1) a tangible code block editor, (2) a nonvisual stage for displaying program execution and (3) an automated method to translate the code blocks constructed into code into executable scenarios on the nonvisual stage. In addition to retaining the key concepts incorporated into Scratch, the developed system must also be low cost and be able to produce programs of moderate size for K-12 instructional purposes. Code blocks will be constructed into programs on a tabletop workspace that is easily within reach for most users. Users should be able to sit at a table alone or in a group to construct programs in class and follow instructions given by teachers. To allow for tinkering, code blocks should be allowed to be left on the workspace even if they do not form part of the program being constructed. Only the code blocks correctly constructed for the program should be executed when the student(s) want to “run” the program. Execution must be done through nonvisual means and in a way that all students are able to follow code execution.

3.3 Tangible Code Block Design

3.3.1 Design Scope. A key component of Scratch is reducing the focus on syntax by only allowing virtual code blocks to be assembled together if the syntax is correct. This is achieved in many cases by having the shape of the left and right block edges define which type of blocks they can connect with, similar to puzzle pieces. For example, an AND block can only accept operand blocks that have isosceles triangle shapes on their left and right sides (Fig 3, bottom left). This limits its operands to blocks or block assemblies that produce a Boolean value (e.g. Figure. 3, top and middle). In addition, the Scratch code editor forcibly ejects a virtual block from the local work area if the user tries to actively make an incorrect connection. To determine how this could be translated to tangible code blocks, in this work, we focussed on the creation and editing of individual expressions using numeric, relational and logical operators and their valid operands. This allowed us to design and assess a tangible solution for the most difficult elements of this approach, which could then be generalized to other blocks.

In addition to determining an effective overall block design, two key elements to incorporate into the operator and operand tangible blocks to reduce the focus on syntax were: 1) having operators only accept valid types of operands (i.e., either Boolean only, or both numeric and Boolean, depending on the operator) and 2) allowing the nesting of operators. For the first element, in Scratch, oval operand “slots” (e.g., Figure. 3, top) accept both oval (numeric) and hexagonal (Boolean) shaped blocks, whereas hexagonal “slots” (e.g., Figure. 3, middle) accept only hexagonal (Boolean) shaped blocks. For nesting, an operand “slot” will automatically expand if a valid nested operator expression (i.e., one that produces a valid variable type for the “slot”) is inserted (Figure. 4, right). Nesting can be performed recursively.

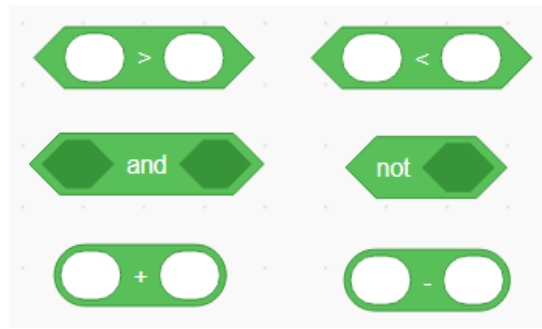


Figure 3. Scratch relational operator (top), logical operator (middle), and mathematical operator (bottom). The external shape represents the type of the output. The shape of the input box represents the type of variable it can accept.



Figure 4. Example of Scratch less than block. It can accept a single variable (left) and also expand to accept an expression (right).

3.3.2 Design Approach. The blocks went through various designs and iterative steps to refine and meet the desired requirements. The feedback from BVI students, their teachers, and rehabilitation professionals were taken into consideration while developing blocks. The prototype blocks were presented to seventeen BVI high school students who were attending 2019 Learning Excellence in Academics Program (LEAP), a summer program that taught BVI students engineering concepts at Virginia Commonwealth University. Designs were modified based on the given feedback from the students. The second prototype blocks were presented to lead technology instructor, Michael Fish, and orientation and mobility instructor, Domonique Lawless, at the Rehab Center of the Virginia Department for the Blind and Visually Impaired (DBVI). The block design was further refined with the comments and feedback. The design was refined until it was believed that current design meets all the requirements and performs desired function. The primary design considerations were: 1) maintaining the reduced focus on syntax that exists in Scratch's virtual code editor, 2) providing the ability for users to construct moderately sized programs (consisting of hundreds of blocks) within arm reach of users aged

eleven and up, 3) making it easy for users to non-visually (and visually) identify the blocks and assemble them with confidence, and 4) keeping the cost low. To produce a set of blocks that was low cost and numbered in the hundreds, it was essential to use passive rather than active blocks, as the cost of embedded electronics quickly adds up for a large number of blocks. In the complete system, the blocks are then tagged and tracked by a camera as they are manipulated by the user within a structured workspace that provides cues for both nonvisual and visual use (Goolsby et al., 2012). However, as the focus of this work is on the operator and operand code blocks, and their connectivity, only their design will be described in further detail.

3.3.3 Basic Block Design. The shapes of the passive blocks were chosen to be rectangular tiles of a standard size. This allows the code to be built in an organized fashion that uses the workspace efficiently. The dimensions chosen for the standard block size was 30mm x 30mm x 15mm, which fits hundreds of blocks within arm reach of a user while allowing the blocks to remain easy to grasp, have an effective tracking tag size and provide legible tactile symbols to identify the command represented by the block. All blocks were 3D printed with an Ultimaker 3 using PLA.

Raised relief symbols on the blocks were used to identify the operator or variable used (Figure. 5 and 6, (a) on the blocks). The size of the symbols was chosen to correspond to a symbol set for which capital letters/braille characters had a height of 8mm, as this size was easiest for blindfolded stakeholders to perceive tactually in pilot testing. A single symbol was used rather than a full word as only a single symbol could: 1) fit on the block at the desired scale, and 2) be determined in a brief contact (as opposed to more time consuming scanning by the fingertip). All raised symbols were also coloured black to provide high contrast between the symbol and its background for low vision users.

Roman letters were used for the variables (e.g., Figure 4) as fewer than 10% of individuals who are blind or visually impaired (BVI), as well as most individuals who are sighted, know Braille. Simplified cartoon shapes of animals were also considered as they were expected to be more appealing to young students. However, pilot testing found these shapes very difficult to perceive when only touch was used due to their complexity. In this study, four variables were created, which were represented by roman letters v, w, x and y. These letters were chosen as they were easy to perceive from each other both tactually and visually.

For the symbols for the operators, we looked beyond the usual computer program symbols as several were difficult to distinguish from each other by touch alone (as needed by our BVI users) in pilot studies. For example, ‘!’ for NOT and ‘|’ for OR were not possible to distinguish. Some problems were also caused by the difficulty in BVI users to precisely orient the blocks, resulting in ‘+’ for addition being confused with ‘x’ for multiplication. To avoid large orientation errors, a raised dot was added in the top left-hand corner of all the blocks as an orientation marker (Figure. 5 and 6, (b)). Smaller orientation errors still occurred and using both ‘+’ and ‘x’ symbols was avoided. Symbols for all the operators were drawn from an extended set of alternative mathematical symbols and chosen for their ease of tactile distinguishability from other operators in pilot testing (e.g. Figure. 6 contains the addition operator and the OR operator).

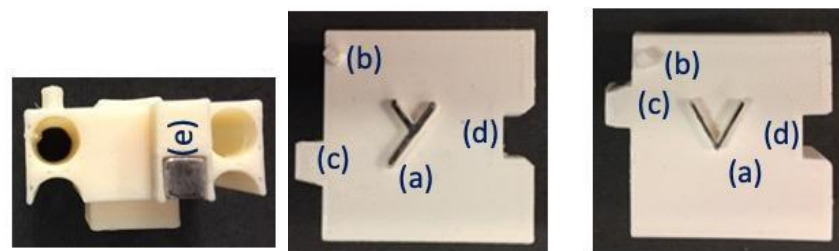


Figure. 5. Example variable blocks. Left image is the left view, the middle and right image is the top view.

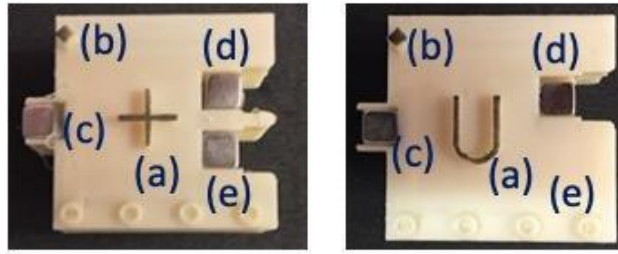


Figure. 6. Example operator blocks. (e) the o shape raised symbol is a texture.

Finally, a texture “strip” at the bottom of a block was used to designate what category of commands the block belonged (e.g., in Figure. 6 The texture indicates an operator and in Figure. 5 the smooth texture indicates a variable), similar to the use of color in the virtual code editor. Texture was chosen rather than another raised symbol as material properties are easier and quicker to determine than shape information (Lederman et al., 1997). It also made the command category easily distinguishable from the command itself. (Texture was not used to represent the actual commands as there were too many to make an easily identifiable set of textures and would be difficult for users to remember the mapping to the commands.) The textures created were made as raised reliefs through the 3D printing process and were based on a subset of the texture palette provided on the Braille Authority of North America website. The texture set chosen was determined through pilot testing with blindfolded stakeholders. The chosen set was also compared to the use of natural textures based on a set of tactile fabric swatches. However, stakeholders strongly preferred the 3D printed textures, which also benefited by the ease of which they could be incorporated into a block.

Color will be used in addition to texture to distinguish code command category in the final blocks to aid those with any usable sight; however, the colors will need to be modified to some degree to allow better contrast for low vision users. For this work the focus was on the presented texture as study participants were blindfolded.

3.3.4 Valid Operator Syntax. Given the above description of the basic block design for operators and variables, operator expressions could be made by a single horizontal line of adjacently connected blocks. However, to ensure valid syntax is enforced through block assembly, a method is needed to be selective about which blocks can be connected. As mentioned previously, in the virtual code editor of Scratch, some operators can accept only Boolean operands and some operators can accept Boolean or regular numeric operands. This is done by creating oval or hexagonal “slots” in the command block to fit an operand. Unfortunately, the virtual oval slot accepts a Boolean operand by morphing into a hexagonal slot when the Boolean value is inserted. This is not something easily implemented tangibly. For our tangible block design, we chose to encode valid syntax by the reciprocal matching of the vertically paired edges of the adjacent blocks. This means that the edge of a Boolean variable would have one shape and the edge of a numeric variable would have a mutually exclusive shape. An operator that accepts only one type of variable would have the reciprocal shape to match that variable. However, an operator that accepts both (e.g., addition) would need to make a clear fit when either are connected.

The two main ways valid syntax could be enforced through edge geometry is through an overall shape match or a more local shape match. Pilot studies with blindfolded stakeholders found that using overall shape match (i.e., an oval reciprocal pair versus a hexagonal reciprocal pair) proved difficult to do correctly, in addition to not addressing the shape morphing requirement. Performance using local shape matching (i.e., the location of a square notch along the edge) was much better. This is consistent with previous perception research that determined that local, abrupt discontinuities are more easily distinguishable than overall shape (Lederman et al., 1997; Lakatos and Marks, 1999). The final geometric design is shown in Figures 5 and 6: the

variable ‘y’ in Figure. 5 is a numeric variable, indicated and enforced by the lower jut on the left edge (c), whereas the variable ‘v’ is a Boolean variable, indicated and enforced by the upper jut on the left edge (c). The addition operator in Figure. 6 can accept both a numeric and Boolean variable as indicated by the two notches (d), whereas the OR operator can only accept a Boolean variable as indicated and by the upper notch (d).

As the correctness of user performance using local shape matching was not as high nor the determination as easy as we would like in pilot testing with blindfolded stakeholders, modifications were made to the blocks to create a “snap to fit” feel. This was performed by placing 5x5x2 mm neodymium rare earth magnets (with a surface gauss of 35234 gauss) of opposite polarity in the notches and juts. This made it much easier for stakeholders to non-visually determine whether the blocks were connected or not. The use of repelling magnets to indicate a connection was not valid, similar to the “ejection” of blocks that do not fit in the virtual code editor, was considered but did not significantly improve the saliency of the feedback or fit with the current design.

One concern with the proposed design was that the location of the notch on an operator block indicated but did not enforce syntax, as blocks could still be assembled if misaligned. A second concern was that the design described the connection between an operator and one operand, despite most operators having two operands. In addition, some Scratch operators have more than two operands. One possible solution to these two problems is to create enclosed “slots” in the command block for the operands, similar to the virtual Scratch environment (Figure. 3). This would also enforce proper alignment of the blocks and their notches by requiring blocks to have an exact vertical edge match. However, this would not allow for nesting of expressions (Section 2.1.5) because the slots cannot expand horizontally. In our design, the

slots are only enclosed horizontally, with copper tubing (Figure. 7) used to align the operand block(s) with the operator by matching the two half cylinder cut-outs on the bottom surface (Figure. 5, left profile). Two copper tubes are used to allow for the largest tracking tag size while preventing any non-uniform flexing of the slot between the lower and upper edges, which could affect interpretation of the edge match.

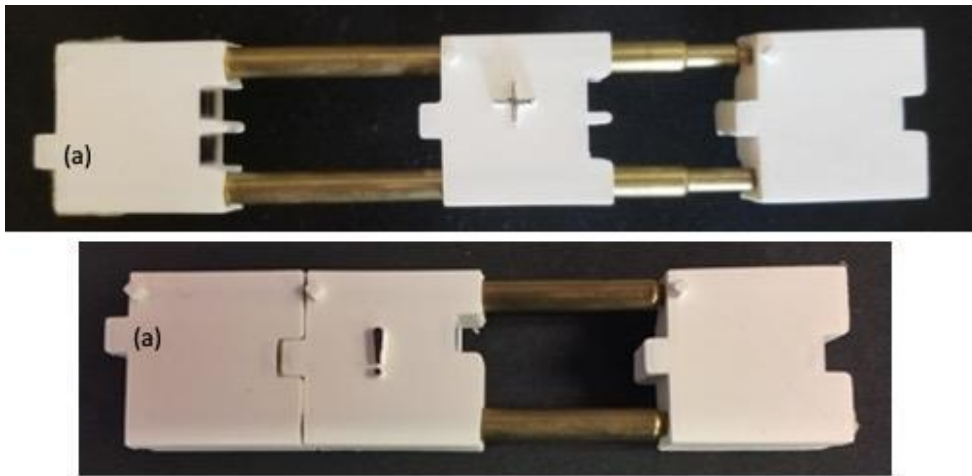


Figure. 7 Example of a binary operator (+) and unary operator (NOT)

The end blocks for each block assembly (Figure. 7) can be thought of as parentheses placed around the operator expression. As the NOT operator does not have a left operand, the left end block is permanently attached to the NOT operator block. The right operand, for all operators, is as mentioned before: where the location of the notch indicates whether a Boolean or regular numeric operand is accepted. For the left operand of any binary operators, the notch/jut pairing occurs between the left end block and the operand block. This design choice was felt to be more consistent in behavior (i.e., easier for users to understand) than to reverse the reciprocal edge matching for the left operand. It also easily expands for commands with more than two parameters.

Finally, in preparation for the use of operator expressions as parameters and to allow nesting, a method is needed to define the output type (Boolean or numeric) of the operator expression. This was specified by the location of the jut on the left end block (Figure. 7, (a)); again, the upper location indicates a Boolean value and the lower location indicates a regular numeric value. In fact, if the operator expression was compressed horizontally into one block, it would have the same juts and notches as an equivalent Boolean/numeric variable.

3.3.5 Nesting. The “slots” for the operands in Scratch’s virtual code editor expand and contract based on the size of the to be inserted expression. They can fit a single variable or expand to fit a complex nested operator expression. To ensure ease of use and a similar coding method for a tangible code editor, this concept needs to be incorporated into the tangible interface design. Several different methods were considered including string with a spindle, elastic string, retractable string with a reel (inspired by Ducasse et al., 2016) and telescoping tubing. Pilot testing was performed with stakeholders to compare these methods.

When using string with a spindle, although it was easy to use and provided useful tangible feedback when expanding a slot, it was difficult to use and sense feedback when contraction was needed due to the difficulty in winding string around a small spindle. This was especially true for BVI users using only haptics. In contrast, the use of elastic string and retractable string were easy to use and provided useful tangible feedback when contracting a slot, but it was difficult to use for expansion. This was because the left end block needed to be pinned down in order for expansion to occur. Although one could do this using suction cups (as in Ducasse et al., 2016), it would make movement of code pieces tedious and cumbersome. In addition, we were concerned about permanent deformation changes in the elasticity of elastic string or a retraction mechanism affecting the ability of such methods to completely contract to

tightly fit a single variable block. The final method, the use of metal telescoping tubing, worked well for both expansion and contraction of a slot, and stakeholders found it easy to manipulate without vision. This method was the one chosen.

As mentioned earlier, the tubing was designed to aid with alignment of the code blocks by matching a half cylinder cut-out on the operand block(s) to the tubing on which it was resting. Two parallel copper tubes were used to ensure that there was minimal flexing between the top and bottom of a slot. Several additional design criteria were used for the telescoping component of the design: (1) the telescoping tubing must be able to collapse to a size to fit a single variable block, (2) it should be able to expand to fit as many blocks as possible within any other constraints, (3) even the thinnest diameter telescoping section must be able to handle the haptic placement of the operand code blocks on top of it without flexing, (4) the tubing, no matter the state of extension, must still allow the operand code blocks to fit together with the operator code blocks through their juts and notches (i.e., one piece cannot sit too far above the other), and (5) the tubing must be able to expand and contract relatively smoothly.

To address design criteria 1, 2 and 4, the telescoping tubing was chosen to be composed of three parts consisting of pairs of outer diameters and lengths, from the outer to inner tube, of: (7/23 inch, 5.5 inch), (5/32 inch, 4.7 inch) and (3/32 inch, 4.3 inch). To address design criteria 3 and 5, the tubing was selected to be made of brass with a thickness of 0.014 inches. For each operator assembly, the left end of the largest tube was attached to the left end block through super glue adhesive, as was the right end of the smallest tube to the right end block. The tube loosely passes through the middle operator block, which provided stability for the tubing and allowed the finite expansion limit to be maximized for a given expression. When fully extended, approximately nine blocks can be shared between all slots.

3.4 Experimental and Analysis Methods

3.4.1 Code Concepts Used. For the user study, a subset of the given operators was used that portrayed all syntax variations that occur: 1) numeric/Boolean versus only Boolean operands, and 2) unary versus binary operators. The specific operators used were: addition and multiplication (from the numeric operators), greater than and less than (from the relational operators), and AND and NOT (from the logic operators). Variables were provided as potential operands, where variables could either be numeric or Boolean in nature. Operator expressions could also be nested to become operands. Only the use of one level of nesting was assessed in this study although the principle could be extended to multiple levels.

Users were asked to do two coding tasks: assemble/code an operator expression and debug an operator expression. In debugging operator expressions, users were required to determine if the expression was invalid and, if so, fix it. The two possible errors that were introduced were: (1) using the wrong number of operands, and (2) using the wrong type of operand. The two coding tasks were first done only with simple expressions: those expressions for which the operand was a variable. They were then done with complex expressions: where an operand may be represented by a variable or a simple expression.

In this study, the use of tangible blocks to perform the coding tasks was compared to the use of typing text on the keyboard. The tangible code blocks with embedded syntax were created as described in Section 2.1, with a complete set of the code blocks used given in Appendix A. Java notation was used for text coded expressions, with study participants told to always use parentheses to enforce order of precedence. The latter was done to reduce the amount of training needed for the text coding task.

3.4.2 Experimental Set-Up. For all studies and methods, participants were blindfolded and seated at a desk with an approximately 30 inch x 48 inch work area. The blocks and block assemblies were organized in a four by three grid of AkroBins plastic storage bin stacking containers that were placed on a small table to the left side of the seated participant.

For the tangible block coding method, two physical copies of the plus, minus, greater than, less than, AND and NOT operator block assemblies were provided to the user, along with two copies of the variable blocks representing Boolean variables x and y , and numeric variables v and w . The top row of the stacked storage bins contained, from the left, the blocks for x and y in a shared bin, and the block for v and w in a shared bin. The bottom two rows contained, from the left, the blocks for each operator in separate bins. Each bin used was labeled with a raised line plate displaying the operator or variable symbol of its contents. For this method, the experimenter sat across the desk from the participant.

For the text coding method, a regular keyboard, which was modified to have a bit of tactile feedback to help with keyboard navigation, was placed immediately in front of the user. The tactile feedback provided were small, adhesive, hemispherical beads attached to the surface of the J key, the left shift key, the right shift key, the backspace key, and the down arrow key on the numeric keypad. The JAWS screen reader, the most common screen reader used by BVIs with a computer, was used to provide audio feedback when typing and reading an expression. The software was used with its factory settings. For this method, the experimenter sat to the side of the participant. This allowed them to share the keyboard to type in the operator expressions to be debugged. The computer's screen was placed so that the experimenter could visibly see what was being typed.

3.4.3 Participants. Nine individuals (5 female, 4 male) who were blind or visually impaired completed the experiment. The age of participants varied from approximately 24 to 64. All participants had vision loss that could not be corrected by prescription lenses, four of whom were completely blind or had light sensitivity only. The conditions leading to blindness were from a variety of issues (diabetes, glaucoma, cataracts, infection, birth defects, etc.). In terms of the content of the study, all participants had negligible to no programming experiences. However, participants had different levels of familiarity with using a computer keyboard before the start of the experiment: some knew where most of the keys were located on the keyboard before the start of the experiment, some knew where a few of the keys were located and some had difficulty locating any keys.

3.4.4 Experimental Design. The experiment was divided into 4 studies. For the first two studies, the participants were required to work with simple operator expressions (Section 2.2.1); and, for the last two studies, the participants were required to work with complex expressions. Participants were required to assemble/code operator expressions spoken by the experimenter in studies one and three. In studies two and four, they were required to debug code that was pre-made by the experimenter. The studies were performed in the same order for all participants. This was because the participants were all novice programmers and training for working with complex expressions builds on that for simple expressions, as does debugging code versus assembling/writing code.

Each participant performed both the tangible block coding method and the text coding method in a counterbalanced within subject design for each pair of construct and debug studies. To increase statistical power in the design, repeated measures were used. In the code construction studies, participants were tested with 6 expressions per study. In the code debugging studies,

participants were tested with 12 expressions per study. To avoid learning effects for a specific expression due to reusing the same expressions for the two different methods in each study, two sets of expressions of approximately similar difficulty were created. The two sets were then randomly assigned to a coding method (without replacement).

The operator expressions used were created algorithmically. The two sets for the first study (constructing simple expressions) both had one expression representing each operator, with the operands for a given operator selected randomly from valid variables (without replacement, if two operands were needed). The order of the operator expressions within each set was randomized for presentation.

The two sets of operator expressions for the second study (debugging simple expressions) both had two expressions representing each operator, with the order of operator expressions within each set randomized for presentation. For each set, three of the twelve expressions (one each from numeric, relational and logical operator expressions) were correct in order to prevent participant bias in always assuming an error occurred. The remaining nine expressions had errors, where an error could consist of the wrong number of operands being used, or one or more of the operands being of incorrect type. For each operator expression, variables were again used without replacement, if two operands were needed.

The two sets of operator expressions for the third study (constructing complex expressions) both had one expression representing each operator (i.e., addition, multiplication, less than, greater than, NOT and AND) as the main operator. For each expression, one of the operands consisted of simple operator expressions and the other was a variable. For binary operators, the location of the nested operator expression was randomly chosen to be on the left or the right of the operator, although the location was balanced across expressions in a set. The

operator for the nested expression was chosen randomly from syntactically correct possibilities (i.e., the resultant was of the correct type for the main operator). Variables used were also chosen randomly from valid variables for the given operator type.

The two sets for the fourth study (debugging complex expressions) were generated by first creating 12 correct expressions for each set, with each operator used twice for the main operator. The operands were determined as described for the third study. Due to the complexity of the errors to be introduced, errors were introduced into the sets by hand. Eight of the expressions were randomly selected to have errors, with the types of errors that could exist equally balanced between: wrong number of operands for the main operator, wrong number of operands for the nested operator, wrong type of one operand for the main operator and wrong type of one operand for the nested operand. Four of the expressions were left correct in order to prevent participant bias in always assuming an error occurred.

3.4.5 Measured Metrics. For studies 1 and 3, involving code construction, the metrics used were: 1) the time for the user to finish writing the given code, indicated by speaking the word “Done”, 2) the number of times participants asked the experimenter to repeat the expression they needed to code, and 3) the correctness of the answer (correct/incorrect). For studies 2 and 4, involving debugging code, the metrics used were: 1) the time it took for participants to determine there was an error or not, indicated by speaking the word “Correct” or “Incorrect”, 2) whether their answer was correct (correct/incorrect), 3) the time (from the beginning of the trial) it took the participant to fix the error, indicated by speaking the word “Done”, 4) the number of times participants asked the experimenter to repeat the intended expression and 5) the correctness of the resulting expression (correct/incorrect).

At the end of the completion of studies with simple expressions with a given method and at the end of the completion of studies with complex expressions with a given method, a user experience survey was conducted (Table 1). Question 1 was meant to be a quick assessment of cognitive load. For the remaining questions, participants were given a statement and were asked to agree or disagree with it on a scale of 1 to 5. The System Usability Scale was used as a basis for the statements. However, statements were modified to evaluate some of the intended goals of the system design, namely: 1) reduction of the difficulty in learning how to program and 2) making learning programming attractive to students who are not interested in programming. Both positive and negative statements (pairs as indicated in the table) were used to evaluate a criterion to reduce answer bias.

3.4.6 Experimental Procedure. The experiment was performed by participants during two three to four hour sessions over the course of two days. Initial training and the studies involving the simple expressions (Studies 1 and 2) were performed in the first session (with the two methods used in random order). Further training and the studies involving the complex expressions (Studies 3 and 4) were performed in the second session (again with the two methods used in random order). Each participant was tested in a private room in the research laboratory to maintain confidentiality. Participants were blindfolded throughout the study to avoid the use of any residual vision. Participants had a rest break between the different methods tested, and also whenever they requested one.

Table 1. User Experience Survey

Question Number	Question/Statement	Answer Scale	Pairing
Q1	What percentage of time were you able to concentrate on programming versus extraneous issues?	0-100	-
S2	I think that using this method would be helpful in learning how to program correctly.	1-5	S7
S3	I thought the method was easy to use correctly.	1-5	S9
S4	I think I would need the support of a technical person to use this method correctly, even after training.	1-5	S8
S5	This method had me doubting whether I understood what I was doing.	1-5	S10
S6	I thought this method made learning how to program fun.	1-5	S11
S7	I thought this method was not very helpful in learning how to program correctly.	1-5	
S8	I would imagine that most people would learn to use this method correctly very quickly.	1-5	
S9	I found the method very cumbersome to use correctly.	1-5	
S10	I felt very confident using the method.	1-5	
S11	If I had to use this method, my enthusiasm for learning how to program would decrease.	1-5	

This research followed the tenets of the World Medical Association Declaration of Helsinki on Ethical Principles for Medical Research Involving Human Subjects. The Institutional Review Board at Virginia Commonwealth University's Office of Research and Innovation (Richmond, Virginia) approved this study.

Day1: Text Coding Method - Simple Expressions, Training and Testing. This part of the session started with participants being trained on using the keyboard and becoming familiarized with the Jaws screen reader. Then participants were trained on how to construct simple expressions with the given operators and variables, followed by testing their ability to correctly

construct simple operator expressions (Study 1). The participants then moved on to practicing and then being tested on their ability to correctly debug simple operator expressions (Study 2). Finally, participants were then asked the user experience survey questions for using the text coding method with simple expressions and for any additional feedback.

The first step of the process was to place the computer keyboard on the table in front of participants and train them to locate the reference keys (the ones with bumps on them: the J key, left shift key, right shift key, backspace key, and down arrow key). Participants were then tasked with finding each of the reference keys when the experimenter asked them in random order; this task was repeated two times. Then, to familiarize participants with the use of the screen reader (JAWS) with the keyboard, participants were asked to type the word “wave” and hear the audio feedback. They were then asked to change the word to “wax,” followed by “vax,” and “vex.” This process was repeated three times to practice erasing and inserting letters with aid of the screen reader.

The next step was to train participants on how to construct simple expressions with the given operators and variables (Study 1). Participants were first trained on how to use the addition and multiplication operators. This started with the experimenter showing participants were to find the symbols on the keyboard that represent the two operators and having participants practice finding them three times. Participants were then introduced to the four variables and instructed that any two numbers could be used with the operator as in regular math. Participants were then asked to construct an expression with each of the operators (e.g., “can you construct the expression for the addition of x and y”). This method of training was subsequently repeated for the less than and greater than operators. It was then repeated for the NOT and AND operators, although certain modifications to the experimenter’s explanation were made: 1) these

operators only worked with binary numbers, 2) only the variables v and w were binary numbers, and 3) the NOT operator only had one input, which was to the right of the operator. After all the operators were presented and practiced, participants were asked to locate all the operators on the keyboard. This process was repeated until they were able to correctly locate the symbols 12 times in a row.

When the training process was finished, participant testing for Study 1 was conducted. For each trial, the experimenter stated the expression to be constructed (e.g., “the addition of x and y”) twice out loud to participants. Participants were then asked to construct the code using the keyboard. Participants were allowed to ask the experimenter as many times as they wanted to further repeat the expression out loud, although the number of requests was recorded. Participants were required to indicate when they were done completing each expression. The relevant metrics were measured for the construction of each expression.

For the debugging code task (Study 2), for each trial, the experimenter typed an operator expression on the keyboard with the screen reader muted. Then, participants were asked to detect whether there was an error in the expression, and to tell the experimenter immediately. If there was an error, participants were asked to further fix it and indicate when they were done. The relevant metrics were measured for the participants component of the trial. A practice set of two expressions (one correct and one incorrect, in random order) was given to the participants for training. The experimenter provided feedback on whether the participant was correct and helped them learn how to debug expressions. Subsequently, testing was conducted but without feedback from the experimenter.

Day1: Tangible Block Coding Method - Simple Expressions, Training and Testing. This part of the session had the same steps for training and testing as the text coding method: 1) a

basic familiarization training for using the blocks, 2) training followed by testing on the construction of simple operator expressions (Study 1), training followed by testing on debugging of operator expressions (Study 2) and then finishing with a user experience survey for the method.

For familiarizing participants with using the tangible code blocks, the experimenter initially focused on the variable blocks. First, the experimenter asked participants to pick one Boolean variable block and one numeric variable block from the first row of the grid of storage bins and place them on the table. Participants were then instructed to pick one block up at a time. When a participant picked up a block, they were taught how to orient the block with respect to the orientation marker, and the difference in the location of a protrusion or cavity for regular and binary numbers. The experimenter also ensured that the participant could separately identify the x, y, v and w variables. Participants were also told that they could leave the variable blocks they explored on the table or put them back in the bins.

Then training on how to construct simple expressions with the given operators and variables began. Participants were first trained on how to use the addition and multiplication operators. The experimenter placed an additional operator assembly on the table, allowing participants to handle it. The experimenters showed participants the different elements of the assemblies and how to properly orient it. Participants were then told where to find the bins containing the addition and multiplication block assembly and were required to practice finding them three times. Participants were then required to place one addition operator assembly and one multiplication operator assembly on the table. They were then shown how both Boolean and numeric variables needed to fit the operators, and how they could tell which variable types were valid. Subsequently, participants were asked to construct a simple expression with each operator.

The experimenter explained how the participants could tell whether the expression was constructed correctly by whether the blocks fit together. This method of training was then repeated for the less than and greater than operators. It was also repeated with the NOT and AND operators, with the experimenter highlighting the differences between these operators to the others through physical differences in the block assemblies. After all the operators were presented and practiced, participants were asked to practice locating all the operator bins. This process was repeated until they were able to correctly locate the symbols 12 times in a row.

When the training process was finished, participant testing for Study 1 was conducted. The same procedure was used as for the text coding method. After each trial, participants were allowed to leave blocks or block assemblies on the table top or put them back in the bins. Training and testing for the code debugging task (Study 2) also used the same procedure as for the text coding method. The main difference was that the experimenter, rather than typing the expression on the keyboard at the beginning of each trial, assembled the operator expression to be debugged. If there was an error, block(s) were missing or not completely connected. After each trial was completed, the experimenter removed the blocks that they used to construct the expression from the workspace.

Day 2: Text Coding Method - Complex Expressions, Training and Testing. The testing and training sessions followed a similar format as for simple expressions on Day 1: general familiarization with the method occurred first, followed by training and testing for Study 3, followed by training and testing for Study 4, and finally finishing with the user experience survey for the method. However, two important differences were made in the training: a) a shortened form of the training for constructing simple expressions was used (as a review), before b) further training in constructing and debugging complex expressions were made.

For training on constructing complex operator expressions, the experimenter explained to participants that a simple operator expression could be used in place of a variable that would normally be used for the main operator. They were also told that they should always put the simple expression replacing the variable in parentheses. The experimenter then led participants through an example (e.g., “the result of w less than y is less than v”). This was followed by participant testing for Study 3, which followed the same procedure and recorded the same metrics as Study 1. The training and testing for debugging complex expressions (Study 4) followed the same procedure as that for simple expressions (Study 2). However, three practice examples were given to participants: one which was correct and two which were incorrect.

Day 2: Tangible Block Coding Method - Complex Expressions, Training and Testing. The testing and training sessions followed a similar format as for simple expressions on Day 1: general familiarization with the method occurred first, followed by training and testing for Study 3, followed by training and testing for Study 4, and finally finishing with the user experience survey for the method. As with the text coding method with complex expressions, the procedure was modified to shorten the training with simple expressions and to provide further training for constructing and debugging complex methods.

For training on constructing complex operator expressions, the experimenter explained to participants that a simple operator block assembly could be used in place of a variable block that would normally be used for the main operator. They were also told that in order to do this the blocks on either side of a “slot” could be pulled apart to provide a larger space to fit the given block assembly. Participants were then asked to extend and contract the telescoping tubing for each of the operand “slots” for a binary operator. The experimenter then led participants through an example (e.g., “the result of w less than y is less than v”). This was followed by participant

testing for Study 3, which followed the same procedure and recorded the same metrics as Study 1. The training and testing for debugging complex expressions (Study 4) followed the same procedure as that for simple expressions (Study 2). However, three practice examples were given to participants: one which was correct and two which were incorrect.

3.4.7 Statistical Analysis. Generalized estimating equations were used to model the outcome metrics. The models included the main effects of Method (Blocks/Screen Reader), Keyboard Experience (Some/Lots) and Degree of Vision Loss (Low Vision/Blind), and all two-way interactions. A compound symmetric matrix was chosen as the correlation matrix because we expected correlation between repeated measures. After the analysis, only the terms that were statistically significant, and, in the case of interaction terms, both main effects were significant, were included in the final model for a given metric. For the results of each study, all metrics that measured time (e.g., response time) and count (e.g., number of time expressions were repeated) were modeled as a Poisson distribution with a log link function. The metric involving accuracy was modeled by a binary logistic response. For the survey results, the fraction of the time the participant thought they spent programming was modeled as a normal distribution with an identity link function. Each positive-negative pair of questions was analyzed individually by first converting the negative question to one that was consistent with the rating for the positive question (i.e., a rating of 1 to 5 was mapped onto a rating of 5 to 1). The question pairs were also modeled as a normal distribution with an identity link function.

3.5 RESULTS AND DISCUSSION

3.5.1 Building Simple Expressions. The means of the two different methods were compared for the time it took to build an expression, the correctness of the created expression

and the number of extra times the experimenter was asked to repeat the word description of the expression (Table 2). For the metric of build time, the main effects of Method ($p < 0.001$) and Familiarity with Keyboard ($p = 0.013$), as well as the interaction effect between the two ($p = 0.002$) were significant. The initial model for correctness only included the main terms as otherwise the model fit did not converge: however, none of the terms were significant. Examining the counts of the observed values suggested that there was a trend for those familiar with using a keyboard to do better with the block coding method than the text coding method, but this was not statistically significant. For similar convergence issues, only the main effects of Method and Familiarity with Keyboard were used for the initial model of the number of times the expression was repeated to the participant: only Familiarity of Keyboard was significant ($p = 0.046$). The marginal mean for the group Familiar with the Keyboard was 0.04 (0.034) compared to 0.13 (0.05) for those who were not. The counts for the actual observed values of the times an expression was repeated for a trial was constant (5.6%) across methods for those familiar with a keyboard, but differed across methods for those not familiar with a keyboard (27.8% for the text method and 0% for the block method).

Table 2. Marginal means for models of metrics with standard error

Groups	Build Time, s (SE)	Correctness, % (SE)	Expression Repeated for Trial (SE)
Blocks method	69 (10)	98 (2.0)	0.04 (0.04)
fwKeyboard	55 (12)		
Not fwKeyboard	86 (18)		
Text method	22 (4.1)	95 (3.0)	0.14 (0.03)
fwKeyboard	10 (2.1)		
Not fwKeyboard	44 (14)		

* Detect Time = error detect time, ED Correctness = correctness of error detection, fwKeyboard = familiar with keyboard
s=seconds, SE = Standard Error

We considered the most important metric to be correctness as this most directly relates to whether any code created could be run without any corrections. We expected that the easier and quicker a user would be able to get to the “run” stage, even if the code was logically incorrect, would maximize enjoyment and minimize frustration for the user (i.e., the more time spent fixing syntax, the less enjoyable and more frustrating the experience). For this study which had the participants constructing simple expressions, the difference between the two methods was not significant. We also expected that the metric of build time contributed to this goal. Although differences in build time as a function of method and keyboard familiarity were significant, the actual time taken in all conditions was very short (less than a minute and a half). Neither of these metrics suggested a preference of one method over the other.

The metric of the times the expression was repeated gives an objective measure related to cognitive load during the build task, as it indicates the degree to which participants were distracted by the mechanics of the build process (whether it was typing keys or building block assemblies). The results suggest that the only advantage of one method over the other was for users who were not familiar with a keyboard, who would benefit, to some degree, by using the blocks method.

Participants also commented that the block coding method was intuitive and easy to use. However, they had one concern about the current design. Although they liked the idea that the magnetic attraction, or lack thereof, could help identify whether blocks can be validly connected, they felt that that the magnetic attraction did not present a strong enough haptic cue to be used. They did think, though, that the magnetic attraction was strong enough to hold the blocks together.

3.5.2 *Debugging Simple Expressions*. The means of the two different methods were compared for the time it took participants to detect if there was an error and if they were correct in that assessment, and the time it took to fix an error and the correctness of the fixed error (Table 3). The latter two metrics (time to fix an error and the correctness of the resulting expression) only included cases in which an error was determined (independent of whether this was correct for the given expression). For the metric of detection time, no effects were significant. The initial model for assessment of error only included the main effects and interaction between Method and Familiarity with Keyboard as otherwise the model fit did not converge: only the main effect of Method ($p = 0.041$) was significant. For the metric of time to fix an error, no effects were significant. The initial model for correctness of a fixed error only included the main effects as otherwise the model fit did not converge: no effects were significant. The counts for the actual observed values of the correctness of the fixed error indicated that Familiarity with Keyboard only had a small decrease in performance when the Blocks method was used (94% compared to 100% for those unfamiliar with the keyboard). Table 3 shows the results for all metrics with models that include Method and Familiarity with Keyboard as part of the model. This allowed for the breakdown of the data along the same lines as the other studies. For the correctness of the fixed error, the interaction term was not included as the model failed to converge.

Table 3. Marginal means for models of metrics with standard error

Groups	Detect Time , s (SE)	ED Correctness, % (SE)	Fix Time, s (SE)	Correctness, % (SE)
Blocks method	17 (3.3)	96 (3.0)	28 (3.9)	96 (3.3)
fwKeyboard	15 (4.1)	97 (2.3)	29 (5.4)	
Not Keyboard	19 (3.1)	93 (3.0)	26 (5.7)	
Text method	17 (2.6)	88 (4.0)	27 (5.2)	83 (4.4)
fwKeyboard	16 (4.1)	92(3.9)	41 (10)	
Not fwKeyboard	18 (2.9)	82(7.7)	18 (5.2)	

* Detect Time = error detect time, ED Correctness = correctness of error detection, fwKeyboard = familiar with keyboard
s=seconds, SE = Standard Error

In addition, the group with more severe blindness showed a tendency to take longer to detect an error (mean = 21 (8.0)) but quicker to fix it (mean = 25 (7.8)) than the group with low vision (means=15 (3.0), 29 (5.7)). They also showed a tendency to be more correct when fixing errors (98% compared to 88%).

Again, we considered the most important metric to be correctness (for identifying that there was an error and to produce a correct expression) as this most directly relates to whether any code created could be run without further debugging of syntax. Participants were significantly better at determining whether there was an error when using the block coding method compared to the text coding method. Although there was no statistically significant difference between the two for correctly fixing an expression, the trend was for participants to perform better with the block coding method. There were no significant differences in method for the detection time or the time to fix the error. The results of the data analysis for this study suggest that use of the block coding method over the text coding method would be a benefit for users independent of their familiarity with a keyboard.

The experimenter also made some notable observations about the two methods. For the block coding method, it was noted that participants really did not use the haptic feedback about the magnetic pull between correct blocks to determine if their code was correct. This was a detriment to performance as surprisingly the local shape cues were not as obvious haptically as was expected when they design was created; this was because any gaps from incorrect syntax were hard to detect on the front surface (Figure. 8). Participants also had some difficulty with the text method due to not knowing exactly where the cursor was located within an expression; this resulted in participants missing the point of deletion and insertion by one character. Often, participants did not check if their edit was done as they intended.

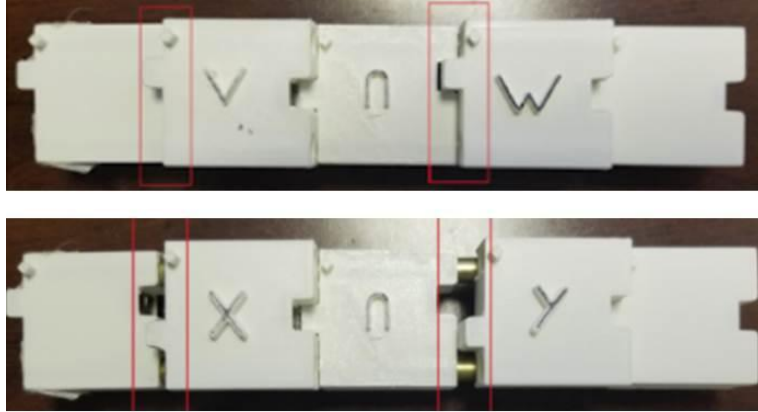


Figure. 8. Top: correct expression, bottom: incorrect expression.

3.5.3 User Survey After Tasks Involving Simple Expressions. The means of the different methods were compared for the fraction of time spent on programming, and each of the paired questions (Table 4). The positive statement within each pair was: S2, S3, S6, S8 and S10. For the metric of the fraction of time spent programming, only the main effects of Method ($p < 0.001$) and Familiarity with Keyboard ($p < 0.001$) were significant. For the S2-S7 pair, the main effects of Method ($p < 0.001$), Keyboard ($p < 0.001$) and Degree of Vision Loss ($p = 0.048$), as well as the interaction between Method and Keyboard ($p < 0.001$) were significant. For the S3-S9 pair, the main effects of Method ($p = 0.023$) and Keyboard ($p = 0.010$) were significant. For the S4-S8 pair, only the main effect of Keyboard ($p = 0.014$) was significant. For the S5-S10 pair, only the main effect of Method ($p = 0.47$) was significant. For the S6-S11 pair, there were no significant effects, although trends in the data showed that, for those not familiar with using a keyboard, the block method was considered more fun compared to the screen reader method. The models from which Table 4 is derived contain the main effects of Method and Familiarity with Keyboard, and their interaction, as well as any other significant terms: this allows us to provide a consistent breakdown in terms of Method and Familiarity with Keyboard across statement pairs.

Table 4. Marginal means for models of metrics with standard error

Groups	S2-7 (SE)	S3-9 (SE)	S4-8 (SE)	S5-10 (SE)	S6-11 (SE)	Fraction of Time Programming (percent)
Blocks method	4.8 (0.11)	4.5 (0.20)	4.0 (0.21)	4.5 (0.26)	4.8 (0.08)	90 (2.0)
fwKeybrd	4.8 (0.10)	4.8 (0.16)	4.3 (0.33)	4.5 (0.31)	4.7 (0.15)	93 (2.8)
Not fwKeybrd	4.7 (0.14)	4.3 (0.36)	3.8 (0.27)	4.5 (0.41)	5.0 (0.0)	87 (2.7)
Text method	3.9 (0.27)	3.7 (0.36)	3.5 (0.34)	3.4 (0.52)	4.0 (0.43)	65 (5.4)
fwKeybrd	4.6 (0.21)	4.5 (0.23)	4.0 (0.27)	4.2 (0.37)	4.6 (0.14)	84 (4.6)
Not fwKeybrd	3.2 (0.43)	2.8 (0.68)	3.0 (0.62)	2.7 (0.98)	3.5 (0.85)	46 (9.8)

* fwKeyboard = familiar with keyboard, SE = Standard Error

In addition, the group with more severe blindness tended to give higher ratings for the S2-7 question pair (mean = 4.6 (0.10)) compared to the low vision group (mean = 4.0 (0.30)).

The metric giving the fraction of the time spent programming provides a subjective measure related to cognitive load during both the build and the debug tasks, as it indicates the amount of time participants were focussed on programming rather than the mechanics of the tasks (i.e., typing keys or building block assemblies). The block coding method provided significantly more time to focus on programming than the text coding method.

Examination of the marginal means for the different statement pairs found that the block coding method was significantly better than the text coding method (of approximately 1 point on the Likert scale) for ease of use, helpfulness in learning how to program and confidence in doing the tasks. Enthusiasm for learning how to program was also higher with this method, although the results were not statistically significant. For all these statement pairs, this was primarily due to perceived differences between the coding methods by participants not familiar with a keyboard. This is of concern as students less likely to know how to use a keyboard (computer) are probably also least likely to have an existing interest in programming: these are the students that Scratch is trying to attract. Although alternate Braille entry devices for computers exist, only

10% of the BVI population know Braille. Proficiency of using a screen reader (with a keyboard) is also not an easy feat as it takes several years of practice to obtain.

3.5.4 Building Complex Expressions The means of the different methods were compared for the time it took to build a complex expression, the correctness of the built expression and the number of extra times the experimenter was asked to repeat the word description of the expression (Table 5). For the metric of build time, only the main effects of Method ($p < 0.001$) and Familiarity with Keyboard ($p < 0.001$), and their interaction ($p < 0.001$), were significant. The initial model for correctness only included the main terms and the interaction between Method and Familiarity with Keyboard as otherwise the model fit did not converge: however, only Method was significant ($p = 0.007$). The variation due to familiarity with the keyboard was very small, as well as non-significant. For the metric of time expression repeated, only the main effect of the keyboard was significant ($p < 0.001$). Table 5 includes the results for this metric that includes Method as part of the model, although there was no real difference between methods.

Table 5. Marginal means for models of metrics with standard error

Groups	Build Time, s (SE)	Correctness, % (SE)	Times Expression Repeated for Trial (SE)
Blocks method	184 (23)	89 (5.9)	1.1 (0.42)
fwKeyboard	139 (24)		0.89 (0.57)
Not fwKeyboard	244 (46)		1.4 (0.43)
Text method	67 (10)	69 (7.6)	1.1 (0.23)
fwKeyboard	31 (7.4)		0.81(0.28)
Not fwKeyboard	147 (26)		1.5 (0.36)

* fwKeyboard = familiar with keyboard *s=seconds *SE = Standard Error

In addition, the group with more severe blindness tended to take longer to build an expression (mean = 154s (36s)) but were more correct (mean = 86 (12)) than the low vision

group (means = 92s (15s), 77 (6.0)). They also tended to ask the experimenter to repeat the expression more (2.25 times compared to 0.70 times).

As for the simple expressions, the most important metric was again considered to be correctness. Unlike for the simple expressions, correctness did significantly vary with Method, with the block coding method performing 20% better than the text coding method on average. However, the difference in the block coding method from 100% suggests that potentially more effective haptic feedback would be useful (as suggested previously by participants in Section 3.1). It should be noted that although using the text coding method was faster in building the expressions than the blocks method, if one would factor in the subsequent response to error messages to fix the code (making the correctness more comparable between the two methods), it is unlikely that the difference is meaningful. The results of the data analysis for this study suggest that use of the block coding method over the text coding method would be a benefit for users independent of their familiarity with a keyboard.

The experimenter observed similar patterns of participant behavior as for the simple expressions, although, not surprisingly, participants struggled more with programming the more complex expressions.

3.5.5 Debugging Complex Expressions. The means of the two different methods were compared for the time it took participants to detect if there was an error and if they were correct in that assessment, and the time it took to fix an error and the correctness of the fixed error (Table 3). The latter two metrics (time to fix an error and the correctness of the resulting expression) only included cases in which an error was determined (independent of whether this was correct for the given expression). For the metric of detection time, only the main effect of Familiarity with Keyboard was significant ($p = 0.017$). The mean for the group familiar with

using the keyboard was 34 sec (5.1 sec) and was 71 sec (12 sec) for those who were not. For the metric of time to fix an error, only the Method – Familiarity with Keyboard was statistically significant ($p < 0.001$). However, this is not meaningful if the main effects are not significant. For the metric of correctly determining that an error existed, only the main effect of Method was significant ($p = 0.012$). For the metric of correctly fixing an error, if the participant thought one existed, again, only the main effect of Method was significant ($p < 0.001$).

Table 6 shows the results for all metrics with models that include Method and Familiarity with Keyboard as part of the model. This allowed for the breakdown of the data along the same lines as the other studies.

Table 6. Marginal means for models of metrics with standard error

Groups	Detect Time, s (SE)	ED Correctness, % (SE)	Fix Time, s (SE)	Correctness of Fix, % (SE)
Blocks method	51 (7.8)	94 (2.1)	91 (14)	85 (4.0)
fwKeyboard	38 (5.1)	97 (1.6)	72 (16)	92 (3.8)
Not fwKeyboard	68 (19)	86 (4.5)	123 (12)	75 (7.5)
Text method	48 (4.8)	82 (5.4)	56 (10)	61 (1.1)
fwKeyboard	30 (5.8)	86 (8.2)	34 (10)	62 (1.2)
Not fwKeyboard	75 (4.4)	78(5.1)	100 (20)	59 (1.8)

* Detect Time = error detect time, ED Correctness = correctness of error detection, SE = standard error, s = second

In addition, the group with more severe blindness tended to fix an error more quickly (mean = 48 sec (12 sec)) compared to those with low vision (mean = 88 sec (15 sec)). As for the simple expressions, the most important metric was again considered to be correctness (for identifying that there was an error and to produce a correct expression). Participants were significantly better at determining whether there was an error and fixing an error if they thought one existed when using the block coding method compared to the text coding method. Again, however, the difference in the block coding method from 100% suggests that potentially more effective haptic feedback would be useful (as suggested previously by participants in Section

3.1), particularly for users not familiar with a keyboard. Pairwise comparison for correctness of detecting an error was found the difference in performance for those familiar with a keyboard and not familiar with a keyboard was significantly different for both the block coding method ($p = 0.012$) but not the text coding method. Pairwise comparisons for correctness of a fixed expression found that the difference in performance between those familiar and not familiar with a keyboard when using the block coding method was significant ($p = 0.045$). Again these results suggest that the block coding method is more effective than the text coding method, although improvements should be made to more easily detect errors (as per Section 3.1).

3.6 User Survey after tasks involving Complex Expressions

The means of the different methods were compared for the fraction of time spent on programming, and each of the paired questions (Table 7). The positive statement within each pair was: S2, S3, S6, S8 and S10. For the metric of the fraction of time spent programming, the main effects of Method ($p < 0.001$) and Familiarity with Keyboard ($p = 0.007$), as well as their interaction ($p < 0.001$) were significant. This was also true for the S2-S7 pair, for which $p < 0.001$ for all terms, and for the S3-S9 pair, for which $p = 0.007$, $p < 0.001$ and $p < 0.001$, respectively. For the S4-S8 pair, the main effect of Method was close to significant ($p = 0.054$) and the main effect of Familiarity with Keyboard was significant ($p < 0.001$). Although the interaction between Method and Degree of Vision Loss was significant ($p = 0.022$), as the main effect of Degree of Vision Loss was not significant ($p = 0.089$), the interaction term was not included in the final model. For the S5-S10 pair, the main effects of Method ($p = 0.033$) and Familiarity with Keyboard ($p < 0.001$) were significant, as well as their interaction term ($p = 0.002$) and the interaction between Method and Degree of Vision Loss ($p < 0.001$). As the main effect of Degree of Vision Loss was not significant ($p = 0.33$), the latter interaction term was not

included in the final model. For the S6-S11 pair, only the interaction effect of Method and Familiarity with Keyboard ($p = 0.038$) was significant. However, this is not meaningful as the main effect of Method ($p = 0.098$) and Familiarity with Keyboard ($p = 0.504$) were not significant. This statement pair does seem to show the same trend as the other statement pairs. The models from which Table 7 is derived contain the main effects of Method and Familiarity with Keyboard, and their interaction, as well as any other significant terms: this allows us to provide a consistent breakdown in terms of Method and Familiarity with Keyboard across statement pairs.

Table 7. Marginal means for models of metrics with standard error

Groups	S2-7 (SE)	S3-9 (SE)	S4-8 (SE)	S5-10 (SE)	S6-11 (SE)	Fraction of Time Programming (percent)
Blocks method	4.5 (0.17)	4.0 (0.16)	3.4 (0.29)	3.6 (0.30)	4.1 (0.32)	75 (4.8)
ffwKeybrd	4.5 (0.24)	4.1 (0.22)	3.7 (0.45)	3.8 (0.35)	4.1 (0.42)	81 (5.1)
Not ffwKeybrd	4.5 (0.24)	4.0 (0.24)	3.2 (0.36)	3.3 (0.49)	4.2 (0.49)	70 (8.2)
Text method	4.0(0.17)	3.7 (0.14)	2.9 (0.27)	3.1 (0.23)	4.1 (0.19)	64 (2.1)
fwKeybrd	4.6 (0.25)	4.5 (0.24)	3.5 (0.47)	3.9 (0.38)	4.4 (0.30)	85 (3.1)
Not ffwKeybrd	3.5 (0.24)	2.8 (0.14)	2.3 (0.27)	2.3 (0.27)	3.2 (0.36)	43 (2.7)

* ffwKeyboard = familiar with keyboard * Detect Time = error detect time, SE = Standard Error

As mentioned previously, the metric giving the fraction of the time spent programming provides a subjective measure related to cognitive load during both the build and the debug tasks. Using the block coding method provides participants who were not familiar with using a keyboard with significantly more time to focus on programming than the text coding method. In addition, for the block coding method, the overall fraction of time spent on programming was lower for these complex expressions than for the simple expressions, whether or not participants were or were not familiar with using a keyboard. In contrast, the values were essentially constant

for the text coding methods. This is presumably because the complex expressions are a bit more involved to physically assemble. If possible, the design should be simplified or the training improved for nesting expressions with the block coding method.

Examination of the marginal means for the different statement pairs found that the block coding method was significantly better than the text coding method (of approximately a half a point on the Likert scale) for ease of use, quick to learn, helpfulness in learning how to program and confidence in doing the tasks. Unfortunately the overall ratings for the block coding methods were approximately half a point lower than for the tasks with simple expressions (although there was no drop for the text coding method). Again, this suggests that the design and/or training should be improved for working with more complex expressions. This may also help improve enthusiasm for this approach, which was also lower than for simple expressions. For these tasks with complex expressions, the difference between the block coding method and the text coding method was primarily due to differences found for participants not familiar with a keyboard. As mentioned earlier (Section 3.3), these are probably the type of participants we are most interested in targeting by teaching with a block coding approach.

3.6 Limitations

There were two main limitations to the study. First, adults were used as participants despite the target audience being children in middle school (grades 6-9). However, the logistics of recruiting BVI children for a study requiring a long time commitment during the school year is a daunting task for an already small target group (i.e., BVIs in general). In addition, we do not expect a difference in performance between children and adults unfamiliar with programming on most measures. This is because the tasks and training were ones that could be handled by middle schoolers. We do expect some differences on the statement pair asking about enthusiasm about

programming with the different methods with children, although we expect similar trends would still hold. The second was that the study focused on the use of syntax rather than programming as a whole. The reasoning for this was twofold: 1) alleviating uses of struggling with syntax was the target of the block coding design, and 2) the duration of the experiment to train and test participants' ability to code programs would need to be much longer. Both of these limitations will be addressed in future studies where middle school students will be involved in a coding camp, and assessed on their performance using the tangible block coding method and the text coding method.

Finally, it should be acknowledged that although many of the survey statements were similar to the System Usability Scale survey, the measures have not been vetted for reliability and validity. Although there was some correlation between the subjective statement of cognitive load and the objective measure used in the build expression task for simple expressions, there was very little correlation for complex expressions. This may have been due to positive bias, despite the use of positive-negative pairs of statements. It may also have been due to participants recalibrating the scale they used between the first day of testing and second day of testing despite the days being close together.

3.7 Conclusions

Overall, the block method took longer to build the code than the screen reader method. This shows that the time participants took to go through the bins, take the code block out, identify the block, and complete the code generally took longer than for them to navigate the keyboard, type the code, and listen to the written code. However, the code that was built using the block method showed higher accuracy than the screen reader method. The reason that

possibly led to this difference is that in the screen reader method, the participants often felt confident about their code even when the written code was inaccurate. This is partially due to the screen reader lacking the immediate feedback when the code is incorrectly written. In contrast, the block method was designed to have immediate and obvious feedback when the code had an error. It is likely that due to the haptic feedback presented by the design of the block, the participants acknowledged that there was an error and therefore took time to fix the code until they felt the code was correct. This is another reason for having longer build time but with higher accuracy in the block method. Although the time to construct the code was longer in the block method, the less than two minute difference was not significant in the scheme of code writing. The most important factor in code writing is accuracy of the written code. If there is an error within the code, the programmer needs to go back, analyze the code, and fix the code. Also, fixed code is also prone to error, and if the coders cannot easily detect where the error occurred, they may have to debug the code line by line until they find the errors. Although the block method takes longer time to build the code, it may take the same or even less time than the screen reader method considering the significantly higher correctness of built code, error detection correctness, and correctness of the fixed code.

To improve the performance of the tangible blocks, the block design needs to be refined and use the stronger magnets. The magnetic attraction in the current design is strong enough to hold the blocks together and be perceived by the participants, but it was not strong enough to instantly be felt and make a decision off of. Different levels of magnetic attraction should be tested on their effectiveness on giving cues about the correctness of the fit. Finally, with additional improvements to the magnetic cue, it is expected that the code block design used in this study will be effective for many of the other Scratch block commands. Development of a

more complete set of commands and assessment through a simulated classroom experience is needed.

Chapter 4. Code Block Design - Further Considerations for Single Line Code Commands

4.1 Introduction

The Scratch programming language was designed to increase engagement and lower hurdles to programming for sighted K-12 students. Part of the design addressed the inability of many students to master the syntax of programming (Resnick et al., 2009) due to the need to remember many rules and pay attention to details when just beginning to learn how to program. Scratch makes the syntax explicit in the visual form of the code blocks themselves: different Scratch code block types are shaped to fit together only in ways that make syntactic sense (Figure 9). In addition, blocks that can fit together snap to connect when close to each other. Blocks that should not be joined syntactically cannot physically connect. If a user attempts to make a connection that is syntactically incorrect (i.e., the blocks do not fit), one of the blocks is virtually “ejected” to another part of the workspace. This makes the syntax intuitive and creates a low floor for getting started with programming, allowing students to immediately start tinkering to create programs.



Figure 9. Scratch Code. Scratch creates not only expandable slots for code blocks, but defines connections with shape. Vertically, the code is attached in sequence through notch-tab pairs that exist only for code blocks that can join in this way. Horizontally, hexagonal and oval shapes define expressions that produce Boolean or numeric output, respectively. The hexagonal “slot” on the if statement indicates that a block that produces a Boolean output is required, such as the greater than operator expression.

Unfortunately, although very beneficial for sighted users, drag and drop virtual interfaces, such as the one used in Scratch, are inaccessible for individuals who are blind or visually impaired (BVI). For BVIs, a design that functions in an alternate modality to vision, such as either audition and/or haptics/touch, is needed. However, it is difficult to conceive of a solely auditory method, which primarily uses a 1-D temporal stream to convey information, to represent 2-D code structures. It is also difficult to imagine an appropriate auditory analogy to shape that could be used to make syntax explicit and intuitive. The construction of 2-D structures and shape does, though, naturally map onto haptics and the physical manipulation of real (tangible) objects. In fact, many researchers have developed coding and other educational tools using tangibles for BVI students (as reviewed in Morrison et al., 2020).



Figure 10. Overview Diagram. The code blocks are organized into bins and brought by the user to the code editor workspace where they assemble them into syntactically correct code. The blocks are tagged on their underside with ArUCo markers that are tracked by a camera through the clear workspace surface. The tracking data will be used to interpret the code and translate it to both online “visual” Scratch code and a separate representation on a nonvisual “stage” (where the stage in Scratch shows the execution of program code through animation of sprites).

Our laboratory is currently developing a tangible interface for the Scratch programming environment to provide accessibility to BVI students (Goolsby et al., 2021). At this point it primarily consists of a tangible code editor (Figure 10), where users can physically manipulate the code blocks to assemble them into programs. In order to be able to create moderately sized

programs, the general shape of a basic block is a 1” by 1” square, which is easily held in the hand and allows over 450 blocks to fit into a rectangular area. This is within the average 11-year old’s arm span, the low end of our target demographic.

The focus of this current work is in support of determining an effective method to provide explicit and intuitive syntax to BVIs through the tangible code blocks themselves. The motivation is to maintain the low floor for getting started with programming that Scratch created for sighted users. As the strengths and weaknesses of haptics/touch are different from vision, it is not clear that a direct replication of the methods used in “visual” Scratch will work for a tangible interface. In addition, physical blocks cannot be easily animated.

4.1.1 Previous Work

In considering the construction of tangible code blocks into syntactically valid expressions for BVIs, several different methods could be used to facilitate the process. Koushik and colleagues (2019) used the overall shape of a block (e.g., square or oval) to describe the block type. This provided a tactile cue for correct program construction but did not mechanically constrain connections to that between blocks with valid syntax only. In the Scratch virtual code editor, Maloney and his colleagues (2010) ensured that only connections with valid syntax could be made by making the edges of the correctly fitting blocks have reciprocal geometry. Both overall edge shape (e.g., for operators) and local edge shape (e.g., for sequencing) were used to define the reciprocal geometry (Figure 9). However, previous haptics research suggests that the determination of overall shape is slow and cognitively demanding, involving the serial integration of information over multiple contacts. In contrast, abrupt discontinuities are available much quicker and easily observed (Lederman and Klatzky, 1993; Lakatos and Marks, 1999).

Similar to “visual” Scratch, our design proposed the use of notches and juts to connect syntactically valid tangible blocks vertically in sequences (Figure 11a). However, instead of using reciprocal overall shapes for operators and parameters, a notch/jut system was used in combination with varied location to indicate syntax. A notch/jut in the upper part of a syntactical edge indicates a Boolean input/output (Figure 3b). A notch/jut in the lower part of a syntactical block interface indicates a numeric input/output. The jut of a numeric variable cannot connect to a notch of an operator requiring a Boolean input (Figure 11c). Surprisingly, although the approximately 5mm gap is easily detected visually, it is difficult for users to detect haptically. We determined through observation and feedback that this difficulty is what caused BVI users to create syntax errors that they could not detect. It also resulted in decreased user confidence in their programming. This suggests that more salient feedback is needed. One solution could be to increase the size of the notches and juts to improve haptic perception of the gap when blocks should not connect. However, this would require an increase in block size that would result in a decrease of the number of blocks and, therefore, the program size that could be constructed within arms-reach.

A second possibility to improve the saliency of a notch-jut connection is to use a plug and socket connection for which a successful connection can easily be gauged. Although this was effective for large blocks connected through cables (e.g., Morrison et al., 2020; 2021), effective plug and socket connections that provide good haptic feedback while being unaffected by wear are relatively costly. This becomes prohibitively expensive for our target design which involves hundreds of elements. In addition, blocks in the middle of a program of adjacently connected blocks are not easily replaced during debugging with this type of connection, as additional space between the blocks is needed to disconnect the plug and socket.

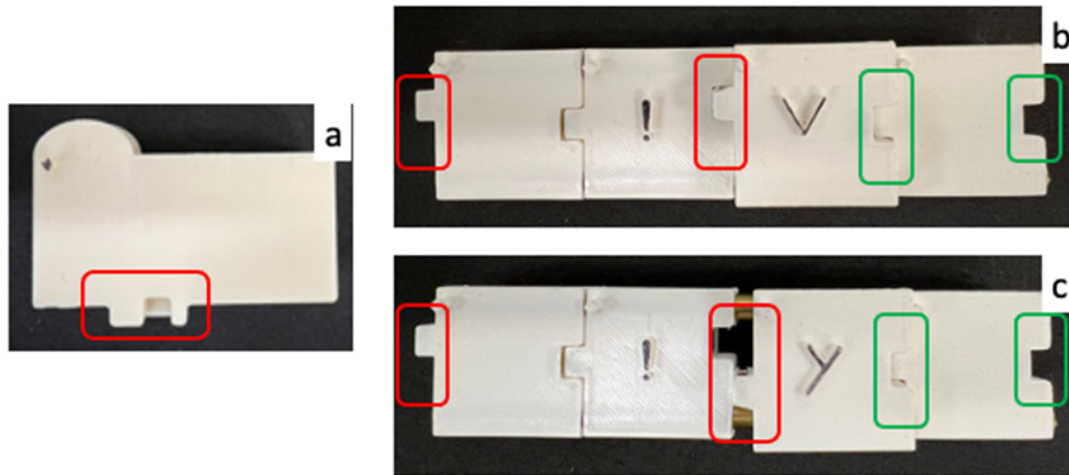


Figure 11. Code blocks showing syntax connections. The red boxes highlight the notches and juts used to define connection syntax. The green boxes indicate notches that are more generic connections. a) Prototype start event block. The curved top (in addition to no notch or jut) prevents a block from connecting above it. The jut on the lower edge allows for connection to a syntactically valid block below it. b) and c) Prototype logical NOT (i.e., negation) block assembly (shown without a parameter in Figure 5), with a Boolean variable (b) and a numeric variable (c) inserted into the parameter slot. An additional, lower notch will be present on other operators that accept numeric variables as parameters as well.

An alternative is to use magnets in the notch/jut connections (Goolsby et al., 2021).

Magnets which are sufficiently strong for syntactically correct blocks that are in close proximity to one another can cause them to move to connect together with an audible “snap”. This is a highly salient cue. Although “visual” Scratch has the ability to not only snap blocks together but also to “eject” blocks if they are syntactically incorrect, we chose not to replicate the latter ability. This was because informal feedback from BVI users suggested that the audible “snap” or lack thereof was sufficient to increase the detectability of valid and invalid syntax. The use of magnetic repulsion with magnetic attraction would also make the mounting the magnets on the code blocks more difficult due to different adjacent polarities.

In this paper we present three studies performed to help develop guidelines for the use of magnets with local shape-location constraints for facilitating syntactically correct code construction with tangible code blocks. The studies performed particularly examined the ability of magnets to provide effective “snap” connection feedback under conditions that may vary in

our code block design. The strength of the magnetics was kept to a level that still allowed users to separate the blocks if needed. Optimizing this process is expected to be important in providing reliable and consistent feedback about connectivity. This is expected to be critical in removing the creation of unintended syntax errors and increasing user confidence.

4.2 Objectives

The saliency of the “snap” cue is dependent on the motion of syntactically correct blocks coming together. This is dependent on both the magnetic attraction force and the mass/inertia of the block in motion. A weak magnetic attraction force on a large, heavy block may result in slower motion and less of a “snap” or, in fact, no “snap” at all. This could have an effect on the perception of connectivity between blocks and, hence, the understanding that the syntax is correct or incorrect. The main hypothesis of studies 1 and 2 is that a BVI user’s ability to easily, non-visually determine whether blocks are connected, thus indicating correct syntax, will improve with the addition of magnets to the design.



Figure 12. “Visual” Scratch set code block. There is a drop down menu for the parameter on the left, and the parameter on the right can be any numeric value or variable.

The second hypothesis is that, as we expect the key physical cue to be created through movement, the magnet strength, as well as the size and weight of the moving block, will affect a user’s ability to detect a correct connection and will also affect the ease of use.

Most programming code components treat parameters similarly, accepting both variables and literals, with the only requirement being differentiating when Boolean or numeric values can be used. However, there is one exception: the assignment operator. In Scratch this is the set code block (as well as a change code block). For this operator, the parameter on the left (Figure 12) can only be a variable. This is implemented in “visual” Scratch through a drop-down menu that is automatically populated when variables are created. This is not possible to do in a tangible interface. One possibility would be to modify the general connection interface for parameters; this would increase the complexity of the design for all blocks. A potentially more prudent approach would be to modify the set code block so that the left hand parameter can be treated as an exception. To do this, we propose using a stopper that will work on decreasing the magnetic strength and, hence, the perception of a connection for literals but not for variables. The main hypothesis of study 3 is that the use of a stopper, in conjunction with the magnet design from studies 1 and 2, will allow BVI users to correctly and easily determine non-visually that two code blocks are not connected. The second hypothesis is that we expect that this will be affected by the design and the length of the stopper. An arbitrarily long stopper cannot be used as this design suffers the same weakness as the plug and socket during debugging.

4.3. Study 1 and 2: Using Magnets for Connections

4.3.1 Code Blocks: General Design

The code blocks are rectangular in shape and small enough to fit in a hand (1.18 x 1.18-inch square), while enabling moderately sized programs to be built within arms-reach of a user. Code blocks were 3D printed using an Ultimaker 3 Extended with PLA filament. The blocks are to be oriented with the side with the tactile symbols facing up (Figures 13.1 and 13.2).

A raised dot is always placed in the top left corner of the top surface to help indicate the orientation of the block (label b in Figures 13.1 and 13.2). The tactile raised symbol indicating the code block command is placed in the middle (label a in Figures 13.1 and 13.2). The juts and notches are used to define the syntax with the code block (labels c and in Figures 5.1 and 13.2). Different versions of the block design allow for no magnets embedded in the notch, a 2 mm thick magnet and a 5 mm thick magnet (label f, Figures 13.4 and 13.5 only show the versions with the magnets embedded). The polarity of all magnets is oriented to be South on a jut of a block and North on a notch of a block.

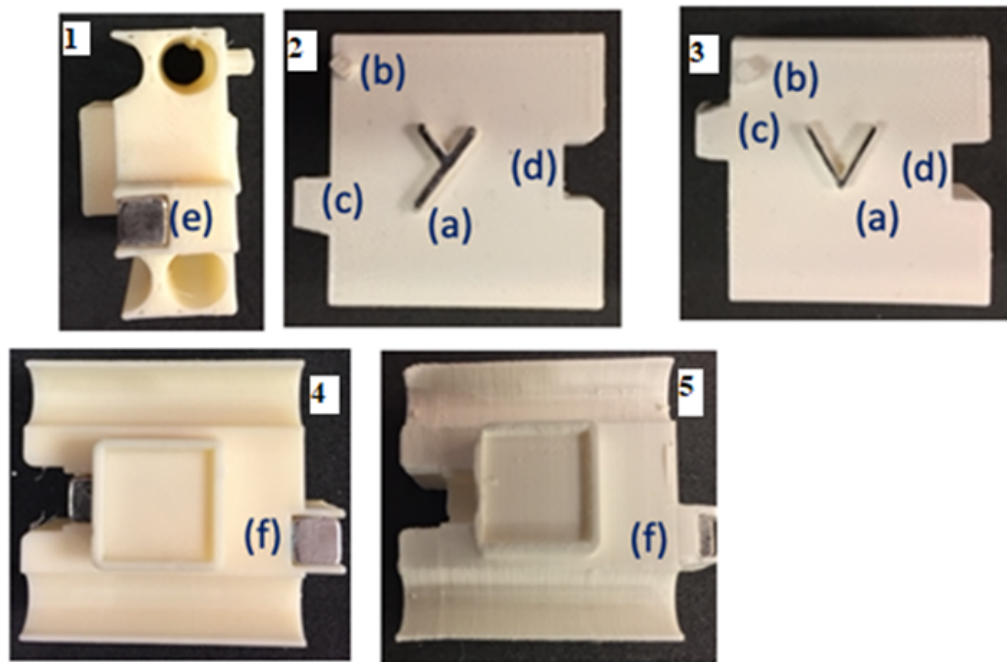


Figure 13. Standard sized variable blocks. 2: Numeric variable. 3: Boolean variable. Labels: (a) raised symbol indicating variable name, (b) raised dot for orientation, (c) jut, with the location along the edge used to distinguish a numeric variable from a Boolean variable, (d) notch for a non-syntactical connection. 5.1 (e) side profile showing the magnet inserted into the juts. 5.4 and 5.5: (f) back profile, showing the two different magnet sizes used in the jut.

4.3.2 Code Blocks: Studies 1 and 2 The test condition for the first two studies was to ask participants to try and assemble a line of code using either a Boolean variable block (Figure

13.3) or a numeric variable block (Figure 13.2) with a logical not operator (i.e., negation) block assembly (Figure 14). The logical not operator has one parameter, which can only be Boolean. This is defined by the notch labeled (a) (Figure 14) near the top edge. A notch lower down would accept a numeric variable. For the variable blocks, the jut on the left edge of each block indicates its variable type (labels c in Figures 13.2 and 13.3). Only the Boolean variable has a jut (label c in Figure 13.3) that fits the notch on the logical not operator. This defines the correct syntax for the logical not operator.

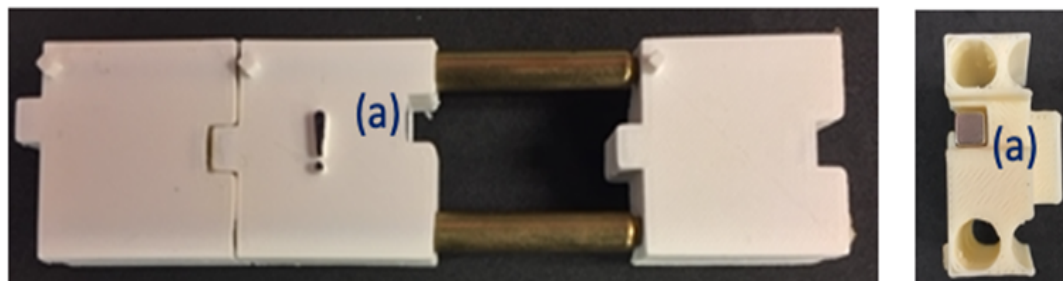


Figure 14. Logical not operator. The assembly is made of multiple blocks and copper telescoping tubing that can expand to accept blocks or block assemblies of different sizes. The notch on the right edge of the left-hand block (a) indicates this operator can only fit together with a Boolean variable or expression. The placement of the magnet in the notch (a) is shown more visibly in the diagram on the right.

One factor we manipulated in both studies was magnetic strength of the syntax connection. For this manipulation, we were able to vary the magnet placed within a jut or notch. Three different magnetic configurations for a jut/notch were used: no magnet, weak magnet or strong magnet. The magnets used were all neodymium rare earth magnets. The weaker magnet had the dimensions of 5x5x2 mm, a surface gauss of 3524 gauss and a strength of .46 kg. The stronger magnet had the dimensions of 5x5x5 mm, a surface gauss of 5666.8 gauss and strength of 1.7 kg. Both magnets were placed in such a way that one of the surfaces is on the outward facing surface of the jut or notch. Combinations of magnet pairs between the connecting notch and jut created four levels of magnetic strength: none, weak, medium and strong (Table 8).

Incorrect syntax only used the strongest magnets as there is no magnetic effect when the syntax is incorrect.

Table 8. Block configurations for conditions for Study 8.

Condition	Variable type	Variable size	Variable magnetic strength	Logical not magnetic strength
incorrect:small	numeric	small	strong	Strong
incorrect:large	numeric	large	strong	strong
none:small	boolean	small	none	none
none:large	boolean	large	none	none
weak:small	boolean	small	weak	weak
weak:large	boolean	large	weak	weak
medium:small	boolean	small	weak	strong
medium:large	boolean	large	weak	strong
strong:small	boolean	small	strong	strong
strong:large	boolean	large	strong	strong

In study 1, the second factor that was manipulated was block size (Table 8). Large size “variable blocks” had the exact same surface and edge features but were double in width (i.e., a 1.18 x 2.36 inch rectangle). In study 2, the second factor that was manipulated was block weight. The conditions were created very similar to that in Table 1, except weight (light and heavy) replaced size (small and large). Heavy blocks appeared the same as light blocks but were approximately 8 times heavier. The heavier blocks were created by hollowing a space in the middle of the top surface of standard blocks and placing dense metals inside. The logical not block assembly was not manipulated in size or weight.

4.3.3 Participants Five BVI individuals (2 female, 3 male) who were blind or visually impaired completed the experiment. The age of participants varied from approximately 24 to 64.

One participant was completely blind, one had light perception only, and the remaining three had low vision. All participants were blindfolded so that they would not be able to use any residual vision as the focus of the study was on the use of haptics alone.

4.3.4 Experimental Design Study 1 manipulated both the magnetic strength of a correct connection and the size of a Boolean variable block to be inserted into the logical not assembly (Table 1). Each block of trials consisted of these eight combinations as well as two catch trials, presented in random order. Catch trials consisted of using numeric variable blocks (with both standard and large sizes), which could not be successfully connected to the logical Not block assembly. These trials were used to reduce positive bias, as well as to subsequently assess any positive bias effect. Four blocks of trials were performed with each participant.

Study 2 manipulated both the magnetic strength of a correct connection and the weight of the variable block to be inserted into the assembly. The study design was the same as study 1 except two different block weights were used rather than two different sizes.

4.3.5 Experimental Procedure For each study, participants ($n = 5$) were blindfolded and sat at a table with the logical not block assembly component in front of them. The code block given to them to try to insert into the assembly was then placed on the table to their right. Participants were given as much time as they wanted to determine whether the code block fit in the code block assembly. They were required to answer yes or no. They were also required to rate the ease by which they determined the answer during the given trial on a scale of 1 to 5, where 1 indicated very easy and 5 indicated very hard.

4.3.6 Statistical Analysis Using G*Power 3.1 (Heine, University of Dusseldorf) to perform a power analysis for repeated measures within factors designed for ease of use that does not take into account the nested factors (which would give the study more power), it was

determined that medium to large effects could be detected. This is sufficient for our purposes as small effects are not likely to translate into functional differences.

For both the binary response variable of correctness and the numeric variable of ease of use, the mean and standard deviation for each subject as a function of condition, as well as the catch trials, were calculated. The catch trials were then separated from the other data. For correctness of the user response, Contingency tables were used to compare the correctness versus the different conditions (3 levels of magnetic attraction x 2 block sizes). Comparison between the use and non-use of magnets was made by making a dummy code for magnet/no magnet. Chi-square and Kendall's tau-b statistics were calculated in SPSS (IBM, Inc.). For the ease of use rating, general estimated equations were used using a normal distribution and identity link function. An exchangeable correlation matrix was selected as correlation between repeated measures is expected. First, models were constructed that only looked at differences due to condition. Then models were constructed that included main effects of magnet strength and block size, as well as their interaction. The effect size was derived from the η^2 value.

4.4 Results: Study 1 and 2

The main hypothesis being tested in studies 1 and 2 is that the addition of magnets to the block syntax connections will improve user's correctness in determining that the blocks are connected (i.e., syntactically correct) and ease of use. The second hypothesis is that the size and weight of the moving block will negatively affect this result.

4.4.1 Study 1 The mean and standard deviation for each condition as a function of subject are given in the Appendix. The catch trials were correct 95% of the time ($N = 40$), suggesting they performed their function in minimizing positive bias. For the remaining data, statistically,

there was a significant difference between the different block combinations (Table 9) for correctness 2 (7, N= 160) = 62, $p < .001$, Kendall's $-b = 0.40$) and ease of use, 2 (7, N = 160) = 20.000, $p < 0.001$, $= 0.35$). Correctness was poorer when no magnets were used 2 (2, N= 160) = 61, $p < .001$, Kendall's $-b = 0.62$). When magnets were used, correctness was 100% independent of block size and magnet strength.

Table 9. Study 1: Mean results of magnetic strength and block size (N = 160)

Combination (magnet strength – size)	Correctness, % (SE) *SE = Standard Error	Ease of Use (SE) 5 point scale *1 best
none-large	0.6 (0.10)	1.55 (0.35)
none-small	0.5 (0.10)	1.70 (0.36)
weak-large	1.0	1.45 (0.25)
weak-small	1.0	1.30 (0.27)
medium-large	1.0	1.25 (0.12)
medium-small	1.0	1.15 (0.13)
strong-large	1.0	1.05 (0.04)
strong-small	1.0	1.00 (0.0)

The model results for considering the effect of block size and magnet strength on ease of use are given in Table 10. The main effect of magnetic strength was not statistically significant, although this was possibly because the effect size was small. This contrasted with participant comments, where four out of five participants said they preferred the stronger magnetic attraction to the weaker ones. The main effect of block size was also not significant. However, the interaction effect was significant ($p = 0.019$) with a moderate effect size ($= 0.25$). This is somewhat consistent with participants' comments that stated that the strength of the magnet

mattered more for ease of use when the block was bigger. However, small blocks with no magnets did not follow this trend as it was the least easy to use.

Table 10. Study 1: Tests of Model Effects for Ease of Use (N = 160)

Source	Wald χ^2	df	Significance	Phi
(Intercept)	86.9	1	< 0.001	0.74
Block Size	0.682	1	0.409	0.065
Magnet Strength	5.532	3	0.137	0.19
Block Weight * Magnet Strength	10.0	3	0.019	0.25

4.4.2 *Study 2* The mean and standard deviation for each condition as a function of subject are given in the Appendix. The catch trials were correct 92.5% of the time (N = 40), suggesting they performed their function in minimizing positive bias. For the remaining data, statistically, there was a significant difference between the different block combinations (Table 11) for correctness ($2(7, N = 160) = 47, p < .001$, Kendall's $\tau_b = 0.36$) and ease of use, ($2(7, N = 160) = 20, p < 0.001, \tau_b = 0.35$). As in study 1, correctness was poorer when no magnets were used ($2(2, N = 160) = 46, p < .001$, Kendall's $\tau_b = 0.54$). When magnets were used, correctness was 100% independent of block weight and magnet strength.

Table 11. Study 2: Mean results of magnetic strength and block weight (N = 160)

Combination (magnet strength – weight)	Correctness, % (SE) <small>*SE = Standard Error</small>	Ease of Use (SE) 5 point scale *1 best
none-heavy	0.6 (0.10)	1.95 (0.52)
none-light	0.7 (0.10)	2.10 (0.67)
weak-heavy	1.0	1.54 (0.32)
weak-light	1.0	1.20 (0.18)
medium-heavy	1.0	1.25 (0.10)
medium-light	1.0	1.1 (0.089)
strong-heavy	1.0	1.10 (0.055)
strong-light	1.0	1.00 (0.0)

The model results for considering the effect of block weight and magnet strength on ease of use are given in Table 12. The main effect of magnetic strength was statistically significant ($p = 0.021$) and a medium effect ($f^2 = 0.25$) with ease of use increasing with increasing magnet strength. This was supported by participant comments, where four out of five participants said they preferred the stronger magnetic attraction to the weaker ones. The main effect of block weight and the interaction effect were not significant, but may be due to their small effect sizes. This contrasted with participants' comments that stated that the strength of the magnet mattered more for ease of use when the block weight was heavier.

Table 12. Study 2: Tests of Model Effects for Ease of Use (N = 160)

Source	Wald χ^2	df	Significance	f
(Intercept)	36	1	< 0.001	0.47
Block Weight	3.6	1	0.059	0.15
Magnet Strength	9.8	3	0.021	0.25
Block Weight * Magnet Strength	3.6	3	0.313	0.15

4.5 Discussion: Study 1 and 2

An important design criterion for our tangible code blocks is that they enable BVI users to easily and correctly determine when syntax connections are valid while building the code. This creates a lower floor for BVI students to get started with programming. It also helps shorten the transition between coding and executing a program, as syntax errors are addressed while constructing the code. This helps students stay engaged. The main hypothesis for studies 1 and 2 was that the addition of magnets to a design that uses local shape – location constraints will improve the correctness of identifying valid syntax connections and increase ease of use. Both studies showed that there was a statistically significant difference in correctness when magnets were used versus not used with a large effect size. In fact, participants in all conditions that used magnets were correct 100% of the time compared to only 60% (S.E. 10%) of the time without magnets. This suggests that without the use of magnets, another method would be needed to provide BVI users with feedback about syntax errors, as users would not be able to distinguish valid (i.e., successfully connected) syntax from invalid (i.e., non-successfully connected) syntax. Although only the logical not operator was used in testing, the syntax connection is of the type used for parameter slots that exist in other operators and most of the command code blocks. It is

also expected that the success of magnets with horizontally connecting blocks should also extend to vertically connecting blocks.

The second hypothesis was that the magnet strength, as well as the size and weight of the moving block, will affect a user's ability to detect a correct connection and will also affect the ease of use. In study 1, the size of the variable block was changed as we expected that some of the syntax connections (e.g., vertical sequencing) will involve block sizes bigger than a standard block. As we expect the key physical cue to be linear movement as the blocks "snap together", study 2 focused on varying weight (as linear movement relates to both force and mass). This allowed us to consider potential differences when using much larger blocks (e.g., if-else statements) within the framework of the logical not operator used in the studies. The magnetic strength of the connection was also varied as blocks with larger mass require larger forces to move them. When magnets were used, correctness stayed at 100% independent of the manipulation of magnet strength, block size and block weight. This suggests that although magnets are essential, their strength (within the range tested) is not critical for correctness. This means that as we expand our code block set, we do not have to be concerned with changing magnets for blocks of bigger sizes.

Ease of use is also important to consider as we do not want the haptic determination of connectivity between blocks to contribute to extraneous cognitive load while learning how to program. Change in magnetic strength did have a significant effect, with a medium effect size, on ease of use in study 2: ease of use increased with increasing magnet strength. However, the effect was not significant in study 1. This difference may be due to the small number of participants used in both studies. Another possibility is that the effect of magnet strength may be more visible with larger blocks (one of the conditions in study 2 used a much heavier block than

in study 1), although the interaction between magnet strength and weight was not significant in study 2. This is, though, consistent with study 1, where the interaction effect was statistically significant with a moderate effect size. As the strongest magnetic connection was still one for which our target population (ages 11 and older) are able to separate connected blocks, to ensure the best ease of use, the strongest magnetic connection should be used for all blocks. If the resulting design is considered for young children, a weaker magnetic connection would be required. This suggests that it will be important to hollow out larger blocks to keep a low weight for ease of use in making a connection.

4.6. Study 3: Using Stoppers with Magnets

The main hypothesis of study 3 is that the use of a stopper, in conjunction with the magnet design from studies 1 and 2, will allow BVI users to correctly and easily determine that two code blocks are not connected using non-visual means. The second hypothesis is that we expect that this will be affected by the design and length of the stopper.

4.6.1 Code Blocks: Study 3 Based on the results from studies 1 and 2, we were able to determine that use of the magnetic attraction increases the accuracy of the correct connectivity. However, the “set” and “change” code block commands have additional rules for their left hand parameter slot: it can only accept a variable (but not a numeric literal). For their design, we still want to incorporate magnets for connectivity as BVIs made it clear that they liked the audible “snap” significantly. What we then need is a method to prevent numeric literals from being used in the left hand parameter slot while still being able to be used in all other slots, including the right hand parameter slot of these commands.

Table 13 Block configurations for conditions for Study 3.

Condition	Parameter Type	Operator Stopper Type	Operator Stopper Length
design 1: fit	variable	Design 1	long
design 1: fit	variable	Design 1	short
design 2: fit	variable	Design 2	long
design 2: fit	variable	Design 2	short
design 1: long	literal	Design 1	long
design 1: short	literal	Design 1	short
design 2: long	literal	Design 2	long
design 2: short	literal	Design 2	short

Instead of an entire redesign of all parameter slots, the overall design can be kept more straightforward if we consider any necessary additions to the “set” and “change” code block assemblies that will avoid adding complexity anywhere else. The main idea was to add a stopper to prevent the “snap” connection for the numeric literals. However, an arbitrarily long stopper cannot be used as it will make connecting and disconnecting code blocks during debugging more difficult. Two different lengths of a stopper were implemented (3mm versus 6mm, as compared to the 4mm depth of the juts/notches). The strong magnetic configuration was used with all designs (as it is the one we will use going forward), which produce a weak magnetic attraction for the 3mm length stoppers but not for the 6mm length ones. In addition, two stopper types were also used: one aligns with the connection notch but has a larger vertical profile, while the other has a lower vertical profile but takes up real estate along the right edge of the “set” block (Figure 15; Table 13). To allow the correct syntax, a variable has a hole in its jut that allows it to connect with the “set” block assembly, while the numeric literal does not (Figure 16).

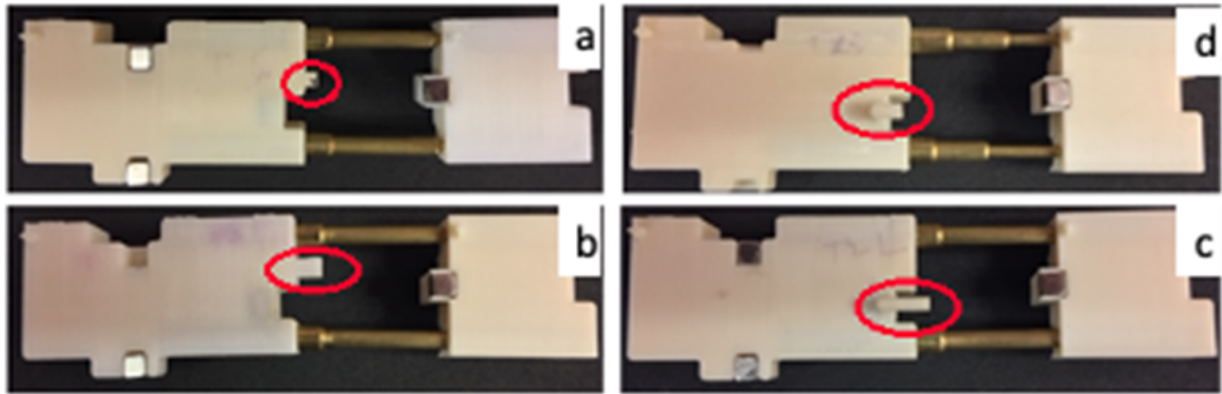


Figure 15. The left-hand side of the “set” blocks created for this experiment. From top left corner, rotating counter clockwise: a) type1-short, b) type1-long, c) type2-long, and d) type2 short. The circled areas are where the stoppers are located.

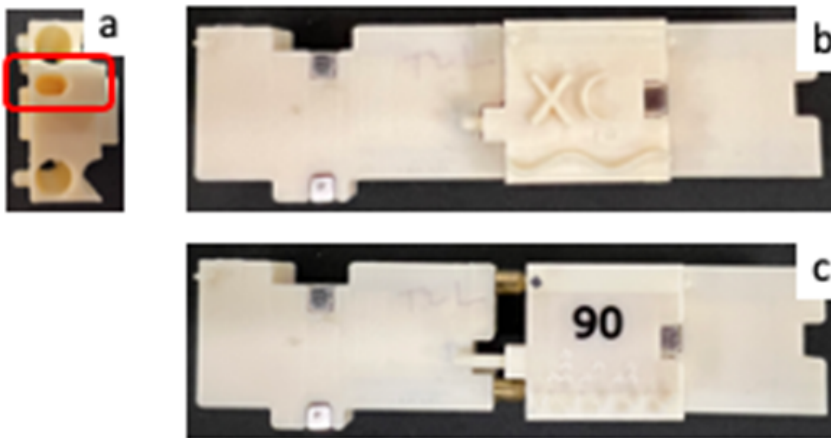


Figure 16. General view of the design and implementation of blocks with syntax exception a) Profile of a variable block that allows a connection with a “set” type2 block (magnets are embedded in the jut, in red, but not across the hole). Numeric literals do not have a hole. b) Variable block connected to a prototype left-hand side of the “set” block. c) Numeric literal not able to connect in the same parameter slot.

4.6.2 Participants. The participants were the same participants that participated in studies 1 and 2.

4.6.3 Experimental Design. Study 3 manipulated the stopper type and length for the “set” block assembly as well as whether a variable or a numeric literal was used. As it was very easy in pilot work for participants to determine when parameters fit (due to the “snap” connection), the number of trials for these conditions were reduced. Instead of treating the short and long lengths of stoppers as separate conditions, they were lumped into one condition. The length

selected for a given trial was randomly selected. We also considered the “fit” conditions as catch trials as we were most concerned about the design in the conditions when a connection should not be made. Each block of trials consisted of the six conditions presented in random order. Four blocks of trials were performed with each participant.

4.6.4 Experimental Procedure and Statistical Analysis. The experimental procedure and statistical analysis were similar to studies 1 and 2. The power analysis that was done was similar to that for the previous studies but taking into account the different number of conditions. It was determined that medium to large effects could be detected, which we deemed sufficient for our purposes as previously stated. The statistical analysis proceeded in a very similar way to studies 1 and 2. Differences were in the fact that the contingency tables were used to compare the correctness to the 2 stopper types and the 2 stopper lengths in addition to the different conditions in general. For ease of use, the model constructed included main effects of stopper type and stopper length, as well as their interaction.

4.7 Results: Study 3

The mean and standard deviation for each condition as a function of subject are given in the Appendix. The catch trials were correct 95% of the time ($N = 40$), suggesting they performed their function in minimizing positive bias. For the remaining data (Table 14), there was only a statistically significant difference between the different designs combined with stopper length for ease of use: $2(3, N = 80) = 20, p < 0.001, \eta^2 = 0.5$. The results for correctness were: $2(3, N = 80) = 7.4, p = 0.061$, Kendall's τ_b approximately 0. However, further analysis of the results for correctness found that the differences due to stopper length were significant: $2(1, N = 80) = 6.1,$

$p = 0.013$, Kendall's $-b = 0.277$. (Differences due to stopper type: 2 (1, $N = 80$) = 1.1, $p = 0.288$, Kendall's $-b = 0.119$).

Table 14. Study 3: Mean results of design type and stopper length ($N = 80$)

Combination	Correctness, % (SE) *SE = Standard Error	Ease of Use (SE)
design1-long	0.95 (0.025)	1.25 (0.17)
design1-short	0.75 (0.079)	1.75 (0.21)
design2-long	1.0 (0.0)	1.05 (0.04)
design2-short	0.85 (0.075)	1.95 (0.35)

The model results for considering the effect of stopper type and length on ease of use are given in Table 15. The main effect of stopper length was significant ($p < 0.001$) with a medium to large effect size = 0.42). Despite the lack of an effect of stopper type and stopper length-type interaction, four of the five participants preferred stopper type 2; however, two of those four participants said that their preference was in combination with a longer stopper length.

Table 15. Study 3: Tests of Model Effects for Ease of Use ($N = 80$)

Source	Wald χ^2	df	Significance	f
(Intercept)	77	1	< 0.001	0.98
Stopper Length	14	1	< 0.001	0.42
Stopper Type	0	1	1.0	0.0
Stopper Length * Stopper Type	1.633	1	0.20	0.14

4.8 Discussion: Study 3

A small number of code blocks, namely the set and change blocks, have different syntax rules for their left hand parameter slot that differs with all other parameter slots used by the operator and command code blocks. Study 3 investigated whether an added stopper to the set and change blocks could be used to prevent an incorrect connection between a numeric literal and the set command. This would allow the set and change code blocks to maintain the “snap” effect for valid syntax, while not impacting the design of all operator and command code blocks. The stopper length that was used did have a significant effect on both correctness of identifying that a numeric literal was not connected and ease of use, with a medium effect size. The high accuracy in identifying the lack of connectedness (95% with S.E. 2.5% for design 1, 100% with S.E. 0 for design 2) suggests that the use of a long stopper can reliably indicate that the numeric literal does not belong in that parameter slot. Although the shorter stopper would be easier for replacing the numeric literal during debugging (by requiring less additional space), it is not sufficient for conveying the needed information.

The stopper type did not have a statistically significant effect on correctness and ease of use. However, when asked, four out of the five participants preferred the second stopper design that was aligned but above the corresponding syntax notch of the set code block. The lack of statistical significance may be due to the small number of participants. This also may be due to the preference the second stopper design for two of the four participants was in combination with a longer stopper length. A third possibility is that other aspects of usability were important to the users.

4.9 General Discussion

The end goal of the design of the code blocks is to embed the syntax into the physical structure and connections of the code blocks to make syntax intuitive to beginning BVI (and sighted) students. This paper particularly concerns itself with providing sufficient feedback in the physical connections' structure to prevent students from misinterpreting when the syntax is correct or incorrect. The consequence of not meeting this criterion is that more feedback would need to be provided at the translation stage between writing the code and code execution. Beginning users tend to find feedback about syntax at the translation stage frustrating and the delay to program execution can greatly dampen engagement.

The Scratch programming language has created a successful solution for the visual, virtual domain. However, a tangible code editor interface for BVI students is neither visual nor virtual. Our original solution was to use reciprocal shape connections and location to define the syntax. Unfortunately, as the results of studies 1 and 2 show, performance in identifying whether a successful connection between blocks was made and, hence, the correctness of the syntax was poor. As is, the design would necessitate more feedback about syntax in the translation stage, which creates problems for engaging beginner learners. An alternative considered in studies 1 and 2 is to embed magnets in the local connection points. This resulted in significant improvement in identifying the correctness of the syntax, with participants performing the task at 100% for all magnet strengths. In contrast, ease of use improved with magnetic strength, suggesting that the strongest magnetic strength that the target user population can still separate apart should be used.

The main, general limitations to these conclusions are that the studies: 1) only considered one program concept, the logical NOT operator; and 2) did not use a more general programming

task. They were not considered at the time as both required further development of different code block types and also considerable instruction to beginner users. However, it was desirable to have feedback about the design at this stage of development. To address the first factor, the variable block was modified particularly over a large range for its weight, which we expected to influence the “snap” effect as it is due to motion. The results for ease of use as a function of magnetic strength and weight suggest that heavy blocks will require stronger magnets. An alternative to this, as it is really the size of block that will be required to increase for some code blocks (e.g., an if-else block), is that the inner portion of the block should be hollowed out to keep the weight of the block low.

Study 3 addressed the fact that there are a small number of Scratch code blocks types, namely “set” and “change” for which the behavior of the first parameter slot is different from all other parameter slots in the Scratch programming language. In “visual” Scratch, this was implemented by a drop down menu that has variable names automatically added to the interface when created: this is not something that can be implemented in a physical interface. Rather than increasing the complexity of the syntax connections for all code block types to take into account variable versus numeric literal, as well as Boolean versus numeric, study 3 considered the use of a stopper to implement the restriction of the first slot for “set” and “change” to variables only. The different stopper designs were able to convey the syntax condition to a reasonable degree for all stopper designs. However, the best design, which reached 100% correctness, was design 2 (with the stopper placed over the corresponding notch) with the longer length.

Unfortunately, the shorter length of stopper did not have as high of correctness as we would like (75% (0.025) for design 1 and 85% (0.075) for design 2). This would necessitate more feedback about syntax in the translation stage, which is less preferable than during

construction as mentioned earlier. The longer stoppers had better performance in terms of identifying the correctness of the syntax connection and ease of use; however, it also has some of the same concerns that were raised for plug and socket connectors in the introduction. In both cases, more space between code blocks is needed to connect and disconnect code blocks, which is potentially difficult when debugging in the middle of multi-line code that is connected together. As this issue will only exist for two types of code blocks, we have deemed this an acceptable trade-off for providing users with better clarity about the syntax. Further testing with multi-line programs is needed to assess this decision.

4.10 Extensions to Other Applications

Many areas in education require the use of some sort of syntax, whether it is computer programming, molecule construction, biological nomenclature, or more basic subjects such as mathematics, logic and languages. Research suggests that active, hands-on learning can be beneficial to the learning process (Yannier et al., 2021) through the construction of knowledge by transforming the world through action (Ackerman, 2004). However, when syntax is involved, beginning learners can feel overwhelmed. It is potentially desirable to lower the floor to get started in these areas as was done with the “visual” Scratch programming language. Many of the above mentioned applications are likely able to use block elements a similar size to that investigated in our three studies and could define how they connect using a similar local, reciprocal shape design to our work. For example, in molecule construction, local, reciprocal shape can be used to define valence electrons.

The results from our studies suggest that the use of local, reciprocal shape is not sufficient for defining syntax connections. It is strongly recommended that the addition of magnets embedded

in the local connections should be used to overcome these problems. The strongest magnets we used should be sufficient for many applications. The largest concern is for applications with younger children who may not be strong enough to pull connected blocks apart. Unfortunately, designs for younger children also use larger blocks that are easy to manipulate. From our results, this has potential cause for concern in terms of decreased ease of use in determining connections. Hollowing out the insides of these blocks may be one way to avoid the trade-off between better determining connections and ease of taking blocks apart. Often, the size of what is being built/assembled by younger children is much smaller than the code length we are considering, an alternative using plug and socket connectors (Morrison et al., 2020) may be an appropriate alternative to consider as, with smaller assemblies, access to individual elements is easier.

One weakness in generalizing the applicability of our studies to other applications is that they also focused on assemblies formed on the 2D plane of the table top; it is possible that different results may have occurred for more three-dimensional structures (e.g., Morrison et al., 2020). However, it is also possible that their negative experience may be overcome by stronger magnets than were used and the use of local juts and notches for firmer attachment.

4.11. Conclusions

The use of local reciprocal shapes and location to define syntax connectivity is not sufficient for BVI users to determine whether the syntax is correct in tangible block coding designs where syntax is meant to be built into the physical connectivity. This can be rectified by embedding magnets that attract in the locations of connectivity. Ease of use improves with magnetic strength of the magnets, although the target age group and their ability to disconnect connected code pieces also needs to be considered. The effect of magnet strength is particularly

pronounced for heavier code blocks. Our results suggest that this be rectified, if possible, by hollowing out larger code blocks. If a small number of code block types have additional specialized rules above the general syntax, it may be warranted to use a solution local to those code blocks than consider the issue in the overall design. The use of a stopper to prevent a connection between non-syntactically correct blocks worked well; however, it was clear that the length of the stopper was important. As a result, the weakness of this method is that it requires more extra space between blocks than desirable when removing a block during debugging. The design of syntax connectivity determined in these studies will be used, in future, to complete the translational of all Scratch commands to a tangible interface. All aspects of the language can be created through the use of slots for expected components within a command, with a block-to-block interface design following the one given here. Future work will examine the ability of BVI users to construct computer programs from the developed block set. This will particularly focus on the ability of BVI middle school students (the age at which Scratch is typically introduced in schools) to get started with learning how to program.

Chapter 5. Code Block Design: Non-Operator Blocks

The previously discussed work focused mainly on operator blocks: creating the ability for them to only accept valid block types as operands and allow nesting of expressions through horizontal expansion. These concepts were used to create movement command blocks (go to X,Y; glide S sec to X,Y; point in D direction; step forward X amount; rotate clockwise A amount and rotate counterclockwise A amount), as well command blocks for wait S sec and beep S sec. The developed technique for implementing exceptions to the basic syntax was applied to the commands for setting and changing of variables, where the element in the first slot can only be a variable.

Design of the symbols to describe the commands focused on selecting ones that were simple to perceive and intuitive to understand. Symbol selection was also based on feedback from stakeholders. One of the textures from the texture set previously determined through stakeholder involvement was selected to represent motion commands and one was selected to represent control flow commands including beep s seconds and wait s seconds (both of which are used to create a pause in a Sprite's action, one with audio feedback and one without). The looks and sounds commands that exist in the virtual Scratch editor will not be implemented in this prototype as they are not needed to fully test the tangible code editor concept.

The remaining step was to come up with the design for key control blocks so that programs of sufficient complexity can be created with the prototype system. This includes: repeat <expression> times statement, if <expression> then statement, if <expression> then statement – else statement, repeat until <expression>statement and wait until <expression> statement.

All of these commands are of a similar format:

If <expression> then

Statement

In this command, <expression> can be replaced by a nested combination of operands, operators and variables. The statement can be replaced by a sequence of statements of variable length.

All <expression>s use the same concept as for the movement command parameters and the operators: they can be constructed through nesting levels of operators on top of each other as designed previously. The expressions are positioned to be attached near the top of a control block and the location of the notch for the connection is dependent on the type of <expression> that is valid for that control function (Figure 13 for a preliminary example). For the repeat command, the <expression> must evaluate to a number. For all other commands, the <expression> must evaluate to a Boolean. Figure 17 shows an example for an if-then command, where the connection on the upper-right side is for a Boolean expression.

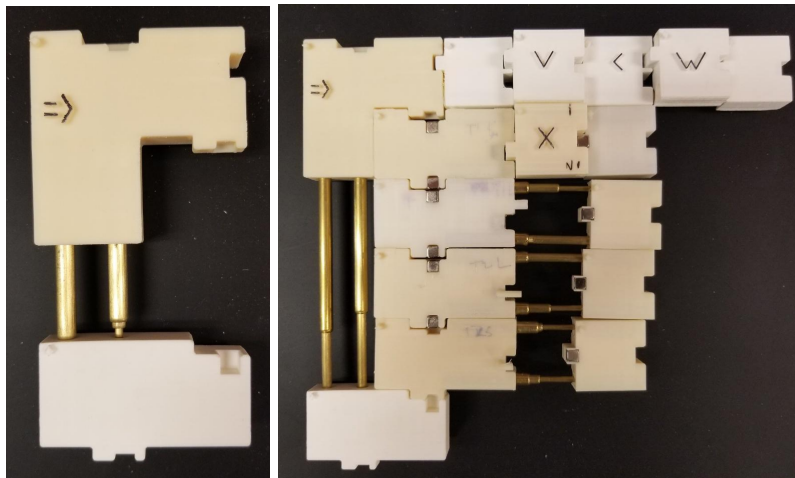


Figure 17. If-then block (left) and example code structure if (v<w) then...(right)

All of these commands need to be able to expand vertically in one or more places to allow for a varying number of “statement” commands to be executed based on the control

condition. Therefore, a method needs to be added to allow the block assembly to expand and contract vertically to allow a statement block to be only one command or many commands, depending on the programmer's intentions. As with visual code, the begin and end of such statements should be clearly defined and nesting should be allowed. This was done by applying the design principles of using telescoping, notches and juts, and magnets that have been effective for horizontal expansion (see Figure 13 and 14 for a preliminary example). Similar to blocks that expand horizontally, the vertically expanding blocks have a start and end block. For conditional blocks like if-then-else blocks that have two distinct statement blocks, it includes a middle block. The telescopes are attached to these types of blocks so that it expands vertically. The notches are the same vertical notches that are used to attach commands in sequence in the main "program", as the same commands can be used in either location (although the resulting behavior of the program will be different).

The overall design of the block assemblies continues to use raised dots to orient the assemblies, textures for block category and symbols for the command. Symbols were selected using a similar approach used for operators, basing them in mathematical, logic and universal symbols that can be easily perceived and intuitively interpreted through both touch and vision. Feedback on the ability to easily identify the haptic symbols was also sought from stakeholders during development.

All of these commands can also be found in the Appendix A.

As the design concepts are not significantly different from what was used for creating nested expressions horizontally, the resulting design was only informally tested and iterated through feedback from stakeholders. The only difference was that the blocks that are not nested were not put on top of the telescopes as horizontally expanding blocks.

Chapter 6. Assessment of Overall System

6.1 Introduction

For the final assessment of the tangible code blocks, we were interested in the use of the code blocks by the targeted population (BVI students in middle school) in a simulated classroom setting within the context of the entire nonvisual interface (code editor and program execution display, Figure 18). Due to time constraints and recruitment difficulties, a limited number of blind and visually impaired adults were recruited to be involved in the assessment one at a time. This chapter describes the data from each of two adult participants in a two day coding camp as case studies.

The main objective of the assessment was to determine if BVI participants used the system in the way we designed it to be used. We also noted any problems and improvements that should be addressed before scaling up the deployment of the system to classrooms. The second objective of this component was to determine if the participants appeared to be enjoying their experience and if it made them excited about programming.

This chapter describes the details of the simulated classroom, the data collection methods, the analysis and conclusions. Bryson Goolsby, Dr. Satinder Gill and Dr. Dianne Pawluk were involved in setting up the classroom and the data collection hardware. Dr. Pawluk and Bryson Goolsby acted as the instructional staff and collected the data.

6.2 Participants

Due to the limited time constraint, participants within the target age group (BVI middle school students) were not able to be recruited. A limited number of BVI adult participants (i.e., two) were recruited and the data analyzed as case studies. One participant was an 18 year old

totally blind female who recently graduated from high school. She was experienced with Braille and tactile diagrams. She had limited experience with coding, the extent of which was an attempt to take a Python coding camp which she quit early. The other participant was a 40 year old, relatively recently blind individual with light sensitivity only. She did not know Braille and did not have any experience with tactile diagrams or programming. Neither participant had any cognitive, motor, somatosensory or auditory impairments.

6.3. Classroom Set-Up

My contribution to the system (Figure 18) was leading the tangible code block design and manufacturing. The other components consisted of the workspace organization (both on the code editor work surface and the code block drawer storage), code block tag tracking and program execution display (i.e., a table top mobile robot acting as a Sprite on top of changeable tactile backgrounds).



Figure 18. Overview of the system. The program execution display (with both a tangible background and mobile robot Sprite visible) is on the left, the code editor work surface is in center, and the code block storage drawers are on the right.

Each student sat in a chair with the system components surrounding them in a U-shape. The instructor (Dr. Pawluk) sat in front of the student on the opposite side of the code editor work surface. The teaching assistant (Bryson Goolsby) who helped with the teaching and provided the Wizard of Oz components of the system sat beside the instructor at another computer terminal.

The code editor consisted of the work surface designed by Bryson Goolsby (Figure 19), the code block drawer organization and the code blocks/block assemblies designed by myself. Both the work surface design and the code block assemblies were designed through two iterations of design and assessment with blind and visually impaired adults. The code block drawer storage used stackable pull out drawers (Figure 20), with added tactile labels on the front that contained the orientation marker and haptic symbol of the code blocks inside. All containers contained more code block assemblies than were needed to complete the curriculum tasks. Drawers were used instead of bins as they were felt to provide easier access to the code assemblies.

One special issue raised in completing the system implementation was how to represent the different values of the number blocks. In “visual” Scratch, for each block, the user enters the value from the keyboard, which then becomes visible on the virtual block. This is not as straightforward for physical blocks and BVI students. Although various methods were considered, for this study the values of the number of blocks were added to the tangible blocks using a Wizard of Oz method. For this method, a large set of number blocks with specific numbers expected to be used in the camp were created by the TA ahead of time. Students could then request a number block when they needed it. If the number block did not exist, the TA would create it directly for them. Each 3D printed number block (Figure 21) consisted of a blank block with an orientation marker and texture to which a Peel N’ Feel label was added with the

required value. The label consisted of both large text and Braille. As negative numbers were not represented in the Peel N' Feel stickers, a circular “bump” was placed before the Braille to indicate a negative number.

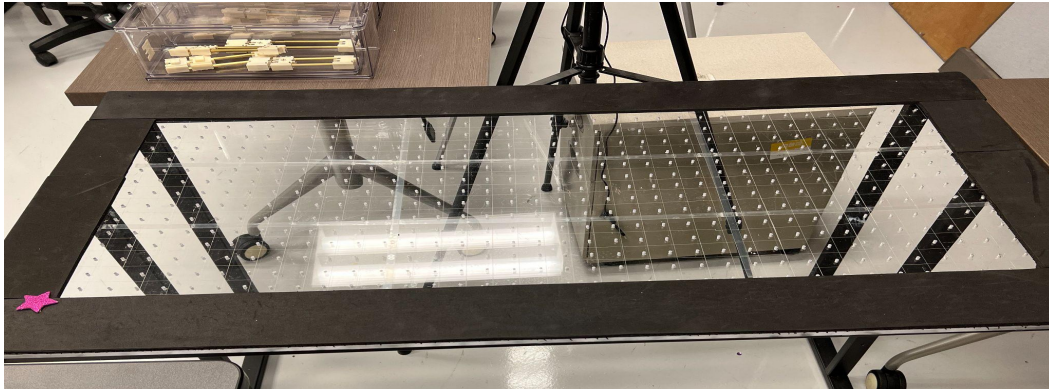


Figure 19. Code editor work surface



Figure 20. Overview of the code block drawer organization.

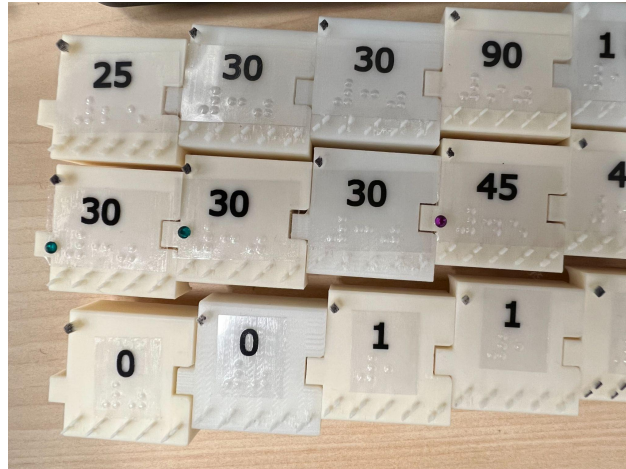


Figure 21. Examples of number blocks. The blocks are labeled with both Braille and large text. Negative numbers are represented by a circular “bump” in front of the Braille number.

The program execution display, or stage, consisted of a small mobile robot (Figure 22) designed with a cap by which the user could grasp it to follow the program execution. Springs were used between the cap and the robot base to minimize the effect of the user on the robot’s motion. The raised edge on the crown of the cap and the middle (red) bump on the brim were used to indicate the orientation of the robot.

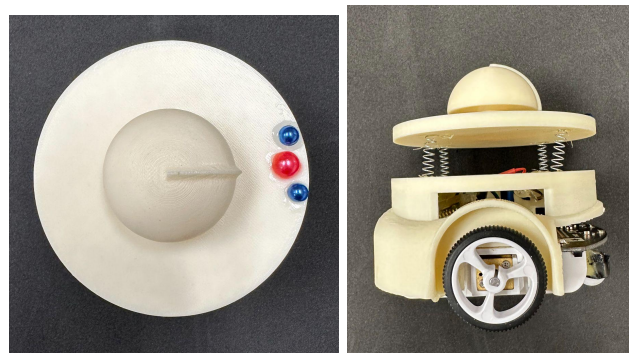


Figure 22. The robot used in this study. The top view (left) and side view (right). Top of the robot has a haptic marker to inform where it is facing.

The mobile robot acted on interchangeable backgrounds. Each background was a 36” x 29” acrylic sheet. Each sheet had a textured boundary indicating the border of the “stage” area, which was defined to match Scratch’s virtual stage area ($x = -240$ to 240 , $y = -180$ to 180). If the robot was programmed to go beyond the border, it would stop when the border was reached and

make an annoying noise. The surface of each sheet was also incised with an x-y grid, with grid lines spaced 25 “units” apart; no color was used to avoid visual clutter for low vision users. Additionally, all sheets had raised tape lines indicating the x- and y-axes as well as “tick ” marks corresponding to the grid lines on these axes. The center of the “stage” was indicated with a raised line cross. Colored raised lines and black textured areas were added to create different scenarios for each background. For example, in Figure 24, the gray lines indicate cave walls that the mobile robot should not pass through. The black textures on the tangible “stage” (e.g., Figure 25) were used in the same manner as colors in virtual Scratch. On backgrounds that had textures in the workspace, the x- and y-axes were made white as this did not interfere with detection of the textures (which were detected by the robot sensors through color). Otherwise the axes lines were black.

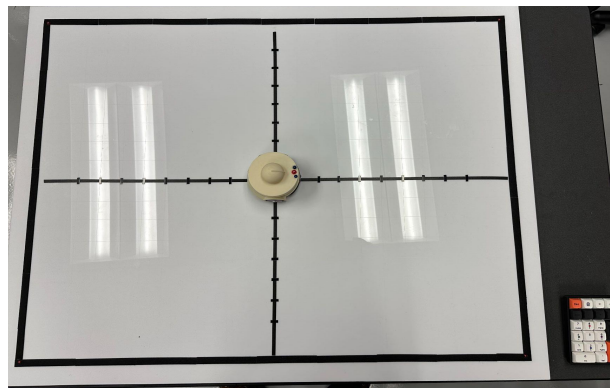


Figure 23. A mobile robot placed on a background with a coordinate system. A number pad is located in the lower right corner.

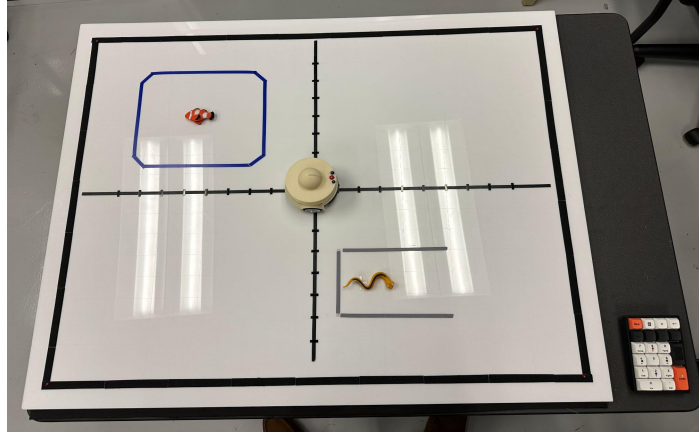


Figure 24. The final project of day 1 of the camp. The robot had to go into the area defined by the raised blue border to get the fish. They then had to “give” the fish to the eel without crossing the walls of the cave (indicated by raised gray walls). The location of the fish and eel are indicated by both figurines and raised +’s on the surface.

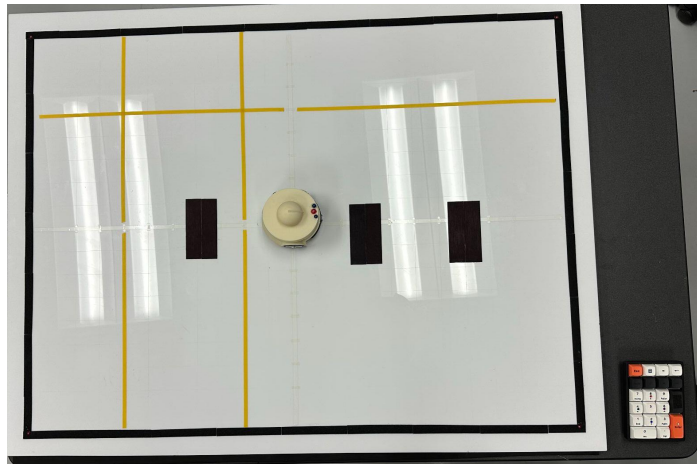


Figure 25. A background used for the first part of the second day. As textures (in black) are used, the x- and y-axis and the tick marks are made with raised white tape lines. The raised, yellow tape lines divide the surface into sub-areas for different practice problems.



Figure 26. The background used for the final project, which is an obstacle course. Again, as textures are present, the x- and y-axis and the tick marks are made with raised white tape lines. The horizontal raised silver lines indicate the boundaries of the robot's lane in the race. The vertical raised silver dashed line indicates the finish line. The black textures indicate obstacles that need to be avoided. The raised yellow lines indicate a portion of the course that acts like a wind tunnel.

The last part of the program execution display was the numeric keypad (shown in the bottom right corner in Figures 23-26). The numeric keypad was used by a participant to “produce” the event to start a code block (either the enter key or an arrow key on the numeric keypad). For the camp, the numeric keypad was not actually connected to anything. The activation of the program code, as well as the translation of the program built with the tangible blocks to the code running the mobile robot, was performed using a Wizard of Oz method. To do this, the TA read the code segments the participant wrote on the code editor surface, and typed the commands into separate programs that controlled the robot on a computer. The TA then watched the participant to see what key(s) they pressed on the keypad. If a key that was pressed was programmed, the TA executed the program on the robot that matched the key pressed. The enter key and the four arrow keys could be programmed as events. All five keys were indicated by round “bumps” placed on them. A sixth key, which had a textured area placed on it, could be

used to query the location and orientation of the mobile robot; this was meant to match a feature that exists on the virtual Scratch stage.

6.4 Curriculum

The curriculum that was developed was for a two day coding camp with one or two blind or visually impaired participants at a time. For the case studies analyzed in this thesis, participants experienced the camp one at a time. The coding camp taught students the programming concepts of: sequencing, events, if-else statements, variables, relational operators and logical operators, along with the concepts of incremental development and program debugging. The camp was a hands-on experience with the instructor describing what was to be done aurally while seated in front of a participant. The material was provided at the pace of the student, although all students were presented with the same material.

The overall structure of the camp was divided into Day 1 and Day 2. On Day 1, the participant was first asked to take a pre-test. Then they were given an introduction to computer science and coding, and about the importance of having a growth mindset when learning. The participant was then provided with a hands-on overview of the environment starting with the stage and mobile robot, including the coordinate system (the background in Figure 23). Then the participant was asked to explore the code editor work surface (Figure 19) and was explained its organization and tactile cues. They were then given a general overview of the organization of the drawers that stored the code blocks and block assemblies. For the remainder of the day, the participant was introduced to the enter key event block, then the motion blocks that used absolute coordinates (go to x,y, glide s seconds to x,y and point in direction d), and then relative coordinates (move x steps, clockwise rotate a and counterclockwise rotate a). There was a lunch

break either before or after the relative motion commands were taught. In the afternoon, the participant was also introduced to the concepts of sequencing, incremental development and debugging. At this time they were also introduced to the wait (wait s seconds) and beep (wait and beep s seconds) commands that could aid with determining when one command ended and another began. At the end of the day, the participant was presented with a story for which they had to program the robot to follow the narrative (Figure 24). After the project was completed and working, the participant was given a quiz on the concepts of Day 1.

Day 2 focused on the concept of conditional expressions. The day started with a review of the stage (program execution) and its coordinate system, the code editor work surface and its organizational cues, and the organization of the storage drawers and the code commands. Then participants were taught about if statements and if-else statements using the background of Figure 25 and the concept/code block of whether the robot is in a textured area or not. They were then introduced to variables; in particular, those for the coordinates of the robot (x,y) and the direction it pointed in (d). These were used to introduce the participant to the use of the relational operators less than and greater than in conditional expressions. There was a lunch break either before or after this concept was taught. In the afternoon, the participant was introduced to the use of the logical operators AND and OR in conditional expressions. They were then introduced to the scenario of being a game developer making a game that used the background in Figure 26. Participants were described what the different haptic feedback cues meant and how the robot was supposed to behave. As part of the process of incremental development, the participant was shown how to write separate pieces of code for individual arrow keys on the numeric keypad. After the project was completed and working, the participant was given a quiz on the concepts of Day 2, then a post-test and the exit interview.

For each of the concepts/commands described above, participants were first introduced conceptually to what the command did. They were then introduced to the symbol representing the command and where to find the corresponding code block/code block assemblies in the storage drawer structure. The participant was asked to bring the block/block assembly to the work surface and orient it correctly. Then the individual components, their location and cues (such as symbols and connections) were explained as the participant explored the block/block assembly. The participant then had one or more exercises in which they had to use the command. Rather than give coordinates for commands, a foam star or arrow was placed on the background to indicate what the robot should do. For example, a foam arrow pointing left at -100, -150 required that the participant program the robot to go to 100,-150 and rotate to point to 270 degrees. In some cases, participants were asked to explore the stage for textures and/or raised lines to determine the needed code expressions as well. Guidance and feedback were provided throughout, primarily by the instructor, but also, to some degree, by the teaching assistant.

6.5 Data Collection

Data collected from each student consisted of: (1) an audiovisual recording of the study participant interacting with the system and a post camp interview. Audiovisual recordings were made throughout the two day camp, except during the initial introduction to computer science, lunch breaks and exams. Two Logitech Pro Webcams were used to visually record the student interacting with the system. One camera recorded the participant interacting with the code editor work surface and was close enough to detect the details of the code blocks and their connections. The other camera provided an overview of the interaction with the worksurface, the storage drawers and the program execution space. To hear what was said clearly, Tascam digital audio

recorders were used with a clip-on microphone. Both the instructor and the student wore individual systems. Systems were synchronized by the TA clapping their hands at the beginning of the recording.

Eleven questions (Table 16) formed the basis for the post camp interview with the student. The interview was conducted by the teaching assistant and an audio recording was made.

Table 16. Interview Questions

#	Question
1	What did you like the best about your experience? Would you explain further?
2	What did you find the most helpful about the system in learning how to program? Would you give an example?
3	What did you like least about your experience? Would you explain further?
4	What did you find gave you the most difficulty in learning how to program with the system? Would you give an example?
5	Now let's talk more specifically about using the physical blocks to create a program. Describe your experience in being able to locate a type of block you wanted, whether in the drawers or on the code editor workspace. Was it easy to find the blocks you wanted? What would make it better (or if they say, yes, easy - even easier) for you? Was it easy to figure out what command a block represented? Again, what would make it better (or if they say, yes, easy - even easier) for you?
6	Now let's think back to how you assembled the blocks in the code editor workspace. What do you think are the strengths and weaknesses of both the workspace and how the blocks connected to each other in assembling them? Would you give an example (ask after they give each strength or weakness)? More specifically, could you tell when a code segment is being nested inside another code segment – for example that $x > 5$ is a code segment nested within the AND operator for the statement $x > 5$ AND $x < 10$.
7	The code blocks had a lot of features to help you assemble your program. Can describe any features that were not very helpful. Would you explain further? Are there any features you would like added to help build the code?
8	Imagine you were in a larger classroom where your teacher is explaining a sequence of code that everyone in the class has assembled in their code editor workspace. Please share your expectations of how you would follow along with the teacher when they refer to a location within the code. Do you think the cues provided in the workspace are adequate? Would you explain further? Are there any additional cues you would like added?
9	Now let's talk about using the physical stage with the mobile robot. How did you feel about writing programs about the mobile robot on the different physical backgrounds to learn how to program? Would you explain further?
10	What did you think of using the mobile robot with the different backgrounds to figure out whether your program worked or not and help you debug your program? Would you give an example?
11	Did your experience during the camp change your view about coding? Would you explain further?

6.6 Assessment of Audiovisual Data and Exit Interview

The main purpose of the assessment was to determine whether participants used the system in the way we designed it to be used. In particular, my analysis centered on assessing whether participants interacted with the code block assemblies and their embedded feedback as intended. If not, the analysis was meant to understand any problems with using the blocks and any needed improvements that should be addressed before scaling up the study to a larger number of students. The assessment was also intended to note ways that students may have been successful in doing tasks through unexpected means to evaluate if further support (e.g., through design modifications) for these strategies was warranted. The second purpose of the assessment was to determine whether the participants enjoyed their experience and if it has made them excited about programming.

The answers to these questions were determined by examining both the audiovisual recordings of participant interaction with the system and their audio responses to the exit interviews. Qualitative content analysis (Schreier, 2012; Saldana, 2021) was used to analyze the audiovisual recordings. Analysis of the exit interview used a less formal procedure, but considered frequency and intensity of the comments.

The procedure to perform the qualitative content analysis was to first develop a code book to code for the task (topic) being executed by the participant, the participant's performance and any strongly exhibited emotions. The development of the framework for the code book was primarily based on the a priori expected behavior of participant interaction with the system. However, the codes were further revised based on observation of the audiovisual material covering the first day of the coding camp for the participants. The code book also went through a refinement process through discussions with Dr. Pawluk and Bryson Goolsby, but this was not

comprehensive due to time constraints. In addition, due to time constraints, reflection and comparison to additional coders to solidify the validity of the code book was not made due to time constraints.

The framework developed for the topic codes can be described in terms of the steps in creating a program with the tangible code blocks. First the desired code block/block assembly needs to be brought to where the program is to be built. To do this, the student may find the block/block assemblies or code segments on the work surface and move them to where the code is being built, or they may need to go to the storage drawers to get the block/block assemblies from the appropriately labeled drawer and bring them to the work surface. At some point or at several points during this process, the student will need to identify the code block they intend to use.

For constructing the code segments on the work surface, the student will need to orient the block/block assembly correctly on the work surface as the blocks will not successfully connect unless they can validly connect together and they are both at the same orientation. The next step will be to place the block/block assembly to be inserted into the program code in the correct place. This may simply be a program command that is next in a vertical sequence of connected commands that are connected together with matching juts and notches at the top and bottom of the first block in its block assembly. For vertically nested statements (i.e., if statements) this will also potentially require the expanding and contracting of the vertical telescoping for the notches and juts of the underlying command block (i.e., if statement) to close around the vertical sequence of commands to be conditionally executed (either if True or if False), as blocks must connect both on their top and their bottom within a condition. For inserting single blocks or block assemblies into parameter (horizontal) slots of commands, this

will require both potentially expanding the horizontal telescoping and the proper placement of the to be inserted block/block assembly on the telescoping tubing. The telescoping must then be contracted so that the notches and juts that define the connectivity are connected together. A student may check whether the syntax will be/is valid before, during or after connecting code blocks into a program.

Finally, if the program segment built is syntactically correct but determined to be logically incorrect, an inserted block/block assembly may be disconnected or replaced. Returning blocks/block assemblies to their proper drawer after completion of a program was not included as this was primarily done by the TA and instructor at lunch time and/or the end of each day.

The codes for participant performance were developed for each task (topic) separately. This included expected methods (i.e., those methods that are expected to arise from the feedback provided in the code blocks) and observed methods. There were more than one expected method, and all were represented in the data tables presenting the results. It was also considered important to indicate the success rate for the method used; in particular, if the method could successfully be performed independently by the student, without aid from the instructor or TA.

As one of the objectives was to determine whether students were enjoying the camp, strong positive and negative emotions exhibited by the participants were also coded. The positive emotions that were coded were enjoyment and pride. The negative emotions that were coded were confusion and frustration. The emotions that were chosen to be coded were ones we expected would affect a student's willingness to persist in learning how to program.

After the code book was developed, the audiovisual data for the two participants was then viewed and coded by myself. The recorded two day camp was divided into time segments based on the change in the topics given in the previous paragraph. A new time segment was said to

start when behavior switched to a new topic (task) in the code book. If the previous behavior was not finished before the new behavior started, a new time segment was still added at the start of the new topic. However, the performance of the previous segment was noted only after that behavior was totally completed. For example, a student may place a block into a parameter slot and only contract the telescoping tubing on one side. They may then perform other behaviors. If they finish contracting the telescoping tubing on the other side before they try to execute the code, then the performance was marked as correct. Due to the limited amount of time to complete the analysis, reliability of the data coding was not assessed; however, the analyzer (myself) strived for consistency in the coding over time.

As the analysis framework had a strong mapping to the coding framework, the coding results were used to describe the data for each of the topics in the code book. The frequency of a particular code/performance pair was determined separately for day 1 and day 2 to examine if there were any behavioral changes over time, such as through learning. In addition, the temporal sequence of codes was further examined for patterns of sequences of two behaviors. The focus in examining sequencing was on the behaviors in the code book that were needed to create a program. The frequency of the sequences of two behaviors was determined in Python using the pandas data analysis toolkit (program provided in Appendix B). This program used the permutation of all the chosen behaviors and went through the data to find out the frequency of all the permutations of two behaviors. The repetition of single behavior was not recorded because it was not expected to be informative.

6.7 Results

6.7.0. Example Programs Constructed. The code segment on the left is an example of the final (debugged) code constructed on Day 1 by one of the participants in relation to the concept

of sequencing. The sequence starts with an Enter key event and uses a combination of absolute and relative movement commands to do the task laid out by the instructor. The code segment on the right is an example of the final (debugged) code constructed on Day 2 by one of the participants in relation to the concept of conditional expressions. The code segment starts with an Enter key event and uses a combination of absolute and relative movement commands. It also uses a simple Boolean evaluation in an if-statement and a relational operator to complete the task.

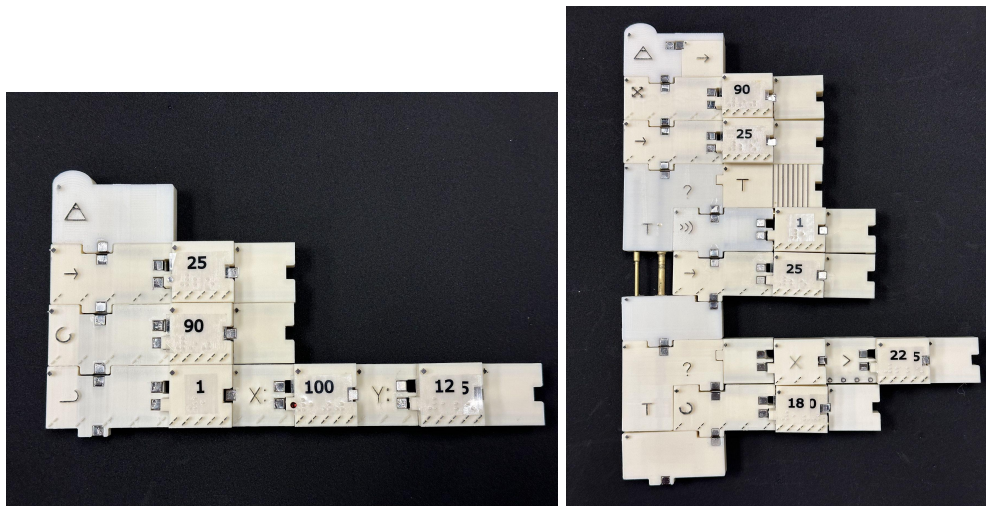


Figure 27. Example of sequential code (left) and nested code (right).

6.7.1. Student 1. Performance.

Identifying a Block. The results from examining the performance of the student for those segments categorized as “identifying block” are reported in Table 17. Four codes were used to differentiate different methods of how this behavior was performed. The first, and expected method of determining a block/block assembly was by applying pressure and rubbing the identifying haptic symbol for the block/block assembly with one or more fingers (Figure 28). The identifying haptic symbol was how the blocks were designed to be identified. The second method, which was not expected, but still allowed the student to independently identify the

block/block assembly was to identify them by exploring the edges of the block/block assembly to identify its overall shape. This may be the overall shape of a single block (Figure 29) or of a block assembly (i.e. the C shape of if-then block or the E shape of if-then-else block). The use of this method may have partly to do with overall shape being used by the instructor to help identify some blocks/block assemblies. However, with the intended expansion of the block commands to more Scratch commands, this method may result in some confusion between blocks/block assemblies that are similar. The third method was seemingly by memory as the student used the block without performing any exploration of the symbols or overall shape. However, this method was also cognitively demanding, separate from actually learning how to program. Finally, the fourth method was when the student verbally asked the instructor to tell them what block they were holding. Although this was not discouraged during the camp, it does raise some concern for students being able to independently identify blocks/block assemblies: something they are likely expected to do in some scenarios where an instructor will not be present or nearby.

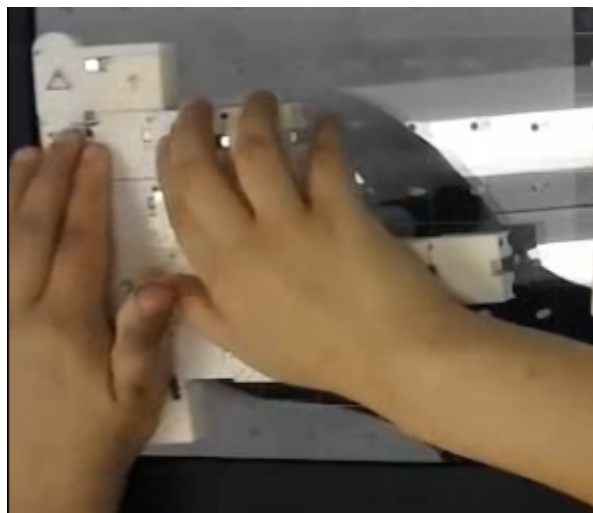


Figure 28. Example of a student identifying a block assembly by feeling and identifying the symbol on the first block with their left hand.

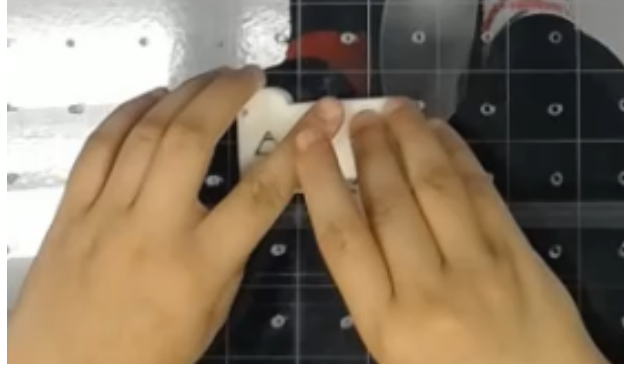


Figure 29. Example of a student identifying a block by feeling the unique shape of an event block along the upper edge with their left hand.

The successful or unsuccessful identification of symbols were recorded along with which symbol it was. Most of the time the student identified the block/block assembly commands correctly. Although the expected method (identification by feeling the tactile symbol) occurred the most, surprisingly, much identification occurred by memory. The use of memory did decrease during the second day, although there is no increase in the other methods used (suggesting that perhaps the student reduced the number of commands they worked with on Day 2).

The results also show that there were instances where the student did not identify block commands correctly. The symbols that the student identified incorrectly and then asked the instructor for help are organized in Table 18 (with a comparison to the total number of times each symbol was identified throughout each day of the camp). Overall performance did not appear to improve over time: errors were consistent across both days. The participant stated that the two symbols that they confused the most with each other were clockwise rotation and counterclockwise rotation. One other symbol the student had difficulty identifying was the beep command. As indicated by the higher incidents of asking the instructor for help, this was less due to confusion with another symbol and more due to confusion with identifying the symbol. This may be due to the symbol used being more familiar to sighted rather than BVI individuals.

The times the student most asked for help from the instructor was when she was building larger code segments. Sometimes this was because she was truly confused with which block she was holding, but other times it appeared she asked the instructor to save time (although one could not tell whether she would have been successful finding the code block on her own).

Table 17. Student 1: Block identification methods and performance of each day of the camp

Student 1	Day 1 total:	Identify correctly	Identify incorrectly	Day 2 total	Identify correctly	Identify incorrectly
Identify by feeling symbols	38	33	5	27	24	3
Identify by feeling overall block shape	2	2	0	2	2	0
Identify from memory	15	14	1	6	4	2
Identify by asking instructor	7	7	0	5	5	0

Table 18. Student 1: Symbols identified incorrectly and asked the instructor for identification

Student 1	Day1 total:	haptically incorrect and asking instructor	Day 2 total:	haptically incorrect and asking instructor
beep	5	2	3	3
CCW	7	4	1	1
goto	12	2	0	0
point	3	1	3	1
step	15	2	8	2
CW	2	1	2	1
Y	0	0*	1	1
AND	0	0*	2	1
Less	0	0*	2	1

* These blocks were not used on Day 1.

Orienting a Block/Block Assembly. In order to assemble the code blocks correctly, they needed to be oriented correctly. To help with block orientation, all blocks were designed with a raised dot in the top left hand corner. Thus it was expected that the easiest way for a student to orient a block would be to feel for the raised dot and orient the block so that the dot can be found in the top left hand corner (Figure 30). This method was labeled “Orienting block using raised dot”.



Figure 30. Orienting the block by feeling for the raised dot located in the top left corner with left thumb

Several other methods could be used, although they require more cognitive effort. It was of interest to determine if the student used any of these methods and why they may be preferred at times. One method was by using the juts and notches along the edges of a block and by which the blocks/block assemblies connect to each other. For commands, this would be by feeling for notches on the right hand side of the block and no connections on the left hand side of the block. For operators, numbers and variables, this would be by feeling for notches on the right hand side, juts on the left hand side and no connections on the top and bottom. Another possible method was by using the identifying symbol for a block by first finding and identifying the symbol, and then orienting it correctly based on memory. Using this method alone could potentially be

problematic as some identifying symbols could only be differentiated if the correct orientations were used (e.g. the AND and OR). Finally, it was of interest whether the “pegs” on the back of the blocks were used to orient a block in the top down direction (they could not be used for orienting the front face of the block). For this behavior, the student was expected to identify the up/down orientation based on finding a raised square when feeling the block surface facing upward. The main purpose behind this query is that the original function of the “pegs” is no longer needed, but it was important to determine if the “pegs” were being used by students for other purposes.

The primary method by which the student oriented the blocks on the work surface was using the raised dot in the upper left hand corner of each block (Figure 30), which was effective (as shown in Table 19). What typically occurred was that she would search on the block/block assembly for the raised dot and then really push on the dot when she found it in the upper left corner. This was the expected method of orienting a block/block assembly. However, it was noted that there were times that the student did not check for the orientation of the block and proceeded to use it (incorrectly).

Table 19. Student 1: Methods and performance of orienting blocks.

Student 1	Total: Day 1	correct	incorrect	Total: Day 2	correct	incorrect
Orienting block using raised dot	62	62	0	68	64	4
Orienting block using juts and notches	0	0	0	0	0	0
Orienting block using symbol	0	0	0	0	0	0
Orienting block using peg	0	0	0	0	0	0

Finding a Block/Block Assembly in a Drawer: Before orienting a block/block assembly to be added to a code segment, it needs to be found first: either on the work surface where the student left it or in the drawer organization. Each of the drawers had a label on their front that the student was expected to feel (Figure 31), at the very least, for confirming the identity of the block/block assembly found within. The student

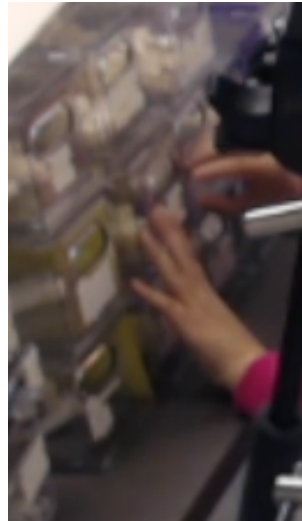


Figure 31. Example of feeling symbols of blocks on the bin.

would apply pressure or rub the plate with the symbol on the front of the drawer in order to do this. There was also a structure to the organization of the drawers. For the larger drawers, the Event commands were found in the first column, the absolute Motion commands in the second column, the relative Motion commands in the third column, the timing commands in the fourth column, etc. The organization of the drawers was explained to the student by the instructor as they were learning new commands. It was expected that students would use this organization to help them more quickly locate the command they were interested in finding. A student could feel the divisions between the drawers when exploring with their hands and would use this method to count the number of columns and rows. This could be tedious for some of the columns further from the start (there were 7 columns and between 2-3 rows in each column).

Two codes were used: 1) “Finding by RC and symbol” to indicate the student felt for the row-column organization of the drawers to help guide them to the approximate location of the code block/assembly they wanted and then felt the symbols on individual drawers in that area to verify they found the correct block/block assembly and 2) “Finding by RC” to indicate the student felt for the row-column organization but did not feel the label, rather they immediately opened the drawer to take a block/block assembly. The first method was how the drawers were designed to be used. The second method was also acceptable although there may be some concern as to why the student did not quickly check the label on the front of the drawer for verification rather than relying solely on memory of the row-column organization. A third code, “Search contact”, was for when the student started feeling the drawers to first find a label on a bin and then feel the symbol on the label, starting with the bin that was closest to them and then searching until they found the block/block assembly they wanted. The use of this method is a concern as it takes much longer than the other methods and does not take advantage of the design of the storage organization; this suggests that a more effective method of organization is needed.

Two other codes were used for when the instructor or TA was involved in guiding the student to the correct drawer. Sometimes the student was able to get to the approximate location of the block/block assembly they wished to find by feeling the row-column organization but needed help from the instructor to find the correct drawer. This was labeled as “Find by RC with GI”. Sometimes the student asked for help from the beginning from the instructor, this was labeled “GI”. The instructor used a variety of instructions including, for example, “move one column over” or “up higher”. These behaviors suggested that the student was not comfortable using the methods that were designed to find the correct block/block assembly. However, it is not

possible to determine if the student would have found the block/block assembly on their own and how frustrated they may have been at that point.

When finding the blocks in the organization of drawers, Student 1 had to be guided more by the instructor at the beginning of the camp. However, as time progressed, she used her memory to locate the correct drawer, typically asking for confirmation by the instructor when she found it (Table 20). Asking for confirmation was not the expected behavior (it was expected that she would be confident in finding the correct block/block assembly on her own), although it was unclear how much of this strategy was due to convenience. There also appeared to be a learning curve for finding the blocks/block assemblies by row-column as, on Day 2, the participant needed somewhat less instructor guidance for finding blocks/block assemblies she used on Day 1 as compared to those that were new on Day 2.

Table 20. Student 1: Method and performance of finding blocks in organization bin

Student 1	Day1: total	GI	Find by RC	Find by RC with GI	RC and symbol	Search contact	Day2 total:	GI	Find by RC	Find by RC with GI	RC and symbol	Search contact
Finding blocks on organization	22	8	8	6	0	0	23	6	5	12	0	0

* GI= Guided by instructor, *RC = Row and column

Bringing the Block/Block Assembly from the Drawer to the Workspace. After finding the correct drawer, the next expected step was for the student to take a block/block assembly from the drawer and bring it to the workspace. The behavior during this step was coded as it was observed that the student had some problems with blocks/block assemblies inside a drawer (duplicates of the command/block) sticking to each other. We wanted to see if this was a significant problem. Three codes were used to describe the student's performance in taking a block to the workspace.

One code “Had problems with block sticking” described the instances when the blocks within a drawer stuck to each other. For example, when first taking a block assembly from a drawer, the student was confused by blocks sticking together as she was not sure what a single block assembly was. The instructor would help her by telling her she had more than one block/block assembly. In later cases, she was able to recognize the block assemblies sticking herself and separate them. Instructor assistance was not needed for the single blocks. The second code “Had problems with large block size” was a potential concern for the larger block assemblies that were potentially more difficult to take out of a drawer. The third code “Bring blocks to workspace no problem” was when there were no issues in getting the blocks/block assemblies from the drawers.

When Student 1 took out the blocks from the drawers and brought them to the workspace, there were two occurrences of block/block assemblies sticking magnetically to other blocks/block assemblies in the drawer (Table 21). The student had to wiggle the block/block assembly to get a single block/block assembly out from the bin, but was able to do this on her own. She did have a bit of confusion as to whether she had one or more block/block assemblies.

Table 21. Student 1: Performance of bringing blocks from the bin to workspace

Bring blocks to the table	total	Bring blocks to workspace no problem	Had problem with blocks sticking	Had problem with large block size
Day 1	22	20	2	0
Day 2	23	23	0	0

Identify Block/Block Assembly Connections and Verify Connected Code Command. Once a block/block assembly was in the workspace and oriented correctly, one step that could occur before a participant tried to connect blocks together was to identify the type of connection ahead of time (whether on the block/block assembly to be inserted or the block assembly to which it

will be attached). Two codes were used. The expected method that was used, “identifying connection using notch and juts”, was to feel for the location of notches and juts along the edges of a block (Figure 32) to determine if there will be a match. An alternate method for those block assemblies that had parameter slots or vertical slots for statements was to feel for an empty slot by finding only bare telescoping tubing between blocks in a block assembly; the code used was “identifying connection feeling slots”.



Figure 32. The student checked the location of a jut using her left hand.

After a connection is made, whether immediately following or later, a student may check the connection between blocks to determine that they are correctly connected to confirm what was created. This is an important component of the tangible blocks as they are meant to provide feedback about valid/invalid connections. The main method that was designed to provide this feedback was the use of magnets in combination with the location of the juts and notches. Blocks that are correctly connected will produce a magnetic attraction to each other. Blocks that are incorrectly positioned will not. These behaviors were labeled with two codes. “Detect connection by tugging blocks” occurred when a student tugged or wiggled a block to see if it was securely connected. “Detect lack of connection with lack of magnetic pull” followed a similar method on the part of a student but resulted in them not feeling a magnetic pull. Another easy to

use behavior that was expected to occur was when sometimes a student left a parameter slot or a nested statement slot empty. When they felt the code blocks in sequence, the student was able to detect the empty slot by feeling only the bare telescoping tubing between blocks in a block assembly and knew that the code was incomplete. This was coded by: “Detect lack of connecting with empty slot”. It was also possible that more subtle methods could be used to determine an incorrect connection: first, by feeling for the gap created by a jut not connecting to its adjacent block, and second, by feeling empty notches and juts. These were also coded.

All the methods were evaluated as to whether a student identified/detected the connection correctly or incorrectly (Table 22 and Table 23). As student 1 was correct all of the time, the correctness of a method was not shown in either table.

At the beginning of Day 1, Student 1 checked for notches and juts before connecting the blocks (Table 22). However, as time progressed, she checked less frequently. On Day 2, she never checked. It appeared that, as she got experience with the haptic feedback created by the magnets when a successful connection occurred, she stopped checking prior to connecting blocks; instead, she relied on the feedback while actually trying to connect the blocks together.

When checking her program, Student 1 used the easier methods. She used the designed method of the combined magnetics and notch/jut locations to determine whether the connections were correct and also felt for empty slots. The latter was also a conscious component of the design that provided feedback about the number of parameters needed for a command.

Table 22. Student 1: Methods of identifying connections

	Day 1: total	Day 2: total
Identify connection using notch and juts	6	0
Identify connection feeling slots	0	0

Table 23. Student 1: Methods of detecting connections

	Day 1: total	Day 2: total
Detect connection by tugging blocks	1	2
Detect lack of connection with lack of magnetic pull	1	1
Detect lack of connecting with empty slot	0	2
Notches and Juts	0	0
Feeling for the gap	0	0

Expanding or Contracting Telescoping To Insert Block/Block Assembly. Another task that almost always needed to occur to attach two blocks together was to expand or contract the telescoping of the underlying block assembly to fit the block/block assembly to be inserted in a parameter slot. If students smoothly expanded the telescoping, placed the block/block assembly on the telescoping and then contracted the telescoping, then block placement was indicated as successful. Sometimes the telescoping got stuck when expanding or contracting and needed to be wiggled to be unstuck (Figure 34). If the student wiggled the telescoping to get it unstuck this was coded as “Place block by wiggle blocks”. Sometimes the student needed help to do this and the instructor did it for them. This was coded as “Place block with instructor wiggling telescope”. The blocks had indentations on their backs in order to sit securely on the telescoping. We also added a code in case a student needed to wiggle a block in order to get it to sit securely on the telescoping tubing.

Two additional codes were included. The first was where the result of the effort was for the block to not sit securely on the telescoping rails. The second was when the student failed to contact the telescoping to fit the inserted block/block assembly (Figure 35).

The only condition that was considered desirable was placing the block successfully without the need to wiggle anything. There was a concern that if a student had to wiggle the telescoping or a block, this could cause frustration over time. It is also possible that students not in the study sample may not be able to overcome the problem. This was why when the instructor had to intervene was even more of a concern. It suggested that, if a student was working independently, they may not overcome this problem. If they fail to place a block securely on the telescoping this may affect the tracking of the blocks on the workspace. Forgetting to contract the telescoping meant that they did not completely finish building code, and this behavior may interfere with the automatic translation of blocks to executable commands in the completed system.

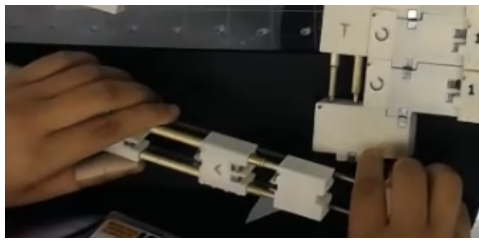


Figure 33. Example of placing block successfully

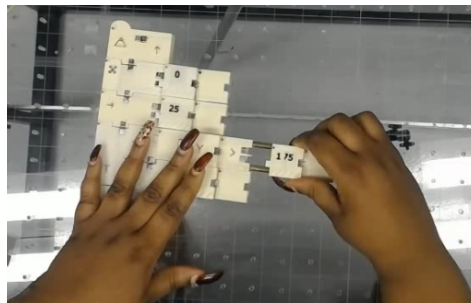


Figure 34. Example of block getting stuck on the telescope and student unable to contract the blocks

When placing blocks on the telescoping tubing in the horizontal direction, the student was successful most of the time (Table 24). However, when the telescoping tubing needed to be expanded more to fit larger block assemblies, there were cases where blocks got stuck on the telescoping at the point where the tubes changed in size (on the horizontally expanding assemblies). The issue was that blocks placed on the right side of the tube size change would get stuck when the telescoping contracted because the tube diameter on the left side was bigger: the blocks could not slide over the bump. In these instances, the instructor had to wiggle the telescoping to “unstick it” to contract the telescoping for the student.

There were also a few instances where the student forgot to contract the telescoping to fit the block, although the block itself was correct in its slot; this was partly because she immediately went to execute the program without checking the code connections.

When expanding and contracting the vertical telescopes, there were no issues observed. This was likely because blocks were not placed on the telescoping, only beside it.



Figure 35. Example of Fails to contract telescope.

Table 24. Student 1: Performance in adjusting telescoping for the block/block assembly to be inserted into the underlying block assembly

Placing block	total	Place block successfully	Place block by wiggling telescope	Place block by wiggle blocks	Place block with instructor wiggling telescope	Fails to put block	Fails to contract telescope
Day 1	43	36	0	0	3	0	4
Day 2	59	51	0	0	4	0	4

Connecting Blocks. Probably the most significant task was connecting the blocks as the syntax of the program structures was embedded into the mechanical constraints imposed by the connections. Although it was possible for a student to check whether two blocks could be mechanically connected together ahead of time (i.e., identify connection), more often than not, students just directly tried to connect the blocks they felt should be in sequence. Because students were not asked to speak out loud about what they were doing (which would have slowed them down significantly), it was difficult to ascertain the cues they were using by directly watching the video of the behavior. However, from the exit interview, both Student 1 and Student 2 indicated that they always used the presence or absence of magnetic feedback to determine whether a connection was successfully connected or not. The students both felt the pull of the magnets of a correct connection forming and the snapping sound that occurred. When no additional behaviors occurred simultaneously this was coded as “Success:magnetic cue”.

If the student tried to feel the edges of the blocks for matching notches and juts on the mating pieces, as well as using the magnetic attraction, then the behavior was labeled “Success:feeling edge”. It was also possible that the instructor had to guide the student in making the connection, this was labeled as “Success: with guidance”. If the student failed to make a connection when the syntax was valid or failed to determine the connection they were making was incorrect when it was incorrect, performance was labeled as “Failed to connect”.



Figure 36. Connecting correctly mating blocks. When a block was close enough to the correct connection point (left), the magnets attracted and connected the block (right). During this process, the student could feel the magnetic attraction. They could also hear a ‘snap’ sound, giving audio feedback of the correct connection. In this example, the student

When connecting a block/block assembly into a parameter/statement slot, Student 1 almost always used the magnetic cue (Table 25). This was the expected, and designed for, behavior when users try to connect blocks to form code lines with valid syntax. The two instances in which the student failed was when she failed to determine that a block should not be connected. The student did feel something was wrong because they did not feel the magnetic attraction, but they did not know what was wrong. In these cases, the instructor had to inform the student that the blocks could not connect because they were in the wrong orientation.

Table 25. Student 1: Performance in connecting block

Connecting block	total	Success: magnetic cue	Success: feeling edge	Success: with guidance	Failed to connect
Day 1	81	81	1	0	0
Day 2	108	106	0	0	2

Disconnecting, Replacing Blocks. Disconnecting blocks typically consisted of removing an inserted block/block assembly from a code line (Figure 37, left). The code line could be

embedded as part of an overall code segment consisting of a series of commands to be executed. Sometimes the blocks that were disconnected were multiple code lines. Replacing blocks occurred during debugging, where a student decided that they need to change the program either while constructing the code segment or after following the robot on the stage. Replacing blocks typically consisted of removing an inserted block/block assembly from a code line and then inserting another block/block assembly in its place (Figure 37, right). Performance for both “disconnecting block” and “replacing block” was coded as either a success or failure. Inability of a student to disconnect/replace a block/block assembly will result in failure to correctly debug a program independently. The inability to disconnect a block would be most likely due to the strength of the magnets and indicate that their strength needs to be reduced to be usable. The result showed (Table 26) that the student did not have any problems when disconnecting/replacing blocks from each other.

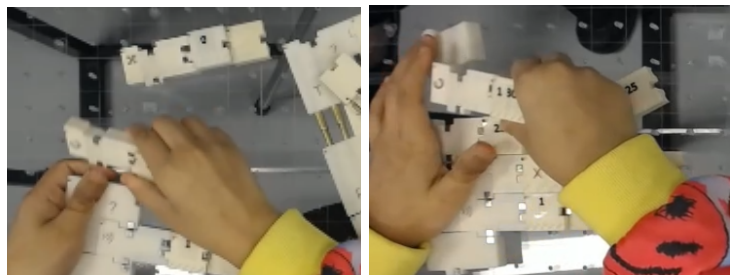


Figure 37. Disconnecting the blocks (left) and Replacing the Block (right)

Table 26. Student 1: Performance in disconnecting and replacing block

	Total: day1	Success	Fail	Total: day2	Success	Fail
Disconnecting block	8	8	0	48	48	0
Replacing Block	27	27	0	14	14	0

Finding a Block/Block Assembly on Work Surface. Students were also encouraged to leave blocks, block assemblies and segments of code that they had created or brought into the

workspace, but were currently not being used, on the work surface so that the blocks could be reused at a later time. One action that then occurred was for the student to find code on the work surface that they wanted to reuse in their current program. The work surface was designed by Bryson Goolsby with a variety of haptic feedback, which was intended to facilitate finding code. The work surface was divided into 9 sections (organized in a 3x3 matrix), divided by raised, transparent tape lines. These 9 sections could be used for reference for finding blocks and communication with the instructor. Evenly spaced grid lines were etched on the work surfaces spaced approximately 1" apart (the spacing of a standard block). Each grid cell had a hole drilled into it in the middle of the right hand side that could be used for counting adjacent placed blocks. A piece of ¼" black foam was placed around the edges of the transparent work surface with a width of approximately 3.5". A sparkly purple foam star was also placed in the lower left hand corner of the foam edge as a reference.

The methods which the student could have used were represented as codes and are reported in Table 27. If students found a block purely based on their memory without using any of the indicators on the workspace or feeling the haptic labels of the blocks they came into contact with, it was coded as "Using memory". If they explored the surface only feeling the haptic block labels to find the block/block assembly they wanted, it was coded as "feeling symbols of blocks: trial and error". If they explored the work surface only feeling the overall shape of the blocks in a block assembly and the connections on the blocks/block assemblies to find what they wanted, this was coded as "feeling block structure".

However, it was of most interest to determine if participants used the structured passive haptic feedback provided on the work surface when constructing their code. The first of the codes focused on the haptic feedback provided by and on the border. The code "using border as

edge” indicated that the border was used as a reference edge to find a block/block assembly (e.g., the block can be found close to the inside edge of both the left and top borders). This is in contrast to “using the border as area” where the border itself was used as a holding area from which to retrieve a block. The behavior was different than using the border as a reference edge as the user would feel for the foam area as a whole rather than tracing the edge between the foam and the hard work surface. Using the star on the border indicated a strategy of placing blocks near the star (either on the border or on the work surface) and retrieving a block by finding the foam start and then searching the area around it.

The second set of codes focused on the haptic feedback on the transparent work surface itself. The code “using section lines” would be coded when the strategy seemed to involve first finding and “counting” section lines before doing a local haptic search in a section area (bounded by section lines) for the block/block assembly. This was expected to involve focusing on finding, and potentially tracking, section lines, followed by localizing a search for a block to a particular section area. Although the main focus of providing grid cell markers (i.e., the holes) was to allow users to count lines in response to an instructor asking the class to “look at line 7 of the code segment”, it is possible that a user would remember where they placed a block by how many grid cell markers it was away from another block. This was coded as “using grid cell markers”.

The performance of the method was divided into: “success without guidance”, “success with guidance from instructor using coordinates” and “success with guidance from instructor using hot/cold”. If a student was able to find a block/block assembly on their own then it was labeled as “success without guidance”. If a student was given help by the instructor, it was indicated as success with guidance from the instructor. The instructor used two strategies. The first strategy was for the instructor to indicate what section the block/block assembly was in, for

example, the top-middle section. The nine sections were indicated by: left, middle, right and top, center, bottom. The second strategy, the “hot/cold” strategy, was when the instructor indicated if the student was getting closer or further away from the block/block assembly they were searching for.

It was expected that a student would use the passive haptic feedback designed into the system through previous design iterations (i.e., section lines, grid lines and grid markers). Regardless of the methods, the need for guidance by the instructor was a concern as students would not likely always have an instructor or TA immediately adjacent to them to help them find block/block assemblies. However, one of the limitations of this study was that intervention from the instructor happened even when students did not specifically ask for help. This makes it difficult to know whether a student would ultimately succeed on their own given no intervention.

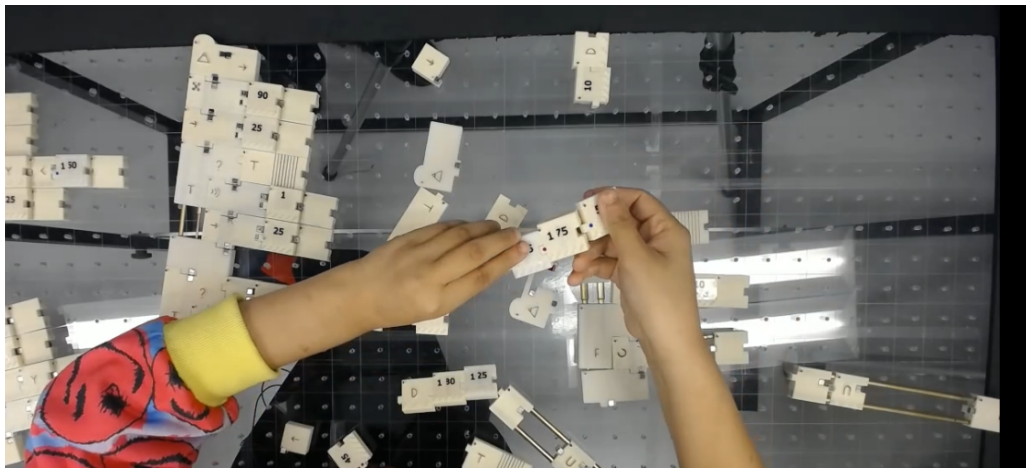


Figure 38. A student searching for the block in workspace by feeling symbol trial and error

When finding blocks/block assemblies/code segments on the work surface, Student 1 relied on her memory most of the time. She did not use any cues provided on the work surface. However, the student had to be guided by the instructor to find the blocks/block assemblies/code segments on the work surface over half the time. Often this involved the instructor guiding the

student by the “hot/cold” method when the student was close to the blocks/block assemblies/code segments she needed. If the student was completely lost and further away from the blocks/block assemblies/code segments, the instructor guided her through using “coordinates”.

Table 27. Student 1: Methods and performance of finding block on workspace

Ways to find block in workspace	Day 1: total	Success without guidance	Success with GI: coordinates	Success with GI: hot/cold	Day 2: total	Success without guidance	Success with GI: coordinates	Success with GI: hot/cold
Memory	15	6	5	4	17	8	6	3
Using border as edge								
Using border as area	0	0	0	0	0	0	0	0
Using star on border	0	0	0	0	0	0	0	0
Using section lines	0	0	0	0	0	0	0	0
Using grid cell markers	0	0	0	0	0	0	0	0
Feeling symbols of blocks: trial and error	0	0	0	0	6	2	3	1
Feeling block structure	0	0	0	0	0	0	0	0

* GI= Guided by instructor

Frequency of Two Code Sequences. The audiovisual recording was also explored for time sequences of behaviors when building a program. The behaviors that were the focus for this examination were the ones involved directly in building a program (Table 28). The frequencies of all the two sequence permutations of the behaviors in Table 28 were generated in Python using the pandas data analysis toolkit for Student 1. The resulting frequencies for sequences of two different behaviors with the highest frequency of occurrence are given in Table 29.

Table 28. Behaviors of students that used to identify the pattern.

Behaviors identified	
identify	find block
orient	bin
connect	identify connection
disconnect	place
check connection	bring to table

* find block= find block within worksurface, *place= placing block, *bring= bring blocks from drawer to table

Table 29. Sequence of behavior by student 1 during Day 1 and Day 2.

Day 1			Day 2		
1st Behavior	2nd Behavior	frequency	1st Behavior	2nd Behavior	frequency
place	connect	52	place	connect	64
orient	place	50	orient	place	51
bin	bring to table	35	connect	orient	31
connect	find block	20	bin	bring to table	25
bring to table	orient	20	connect	bin	14
connect	bin	17	bring to table	orient	14

* find block= find block within worksurface, *place= placing block, *bring= bring blocks from drawer to table

The result (Table 29) shows that the highest frequency two behavior sequence was placing the block on the work surface and then connecting the block: this is understandable in that the placing the block and connecting the block are the key components for building code segments. The expected behavior before placing the block is orienting the block, so occurrences of other behavior before placing blocks were reported to compare with the expected behavior (Table 30): the results confirm that for Student 1, the expected behavior is what primarily occurs. We were also interested in determining whether the expected behavior of the student first orienting a block/block assembly when it is brought from its drawer to the table. We used a similar procedure for placing a block and reported the result on Table 31 and results show that the expected behavior occurred the most.

Table 30. Student 1: Sequences of behavior when “place” is the second behavior

Day 1			Day 2		
1st Behavior	2nd Behavior	frequency	1st Behavior	2nd Behavior	frequency
orient	place	50	orient	place	51
identify	place	1	connect	place	10
find block	place	1	identify	place	3
			disconnect	place	1

* find block= find block within worksurface, *place= placing block, *bring= bring blocks from drawer to table

Table 31. Student 1: Sequences of behavior when “bring to table” is the first behavior

Day 1			Day 2		
1st Behavior	2nd Behavior	frequency	1st Behavior	2nd Behavior	frequency
bring to table	orient	20	bring to table	orient	14
bring to table	bin	9	bring to table	bin	4
bring to table	find block	5	bring to table	find block	2
bring to table	disconnect	1	bring to table	connect	2
			bring to table	identify	2

* find block= find block within worksurface, *place= placing block, *bring= bring blocks from drawer to table

6.7.2 Student 1: Emotions. As one of the objectives was to determine whether students were enjoying the camp, strong positive and negative emotions were coded. Each emotion was coded as having either a high or low level. The positive emotions that were coded were enjoyment and pride. Enjoyment was coded when a participant expressed delight in what they were doing. If enjoyment was expressed verbally, such as saying “this is fun”, it was coded at a high level. If enjoyment was only expressed through a facial expression while focused on the task and system, such as grinning when the robot did what they wanted, enjoyment was coded as low level. Pride was coded when a participant expressed a sense of accomplishment in what they did. If pride was expressed verbally, such as saying “I did it!”, it was coded at a high level. Pride was coded at a low level if the participant appeared unwilling to disconnect any blocks from a working program when the task is complete and/or they are making a new program.

The negative emotions that were coded were confusion and frustration. If confusion was expressed verbally, such as saying “this is confusing”, it was coded at a high level. If confusion was only expressed through facial expressions, such as being puzzled or overwhelmed, it was coded as low level. Frustration was coded when confusion appeared to turn to anger. If frustration was expressed verbally, such as angrily telling the instructor that they are not being helpful, it was coded at a high level. If frustration was expressed only through facial expressions, such as being angry or disgusted, it was coded as low level.

The main limitation with this code book design was that it was difficult to differentiate a person’s facial expression between enjoyment and pride, or frustration and confusion. In most cases, the person’s verbal expression before or after the given facial expression was used to determine the category of the emotion.

In determining the frequency at which these emotions are expressed it was intended to determine if participants did or did not enjoy interacting with the designed system. This was an important consideration as the system is not only intended to be accessible but to potentially encourage those not necessarily interested in programming to get interested in it. However, it is also acknowledged that confusion and frustration are normal components of learning. What is an acceptable level of confusion and frustration can be difficult to determine.

The frequency at which these emotions occurred for Student 1 are reported in Table 32. The student showed enjoyment, while interacting with both the coding blocks and the mobile robot, but mostly when she was interacting with the robot. The instance in which she showed pride, was when she determined that she successfully completed the story scenario at the end of day 1. Student 1 also showed confusion and frustration on a few occurrences. One issue she had trouble understanding was the coordinate system used on the stage; in particular, the negative

directions of the axes and the relationship of the coordinate system to angular values. In other instances, she did not understand why the robot moved as programmed.

Table 32. Student 1: Emotions displayed during each day of the camp

Student 1	Day 1	Day 2
Enjoyment, High	12	6
Enjoyment, Low	5	13
Pride, high	0	1
Pride, low	0	0
Confusion, High	1	2
Confusion, Low	3	4
Frustration, High	1	2
Frustration, Low	2	3

Student 1: Exit Interview. In the exit interview, the student said she enjoyed the camp because she was doing the things that she would not normally do. She liked the coding blocks, including all the haptic feedback that was provided such as the symbols, magnets, telescoping, and the raised dots. For example she stated: “It was easy to tell which block goes with which block, so it didn’t cause confusion on that end”. She also liked that she was coding to control the robot’s movement and found controlling it “intriguing”. Although she had problems finding blocks in the drawers, she did not think it was hard to do. She just thought that she needed more time and experience. The part of the system she did not like were the backgrounds, which she stated was because her haptic detection skills were not the best and she was unfamiliar with tactile diagrams. She found that determining where the axes were in the background was problematic. She suggested using more aggressive haptic markers.

When the student was asked about her strategy for debugging code, she said the primary way that helped her to debug the code was trying to execute it, seeing what happened and then modifying the code. When asked about how to find a specific part of a code sequence, the

student said she would count blocks instead of relying on surface markers. Despite a positive overall experience, the student did not plan to pursue programming in the future, but sees potential for the system to aid BVIs in learning programming.

6.7.2 Student 2: Performance.

Identifying a Block. The results from examining the performance of the student for those segments categorized as “identifying block” are reported in Table 33. Most of the time the student identified the block/block assembly commands correctly. The student used the expected method (identification by feeling the tactile symbol) most of the time. On Day 2, she also used the overall shape of a block assembly two times to identify a block assembly; this was specifically when she was identifying the if-then, and if-then-else block assemblies. However, it should be noted that the instructor did describe the overall shape of the if-then and if-then-else block assemblies to help orient the blocks and locate the notches and juts.

The results also showed that there were instances where the students did not identify the blocks/block assemblies correctly. The symbols that the student identified incorrectly are shown in Table 34 (with a comparison to the total number of times each symbol was identified throughout each day of the camp). Student 2’s overall performance did not appear to improve over time: her errors were consistent across both days. Although she was pretty efficient in haptically exploring symbols and identifying them, she found some symbols difficult to identify even when she did so correctly. This was discovered in the exit interview where Student 2 commented that they had difficulty discriminating between the clockwise rotation and counterclockwise rotation commands. She also had trouble with point and beep, as shown in Table 34. This was supported by the fact that she asked the instructor for confirmation about what she was holding.

Table 33. Student 2: Block identification methods and performance of each day of the camp

Student 2	Day 1 total:	Identify correctly	Identify incorrectly	Day 2 total	Identify correctly	Identify incorrectly
Identify by feeling symbols	27	26	1	42	41	1
Identify by feeling overall shape	0	0	0	2	2	0
Identify from memory	0	0	0	0	0	0
Identify by asking instructor	0	0	0	2	2	0

Table 34. Student 2: Symbols identified incorrectly and asked the instructor for identification

Student 2	Day1 total:	haptically incorrect and asking instructor	Day 2 total:	haptically incorrect and asking instructor
beep	0	0	2	1
step	2	0	9	0
point	2	1	5	1
Y	0	0	4	1

Orienting a Block/Block Assembly. The primary method by which Student 2 oriented the blocks on the work surface was using the raised dot in the upper left hand corner of each block, which was successful for her for the most part (as shown in Table 35).

Table 35. Student 2: Methods and performance of orienting blocks.

Student 2	Total: Day 1	correct	incorrect	Total: Day 1	correct	incorrect
Orienting block using raised dot	37	37	0	88	87	1
Orienting block using juts and notches	0	0	0	0	0	0
Orienting block using symbol	0	0	0	0	0	0
Orienting block using peg	0	0	0	0	0	0

Finding a Block/Block Assembly in a Drawer: When finding the blocks in the organization of drawers, Student 2 had approximately an equal number of instances where she was able to find a block/block assembly on her own and where she asked the instructor for help (Table 36). This was consistent across the two days. The one thing that differed was her increased confidence when finding block/block assemblies on her own as, on the second day, she reduced the need for confirmation once she found them. This was the expected behavior in that the student was able to independently use the row-column organization method to find the drawer for a block/block assembly (as similar variables/commands were grouped together by column) followed by confirmation by feeling the tactile symbol for that block/block assembly at the front of the drawer. There also appeared to be a learning curve for finding the blocks/block assemblies. Although the duration of time to find blocks was not measured, it was observed that on Day 2 the participant needed less time finding blocks/block assemblies she used on Day 1 as compared to those that were new on Day 2.

Table 36. Student 2: Method and performance of finding blocks in organization bin

Student 2	Day1: total	GI	Find by RC	Find by RC with GI	RC and symbol	Search contact	Day2 total:	GI	Find by RC	Find by RC with GI	RC and symbol	Search contact
Finding blocks in organization	11	5	1	3	2	0	42	19	0	3	20	0

* GI= Guided by instructor, *RC = Row and column

Bringing the Block/Block Assembly from the Drawer to the Workspace. When taking out the blocks from the drawers and bringing the blocks to the workspace, no problems with copies of the block/block assemblies sticking to each other was observed (Table 37).

Table 37. Student 2: Performance of bringing blocks from the bin to workspace

Student 2	total	Bring blocks to workspace no problem	Had problem with blocks sticking	Had problem with large block size
Day 1	11	11	0	0
Day 2	35	35	0	0

Identify Block/Block Assembly Connections and Verify Connected Code Command. As with any participant, one step that Student 2 could use before they tried to connect blocks/block assemblies together was to haptically feel the block/block assembly ahead of time around the location of the connection to predict whether there would be a matching fit. On Day 1, she did not check at all before trying to connect blocks (Table 38). On Day 2, she checked once: this was when preparing to connect a long, complex expression to the if conditional input in order to figure out the location where she should make the connection.

She exhibited similar behavior for when she chose to detect connections of already connected blocks. She did not check the connections on Day 1, but checked once when building a large code segment controlling the robot for the maze problem. In that case, she put a complex

statement into the wrong place and was confused for a few seconds. She then checked the connections and felt no magnetic pull, which led to her readjusting the connection.

Table 38. Student 2: Methods of identifying connections

Student 2	Day 1: total	Day 2: total
Identify connecting using notch and juts	0	1
Identify connection feeling slots	0	0

Table 39. Student 2: Methods of detecting connections

Student 2	Day 1: detected	Day 2: detected
Detect connection by tugging blocks	0	0
Detect connection with lack of magnetic pull	0	1
Empty slot	0	0
Notches and Juts	0	0
Feeling for the gap	0	1

Expanding or Contracting Telescoping To Insert Block/Block Assembly. Another task that almost always needed to occur to attach two blocks together was to expand or contract the telescoping of the underlying block assembly to fit the block/block assembly to be inserted in a parameter slot. The student was successful in operating the telescoping all of the time (Table 40).

Table 40. Student 2: Performance in placing blocks on telescope

Placing block	Total	Place block successfully	Place block by wiggling telescope	Place block wobble blocks	Place block with instructor wobble telescope	Fails to put block	Fails to contract telescope
Day 1	30	30	0	0	0	0	0
Day 2	52	52	0	0	0	0	0

Connecting Blocks. When connecting a block/block assembly into a parameter/statement slot, the participant always used the magnetic cue (Table 41). The salient cue appeared to be not only the magnetic force created by the magnets, but also the ‘snap’ sound created by the two blocks forming the magnetic connection. This was the expected behavior when users tried to connect blocks.

Table 41. Student 2: Performances in connecting block

Connecting block	total	Success: magnetic cue	Success: feeling edge	Success: with guidance	Failed to connect
Day 1	53	53	0	0	0
Day 2	98	98	0	0	0

Disconnecting, Replacing Blocks. The student also did not have any problems when disconnecting blocks from each other (Table 41).

Table 42. Student 2: Performance in disconnecting and replacing block

	Total: day 1	success	Fail	Total: day 2	success	Fail
Disconnecting block	15	15	0	11	11	0
Replacing Block	9	9	0	13	13	0

Finding a Block/Block Assembly on Work Surface. We also wanted to investigate the method the student used to find blocks/block assemblies and code segments she left on the surface for reuse. The methods which the student used and her performance when finding code on the work surface are reported in Table 43. When finding code on the work surface, the student relied on her memory most of the time, but had to be guided by the instructor a few times. Unlike student 1, student 2 performed trial and error exploration, feeling the symbols on the blocks, to identify the

block that she wanted to use. She would search the approximate location of the block that she used before, relying on her memory, and then feel for the symbols on the blocks in that area to identify the block that she wanted to use. She was successful most of the time, but she still needed guidance when there were a lot of blocks on the workspace and when she forgot the general location of the block. She would be guided with coordinates when she was completely lost, and be guided with hot/cold when she was searching in the approximate location of the block.

Table 43. Student 2: Methods and performance of finding block on workspace

Ways to find block in workspace	Day 1: total	Success without guidance	Success with GI: coordinates	Success with GI: hot/cold	Day 2: total	Success without guidance	Success with GI: coordinates	Success with GI: hot/cold
Memory	8	7	1	0	23	18	3	2
Using border as edge	0	0	0	0	0	0	0	0
Using border as area	0	0	0	0	0	0	0	0
Using start on border	0	0	0	0	0	0	0	0
Using section lines	0	0	0	0	0	0	0	0
Using grid cell, haptic markers	0	0	0	0	0	0	0	0
Feeling symbol: trial error	1	1	0	0	17	11	6	1
Feeling code Structure	0	0	0	0	0	0	0	0

* GI= Guided by instructor

Frequency of Two Code Sequences. Again, as with student 1, the audiovisual recording of student 2 was also explored for time sequences of two behaviors (sub-categories). The procedure was exactly the same as that of student 1.

Table 44. Sequence of behavior by student 2 during Day 1 and Day 2.

Day 1			Day 2		
1st Behavior	2nd Behavior	frequency	1st Behavior	2nd Behavior	frequency
place	connect	55	place	connect	52
orient	place	55	Block orientation	place	50
bin	bring to table	29	bin	bring to table	35
connect	orient	23	connect	find block	20
connect	disconnect	21	bring to table	Block orientation	20
disconnect	orient	18	connect	bin	17
find block	identify	16	connect	Block orientation	16
connect	bin	15	disconnect	Block orientation	16
connect	find block	15	find block	identify	15

* find block= find block within worksurface, *place= placing block, *bring= bring blocks from drawer to table

The results (Table 44) show that the highest frequency two behavior sequence was placing the block on the work surface and then connecting the block, as with student 1 and as expected. The expected behavior before placing the block is orienting the block, so occurrences of other behavior before placing blocks were reported to compare with the expected behavior (Table 45): the results confirm that for student 2, the expected behavior is what primarily occurs. We were also interested in determining whether the expected behavior of the student first orienting a block/block assembly when it is brought from its drawer to the table. We used a similar procedure for placing a block and reported the result on Table 46 and results show that the expected behavior occurred the most.

Table 45. Student 2: Sequence of behavior when “place” is a second behavior

Day 1			Day 2		
1st Behavior	2nd Behavior	frequency	1st Behavior	2nd Behavior	frequency
orient	place	55	orient	place	50
identify	place	1			

* find block= find block within worksurface, *place= placing block, *bring= bring blocks from drawer to table

Table 46. Student 2: Sequence of behavior when “bring to table” is a first behavior

Day 1			Day 2		
1st Behavior	2nd Behavior	frequency	1st Behavior	2nd Behavior	frequency
bring to table	orient	13	bring to table	orient	20
bring to table	bin	7	bring to table	disconnect	1
bring to table	find block	4	bring to table	bin	9
bring to table	connect	3	bring to table	find block	5
bring to table	disconnect	2			
bring to table	identify	1			

* find block= find block within worksurface, *place= placing block, *bring= bring blocks from drawer to table

Student 2: Emotions. The same method of examining strong positive and negative emotions was used as for student 1. The frequency at which these emotions occurred are reported in Table 47. The student showed enjoyment mostly when she was interacting with the robots. She showed high enjoyment by saying “yay!” or “this is interesting!” It was observed that she displayed enjoyment while building code itself more frequently than student 1. The instance that she showed pride, was when she was able to solve the final problem on day 2 by creating a large code. She displayed her emotion by saying, “we did it! This was really fun.”

Table 47. Student 2: Emotions displayed during each day of the camp

Student 1	Day1	Day2
Enjoyment, High	5	8
Enjoyment, Low	4	4
Pride, HIGH	0	1
Pride, LOW	0	0
Confusion, HIGH	0	0
Confusion, LOW	0	0
Frustration, HIGH	0	0
Frustration, LOW	0	0

Student 2: Exit Interview. In the interview, the student said that she liked the idea of programming with blocks, and the use of textures and symbols to identify the blocks. The blocks were accessible, and helpful. She felt all the features were useful. She liked the idea of a raised dot for orienting blocks, and thought that it was “brilliant”. She thought that finding blocks in the storage drawer organization was not hard, although she had to ask for confirmation a few times. She believed that, if she got more experienced, she would have fewer problems. She did find the way the drawers were organized (having similar block types in adjacent locations) was helpful. She commented that the counterclockwise and clockwise command symbols were too similar, but overall symbols were readable.

She also found that finding blocks on the work surface was easy; the difficulty was that there could be a lot of them on the work surface. However, she said that she did not use the features of the workspace, but indicated that other people might. The participant also suggested that having an empty drawer or bin to place blocks/block assemblies that a user is not expecting to use right away would be helpful as it would get the blocks/block assemblies off the work surface, but it would still provide a smaller location than the drawers in which to find blocks a user may end up wanting later. She also wanted a bigger workspace so she could write bigger programs.

She said how the code blocks connectivity was designed was helpful, such as what happens when blocks go together versus not go together (magnetic cue). She liked the idea of the telescoping tubing to expand parameter slots. She was able to tell which assemblies were nested by searching for the middle block of the operator. Identifying the errors was easy to determine, and the feedback used was easy to interpret: she said it was obvious to her when blocks did not belong together.

The student found that manipulating the code blocks helped her learn and the use of the robot helped her debug her program. As she was able to remember what her code was supposed to do, if the robot did something wrong, she would locate the part of the code that corresponded to that action. Her strategy to locate that part of the code was to first find the event block associated with that code segment. Then she would count the lines from there to where she expected the problem to be. In terms of the code execution, she found that feeling and hearing the robot move was interesting

Her view on coding changed after the camp. She said that she was now considering learning more about coding and possibly getting into a STEM field.

6.8 Discussion

The overall experience during the code camp for both students was positive. They enjoyed manipulating the blocks and controlling the robots. Although student 1 did not change her mind on coding after the camp, she mentioned how it could help other BVI students learn how to program. She also was excited that she actually did successfully create programs, which she did not think she would be able to do. The second student said she would consider coding in the future.

Both students were able to orient the blocks correctly using the implemented design features and, then, were able to connect the blocks without any major issues. Both the analysis of the audiovisual recording and the exit interview showed that they found the magnetic cue and the “snap” sound when blocks successfully connected together helpful. The results also showed that both students were able to disconnect and replace blocks successfully, which indicates that the magnetic attraction was not too strong to interfere with disassembling code segments.

However, a few issues occurred that suggest that some of the design features of the code blocks and organizational methodology needed improvement. These were: 1) the readability of the haptic symbols of the code block commands, 2) the functioning of the telescoping tubing and 3) the placement of used block/block assemblies and/or code segments. For the first issue, although the students used most of the symbols well, a few were problematic. The biggest problem was confusing the clockwise and counterclockwise symbols, as they did not feel there were significant haptic differences between them. Some symbols, such as the one for the point command, were also hard to remember as it was not intuitive. The symbol for the beep command also needed improvement as: although the symbol used was a common one for sighted users, it is not one that our BVI participants were familiar with. The code blocks would benefit from a redesign of the symbols, especially if the code block commands are expanded to include Scratch blocks in the Sound category and additional Control block assemblies. The addition of more code block commands means that more care will be needed to ensure that the symbols are easy to differentiate from each other.

The problem with the telescoping tubing was the part where the tubing changed its size as it, inevitably, created a little bump. Unfortunately, the code blocks could get stuck trying to slide across the bump, which prevented the telescoping tubing from contracting. This caused a few instances where a student was unable to complete the code on her own because the telescoping was not able to contract. The student needed the instructor or TA to wiggle the block assembly in order to get the telescoping to successfully contract. Although there were only a few instances when this occurred, the students felt stalled until this could be resolved. In extreme cases, the students might stop using the system due to frustration. We could instruct them to

connect the blocks in a specific way to avoid this issue, but we cannot expect students to follow directions all the time.

The third issue was what to do with blocks after they were used. The system was designed to make the work surface tinkerable, as on the virtual Scratch code editor. This meant encouraging the students to leave used blocks/block assemblies and code segments on the work surface for reuse. However, differences between touch/haptics and vision (i.e., the field of view in particular) made it much more difficult for BVI users to find blocks and code segments left on a tangible work surface compared to sighted users on a visible, virtual work area. In fact, the primary strategy used by the BVI students to locate blocks or code segments placed on the work surface was by memory. The use of a search strategy was secondary due to the time consuming process of serial exploration with touch. This meant that when the number of code blocks on the work surface was small, the students were able to locate the block they were looking for without problem by memory. However, when the number of blocks on the workspace got larger, it was much harder to remember where all the different blocks were located and the student had to serially explore the blocks to find the ones they needed. The second student suggested that it would even be easier to search through used blocks or code segments placed in an empty bin on the workspace than physically explore the work surface spatially. Various options need to be considered to allow the students to more easily find code on the work surface independently.

Analysis of the audiovisual recordings for sequences of two behaviors showed that the main two behavior (task) sequence the participants followed was orienting the code blocks before placing them on the work surface. This sequence that arose naturally from the design of the system was essential to connect blocks correctly together. It suggests that the students realized the importance of orienting the blocks before trying to connect them.

Although student interaction with the mobile robot executing the code was not analyzed from the audiovideo material, it should be acknowledged that the use of the robot was primarily what made both students excited about programming. They liked the idea of controlling the robot and stated that it made learning more interesting. Both students talked about the robots being easy to follow on the execution display and helpful for debugging programs. However, student 1 thought the raised lines to indicate the x and y axes, and the units along them, needed to provide stronger haptic feedback. There were also instances where student 1 displayed frustration and confusion due to having trouble understanding the coordinate system being used, particularly the negative part of the axes and the angles. This was, in part, because student 1 had limited experience with tactile graphs and these mathematical concepts. In the future, the curriculum should be adjusted for a student's level of mathematical understanding to include these concepts.

6.8.1 Limitations in the Study

One of the limitations of the study was that adults were used for the assessment instead of middle school students and data from only two participants was collected. The participant who was 18 was similar to the expected skill levels of the targeted BVI middle school students in that she knew Braille, was familiar with tactile diagrams and only recently graduated from school. The other participant was only recently blind, did not know Braille, had never used tactile diagrams and had not been in school for some time. Although the second participant was not typical of most students, we are interested in lowering the hurdles for all students and she provided a challenging case. In a future iteration of the study, a larger number of participants, drawn from the target age group, will be recruited. However, the two case studies presented here are with participants with very different backgrounds, providing some consideration for variability between participants.

Another limitation of the study was that the participants took the 2 day camp one at a time. This allowed the instructor and the TA to focus on the one student present. Although this is typical for when new technology is introduced, it does present some limitations when compared to “normal” classroom conditions. The instructor and TA were both able to be very responsive to a student and also intervene without prompting when they saw the student struggling. This was particularly helpful for the participant who did not know Braille and never used tactile diagrams as she relied on help for interpreting the Braille numbers and tactile symbols; although these lack of skills are less common in school age BVI students. It also presented a bias in that the instructor could individualize instructions to the single student, and their current workspace and execution display. For example, rather than asking a student to “find line 3 in the code segment”, the instructor may say “move your hand downwards.....now stop”. This may have resulted in some design features not needing to be utilized to perform tasks, even though they may be useful in other situations (i.e., a regular classroom) where the customized instructions could not be used by the instructor. In particular, the feedback designed for navigating the code editor work surface and the drawer system need to be further studied in larger class sizes to determine their utility and challenges.

There were also some potential limitations due to the current state of the system being tested, in particular, the inclusion of Wizard of Oz methods. The most likely limitation was with the method used to provide the number blocks: this allowed students to have easily created blocks with effective feedback (Braille and large print) for any number as the TA created any needed number blocks. A non-Wizard of Oz solution is likely to either restrict the availability of numbers (e.g., a limited number of pre-made blocks with Braille and large print) or use methods that may not provide feedback (e.g., no Braille and large print, with the value for a block being

entered through the numeric keypad). The first solution may result in preventing students from implementing their program the way they want due to unavailable numbers and the second solution may result in more memory work. On a positive note, automating the translation from the tangible code segments created to the mobile robot execution will likely alleviate the confusion due to the human “Wizard” incorrectly performing this translation.

There were also limitations in the degree of saliency of the materials that could be used for the background: something one of the participants remarked upon. This limitation was due to the robot’s inability to move across even moderately textured surfaces, which limited what types of materials could be used. Also, the robot was programmed to detect the textured surfaces by detecting dark colors, which made dark colors unavailable for use on the axes for those backgrounds. The BVI students did not like when the axes were provided in lighter colors. Finally, although better than most educational robots in terms of accuracy, the mobile robot still created issues in accuracy over longer periods of time and required the TA to reset the robot between programs. All of these issues are planned to be addressed by the construction of a mobile robot with the required specifications rather than using off-the-shelf technology.

Although not specific to the objectives mentioned at the beginning of the chapter, the study was also limited in terms of assessing whether BVI students were able to learn programming concepts. This was primarily due to the duration of the coding camp which lasted only 2 days. As such, despite tests and quizzes being given, they were not used to analyze changes in performance.

6.8.2 Limitations in the Analysis

The qualitative content analysis performed also had some limitations due to the time constraints. First, the focus on the analysis was only on the interactions of the participants with

the coding blocks. This was limited in two ways: 1) we were actually interested in participant interaction with the entire system, and 2) the components of the system do not work in isolation. Due to the latter issue in particular, it is possible that some aspects of the use of the code blocks were not captured. Second, the reliability of the coding was not assessed. Although re-analyzing the data at a later time to assess reliability was planned, time constraints prevented this from happening. Reliability could also be done in the future with the use of another coder; however, a more detailed code book is needed for them to follow the methodology used in this study.

There were also two further limitations to the analysis of the audiovisual data. First, I could not attend the 2 day camp in person to observe the participants. As the cameras and audio did not necessarily capture the details of all behaviors, there may be some aspects of the camp missed in the analysis. Second, the performance for some behaviors could not be determined as all outcomes occurred immediately and it was unclear which steps led to the outcome. This would have been more clear if the participants were asked to talk out loud to describe what they were thinking. This could have also captured more cognitive aspects of program creation. However, this would have significantly affected the learning experience as it would have slowed activities significantly.

6.9 Conclusions

The results show that the students were using the code blocks/block assemblies in the way we designed them to be used and were enjoying using the overall system throughout the camp. They were able to complete the planned curriculum and program projects successfully. However, the code block design showed a few weaknesses that need to be addressed in the

future. The first was that some of the haptic symbols for the code commands, most notably the clockwise and counterclockwise rotations, were found to be hard to distinguish from each other, which caused confusion. The second was that sometimes the telescoping tubing did not contract properly because of its interaction with code pieces sitting on the tubing. The third was that it was not as easy to nonvisually find code blocks and segments left on the tangible workspace as compared to visually finding code in the virtual code editor workspace. The next iteration of the design will need to address these problems, particularly to ensure that BVI students are able to use the code blocks to build programs on their own without any intervention by a teacher or TA (such as what would occur at home or in a busy classroom).

The limitations of the current case studies also need to be addressed through further human studies. First, participants need to be recruited from the target age group of middle school students, as well as in sufficient numbers to represent the variation in the population. Second, the analysis should also include a more rigorous evaluation of the reliability and validity of the code book developed. Finally, the code book should be extended to include the interaction with the entire system and be refined.

Future studies should also consider a more realistic approximation to a normal classroom with multiple students and lasting for a longer duration. However, the analysis of the two case studies given in this chapter suggests that the tangible code block design developed does hold promise as an accessible means for BVI students to learn how to code.

Chapter 7. Conclusion

This thesis is part of a larger project to make the Scratch programming environment accessible to BVI students to enable these students to experience the lower barriers to programming that Scratch offers. This is intended to increase opportunities for all BVI students to become interested in and prepare themselves for STEM fields. The main contribution of my work was creating a tangible code block design that: 1) was easy to perceive and understand by both BVI and sighted students, 2) retained the reduced need to struggle with syntax of Scratch, 3) allowed code construction through action, 4) and co-construction with other BVI and sighted students, and 5) could create moderately sized programs at low cost. The main difficulties in translating the visual, virtual code blocks into physical, tangible code blocks were that: 1) the visual perceptual dimensions that were used are not effective in the haptic domain, and 2) several actions that were programmed to occur in the virtual environment cannot occur in physical environments. In this dissertation, we discussed the motivation behind our research, the design processes in developing the tangible code blocks, the final design features and details of the design of the blocks, and experiments conducted to test the efficacy of the blocks.

The design of the code blocks focused on the development of low cost passive tangible blocks with: haptic feedback providing effective identification for non-Braille users (as only a small fraction of the BVI population knows Braille), an inherent design that embedded program syntax into the connectivity of the code blocks, and a method to allow nesting of expressions for parameters and operands, and expansion of the statement component of control blocks from a single to multiple statements. The design and development process went through two main iterations and involved input from stakeholders, particularly BVIs, and assessment by our targeted (BVI) population.

The most unique contributions from the first design iteration were: 1) the use of local edge shape (notches and juts) of the blocks themselves to provide selective connections that implemented the syntax used in Scratch, and 2) the use of telescoping tubing to allow for nested operator expressions to be used for operands and parameters of functions, as well as the statement blocks of control commands. These concepts were implemented in the first design iteration for prototype block assemblies for the mathematical, relational and logical operators. The use of the prototype tangible block assemblies with tangible blocks for boolean and numeric variables was compared to a text based method for the two main programming tasks of assembling a line of code and debugging a line of code. BVI participants were required to perform these tasks for both simple and more complex operator expressions. Their performance and feedback on the usability of each method was recorded. The results indicated that BVI participants produced correct code significantly more often when doing the tasks with the code blocks than with the text method. The use of tangible blocks was particularly advantageous for participants who were not very familiar with using a keyboard, as they were able to spend significantly more time on actual programming compared to finding the blocks/keys on a keyboard.

One weakness of the code block design used in the first iteration was that it was difficult for BVI participants to easily determine connectivity between validly connecting code blocks. It was hypothesized that this could be improved by increasing the strength of the magnetic connection when blocks are validly connected. The second design iteration of the code blocks studied the effect of embedding different degrees of magnetic attraction within the local shape connection on the ability to non-visual determine connectivity between blocks. BVI participants performed a set of studies to determine the effect of magnetic strength on non-visually

determining the connectivity between blocks, where the inserted block varied in size and weight. Results suggested that magnetic attraction was critical for participants to perform at high accuracy. Ease of use increased, not only with the use of magnets, but with the strength of the magnets used in the study. There were some differences in ease of use between blocks of different sizes and weights but the results converged for the strongest magnetic strength used in the study.

The second design iteration also considered the implementation of a larger set of commands. Many of the commands had syntax consistent with that developed for the operators. However, two notable exceptions were the “set” and “change” commands, which added further restrictions on the syntax. Therefore a method was considered to implement the additional syntax rule for these commands without having to modify all code commands. In particular, we considered the use of different “stopper” designs to prevent numeric literals from being placed in the left slot of a “set” or “change” command, which could only accept a variable. A study was performed to compare the use of 4 design alternatives by BVI participants for determining the correct syntax. The study found that the length of the stopper was important for both performance and ease of use. Both metrics were significantly better when the stopper was long enough that no magnetic field at all could be felt.

The improved block design from the second design iteration was then applied to a set of code block commands that included most of the motion commands, key press events, conditional statements, the operators, the touching texture (color) expression, as well as variables x, y and direction. In addition to the design elements tested through the design iterations, telescoping was also used vertically in the implementation of conditional statements. Sufficient repetitions of each of the different code block commands were made for utilization in a 2 day camp.

The final question asked was whether BVI participants interacted with the complete system in the way we intended it to be used. My component focused on analyzing their interaction with the tangible code blocks. For my analysis, two adult BVI users interacted with the entire system in a two day coding camp. The programming concepts covered were: sequencing, incremental development, debugging, relational and logical operators, variables and conditional statements. The main objective of the analysis was to determine if the code blocks, when used in a learning context with the whole system, were used in the manner for which they were designed. The second objective was to determine if the participants were enjoying interacting with the system and were excited about programming. The results suggested that the participants did interact with the code blocks as intended, but minor improvements could be made to improve ease of use. Two areas in particular that should be improved are: (1) providing easier to identify tactile symbols for some of the code commands and (2) improving the design of the connection between the telescoping tubing and their blocks in the assembly. Overall, the participants enjoyed learning through physically constructing programs and haptically following a small, mobile robot. They also had a more positive attitude towards coding at the end of the camp.

7.1 Future Work.

In the future, the two day code camp needs to be conducted in a more refined manner, addressing some of the limitations of the current study caused by the limited time frame. Participants need to be recruited from the target age group of middle school students and in a

larger number. The code book should be better defined, and assessments of reliability and validity should be conducted. This will provide stronger confirmation as to the usability of the system and provide a better understanding on how the intended students interact with the system.

For the system design, results from these two case studies suggest that the haptic symbols used to represent the blocks should be re-evaluated for their usability. This should especially consider the clockwise and counterclockwise rotation commands, the point command and the beep command. The fabrication of the block assemblies with the telescoping tubing also needs to be improved to provide better alignment of the tubing to avoid problems with sticking. Finally, to provide more salient backgrounds, the control of the mobile robot needs to be improved so that it is capable of traversing more significant textures and the color sensor used to differentiate better between colors.

Probably the most needed work to be done on the system design is to automate the parts of the overall system that were implemented in the coding camp using Wizard of Oz methods. One place in which a Wizard of Oz method was used was in generating the number blocks. The tradeoffs between providing a fixed number of blocks at given values that can use Braille and large text, and a method that will allow a user to assign any value to a number block, but with no Braille or large text, need to be considered.

The second place that a Wizard of Oz method was used was in translating the code assembled by the BVI student on the work surface to program code that could be executed by the mobile robot. Each code block/block assembly does have a unique tracker associated with it for which the code block ID and location on the work surface (x, y and θ) can be provided. This, as well as information about the properties of the different code commands can be used to

convert the created tangible code block sequences into program code that is automatically downloaded to the mobile robot.

In performing the translation from tangible blocks to mobile robot code execution, it will be important to maintain the properties of liveness and tinkerability of Scratch. For example, Scratch allows for programs to be executed even if the code is incomplete (such as by not adding all the parameters a command may need) or if extra code snippets are left on the virtual editor workspace. Scratch does not perceive an error, but will not execute incomplete lines of code or code snippets that do not begin with an event. Scratch also allows the program code and/or parameters to be changed while the program is run: “In highly tinkerable construction kits, there is a very short interval of time between making a change and seeing its effect.” (Resnick and Rosenbaum, 2013).

Finally, after addressing the design issues mentioned, a larger and longer study of the target BVI population interacting with the system is needed to determine the effect of the system on promoting BVI student interest in programming and their ability to learn programming concepts. This should be done in a more realistic classroom setting. This includes learning over a longer time duration for which students will truly be able to be assessed in their learning. This also includes a more realistic composition of people in the classroom. In particular, the fact that there will be more than one student and the student to teacher ratio is unlikely to be as high as in the current study. In addition, it is common for BVI students to go to schools with sighted students, and so learning in a classroom with sighted students is also important to assess. This includes the consideration of collaboration between BVI students and sighted students.

References

- Ackermann, E.K. (2004). Constructing Knowledge and Transforming the World. In M Tokoro and K. Steels (Eds.) *A Learning Zone of One's Own: Sharing Representations and Flow in Collaborative Learning Environments*. IOS Press, Washington, DC.
- Bdeir, A. (2009). Electronics as material: littleBits. Proceedings of the Third International Conference on Tangible and Embedded Interaction, Cambridge, UK.
- Blikstein, P., Sipitakiat, A., Goldstein, J., Wilbert, J., Johnson, M., Vranakis, S., Pedersen, Z. and Carey, W. Project Blocks: designing a development platform for tangible programming for children. Google Website: projectbloks.withgoogle.com.
- Computer and Information Technology Occupations : Occupational Outlook Handbook: U.S. Bureau of Labor Statistics. (2022, April 18). *U.S. Bureau of Labor Statistics*. <https://www.bls.gov/ooh/computer-and-information-technology/home.htm>
- Ducasse, J., Mace, M., Serrano, M., Jouffrais, C. (2016). Tangible Reels: Construction and exploration of an interactive and collaborative map for the visually impaired. *CHI '16 Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 2186-2197.
- Edman, P.K. (1992). Tactile Graphics. American Federation of the Blind Press, New York.
- Erickson, W., Lee, C., von Schrader, S. (2018). Disability Statistics from the American Community Survey (ACS). *Ithaca, NY: Cornell University Yang-Tan Institute (YTI)*. Retrieved from Cornell University Disability Statistics website: www.disabilitystatistics.org
- Goolsby, B., Pawluk, D., Kim, H.W. and Fusco, G. (2021). A Tangible Block Editor for the Scratch Programming Language. *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, Yokohama, Japan.
- Gordon, Claire C. et. al 1988 Anthropometric Survey of U.S. Personnel: Summary Statistics Interim Report. March 1989.
- Kelly, A., Finch, L., Bolles, M., & Shapiro, R. B. (2018). BlockyTalky: New programmable tools to enable students' learning networks. *International Journal of Child-Computer Interaction*, 18, 8–18. <https://doi.org/10.1016/j.ijcci.2018.03.004>
- Koushik, Guinness, D., & Kane, S. (2019). StoryBlocks: A Tangible Programming Game To Create Accessible Audio Stories. *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 1–12. <https://doi.org/10.1145/3290605.3300722>
- Horn, & Jacob, R. (2007). Tangible programming in the classroom with tern. *CHI '07 Extended Abstracts on Human Factors in Computing Systems*, 1965–1970. <https://doi.org/10.1145/1240866.1240933>

- Hu, Zekelman, A., Horn, M., & Judd, F. (2015). Strawbies: explorations in tangible programming. *Proceedings of the 14th International Conference on Interaction Design and Children*, 410–413. <https://doi.org/10.1145/2771839.2771866>
- KOCA, S.; ÇAKIR, R. Effect of educational robotic applications on students' cognitive outcomes. *Behaviour & Information Technology*, [s. l.], v. 41, n. 15, p. 3329–3345, 2022. DOI 10.1080/0144929X.2021.1984580. Disponível em: <https://search-ebscohost-com.proxy.library.vcu.edu/login.aspx?direct=true&AuthType=ip,url,cookie,uid&db=a9h&AN=160904499&site=ehost-live&scope=site>. Acesso em: 18 jan. 2023.
- Koushik, V., Guinness, D., and Kane, S. K. (2019). StoryBlocks: A Tangible Programming Game to Create Accessible Audio Stories. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (1-12)*. Association for Computing Machinery.
- Lakatos, S. and Marks, L.E. (1999). Haptic Form Perception: Relative Saliency of Local and Global Features. *Perception and Psychophysics*, 61 (5), 895-908.
- Ladner, R. E. (2015). Design for user empowerment. *Interactions*, 22(2), 24–29. <https://doi.org/10.1145/2723869>
- Lederman, S.J. and Klatzky, R.L. (1993). Extracting object Properties Through Haptic Exploration. *Acta Psychologica*, 84, 29-40.
- Lederman, S.J. and Klatzky, R.L. (1997). Relative availability of surface and object properties during early haptic processing. *Journal of Experimental Psychology. Human Perception and Performance*, 23(6), 1680–1707. <https://doi.org/10.1037//0096-1523.23.6.1680>
- Lederman, S.J. and Klatzky, R.L. (2009). Haptic perception: a tutorial. *Attention, Perception and Psychophysics*, 71(7), 1439-1459.
- Loomis, J. M. (1981). On the tangibility of letters and braille. *Perception & Psychophysics*, 29(1), 37–46. <https://doi.org/10.3758/BF03198838>
- Ludi, & Jordan, S. (2015). A JBrick: Accessible Robotics Programming for Visually Impaired Users. In *Universal Access in Human-Computer Interaction. Access to Learning, Health and Well-Being* (pp. 157–168). Springer International Publishing. https://doi.org/10.1007/978-3-319-20684-4_16
- Maloney, J., Resnick, M., Rusk, N., Silverman, B. and Eastmond, E. (2010). The Scratch Programming Language and Environment. *ACM Transactions on Computing Education*, 10(4), 1–15. <https://doi.org/10.1145/1868358.1868363>
- Milne, & Ladner, R. (2018). Blocks4All: Overcoming Accessibility Barriers to Blocks

- Programming for Children with Visual Impairments. *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 1–10.
<https://doi.org/10.1145/3173574.3173643>
- Morrison, C., Villar, N., Thieme, A., Salandin, O., Cletheroe, D., Saul, G., Edge, D., Grayson, M. and Zhang, H. (2020). Torino: A Tangible Programming Language Inclusive of Children with Visual Disabilities. *Human-Computer Interaction*, 35 (3), 191-239.
- Morrison, C., Villar, N., Hadwen-Bennett, A., Regan, T., Cletheroe, D., Thieme, A. and Sentence, S. (2021). Physical Programming for Blind and Low Vision Children at Scale. *Human-Computer Interaction*, 36 (5-6), 535-569.
- Papert, S. *Mindstorms: Children, Computers, and Powerful Ideas*; Basic Books: New York, NY, USA, 1993; ISBN0465046746.
- Piaget, Roberts, G.-A., & Piaget, J. (1973). *To understand is to invent : the future of education* (Roberts, Trans.). Grossman Publishers.
- Pires, Rocha, F., de Barros Neto, A., Simão, H., Nicolau, H., & Guerreiro, T. (2020). Exploring accessible programming with educators and visually impaired children. *Proceedings of the Interaction Design and Children Conference*, 148–160.
<https://doi.org/10.1145/3392063.3394437>
- Resnick, M., Maloney, J., Monroy Hernandez, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Sliver, J., Silverman, B. and Kafai, Y. (2009). Scratch Programming for All. *Communications of the ACM*, 52 (11), 60-67.
- Resnick, M., & Rosenbaum, E. (2013). Designing for Tinkerability. In Honey, M. & Kanter, D. (Eds.), *Design, Make, Play: Growing the Next Generation of STEM Innovators* (pp. 163-181). Routledge.
- Rong, Chan, N. F., Chen, T., & Zhu, K. (2020). Toward Inclusive Learning: Designing and Evaluating Tangible Programming Blocks for Visually Impaired Students. In *Human-Computer Interaction. Human Values and Quality of Life* (pp. 326–338). Springer International Publishing. https://doi.org/10.1007/978-3-030-49065-2_23
- Theresa Beaubouef and John Mason. 2005. Why the high attrition rate for computer science students: some thoughts and observations. *SIGCSE Bull.* 37, 2 (June 2005), 103-106.
 DOI: <http://dx.doi.org/10.1145/1083431.1083474>
- Sabuncuoglu. (2020). Tangible Music Programming Blocks for Visually Impaired Children. *Proceedings of the Fourteenth International Conference on Tangible, Embedded, and Embodied Interaction*, 423–429. <https://doi.org/10.1145/3374920.3374939>
- Sáez-López, Román-González, M., & Vázquez-Cano, E. (2016). Visual programming languages

- integrated across the curriculum in elementary school: A two year case study using “Scratch” in five schools. *Computers and Education*, 97, 129–141.
<https://doi.org/10.1016/j.compedu.2016.03.003>
- Schreier, M. (2012). *Qualitative Content Analysis*. Sage Publications Ltd, Thousand Oaks, California.
- Saldana, J. (2021). *The Coding Manual for Qualitative Researchers*. Sage Publications Ltd. Thousand Oaks, California.
- Stefik, S. Siebert, M. Stefik, and K. Slattery, An empirical comparison of the accuracy rates of novices using the quorum, perl, and random programming languages. *In Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*, pp. 3-8, ACM, October 2011.
- Stephanie Ludi. 2015. Position paper: Towards making block-based programming accessible to blind users. *IEEE Blocks and Beyond Workshop*, 67-69
- Thieme, A., Morrison, C., Villar, N., Grayson, M., & Lindley, S. (2017). Enabling Collaboration in Learning Computer Programming Inclusive of Children with Vision Impairments. *Proceedings of the 2017 Conference on Designing Interactive Systems*.
<https://doi.org/10.1145/3064663.3064689>
- Van Boven RW, Johnson KO. The limit of tactile spatial resolution in humans: grating orientation discrimination at the lip, tongue, and finger. *Neurology*. 1994 Dec;44(12):2361-6. doi: 10.1212/wnl.44.12.2361. PMID: 7991127.
- Yannier, N., Hudson, S.E., Koedinger, K.R., Hirsh-Pasek, K., Michnick Golinkoff, R., Munakata, Y., Doebel, S., Schwartz, D.L., Deslauriers, L., McCarty, L., Callaghan, K., Theobald, E.J., Freeman, S., Cooper, K.M., Brownell, S.E. (2021). Active Learning: “Hands-on” Meets “Minds-on”. *Science*, 374 (6563), 26-30.
- Zuckerman, O., Grotzer, T., & Leahy, K. (2006). Flow blocks as a conceptual bridge between understanding the structure and behavior of a complex causal system. *7th International Conference of the Learning Sciences*, 880-886, Bloomington, IN, USA.

Appendix A

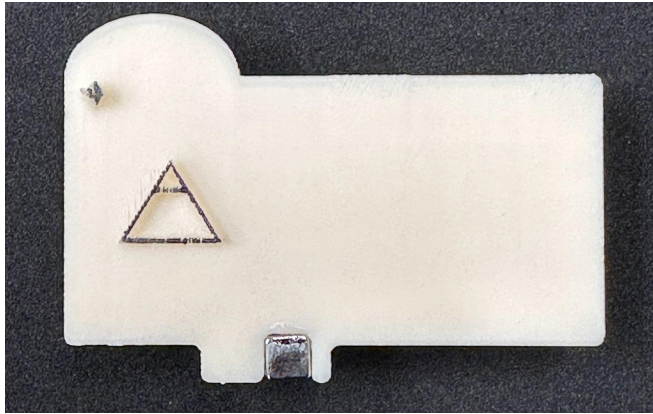


Figure A1. Event block for Enter Key

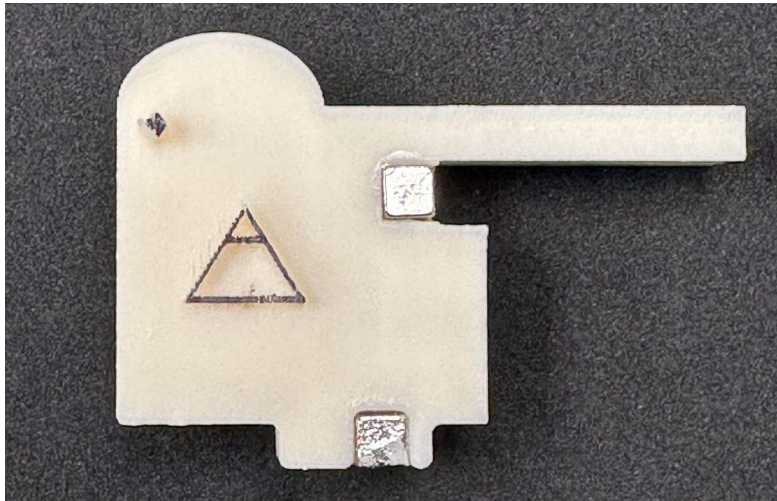


Figure A2. Event block for all keys other than Enter Key

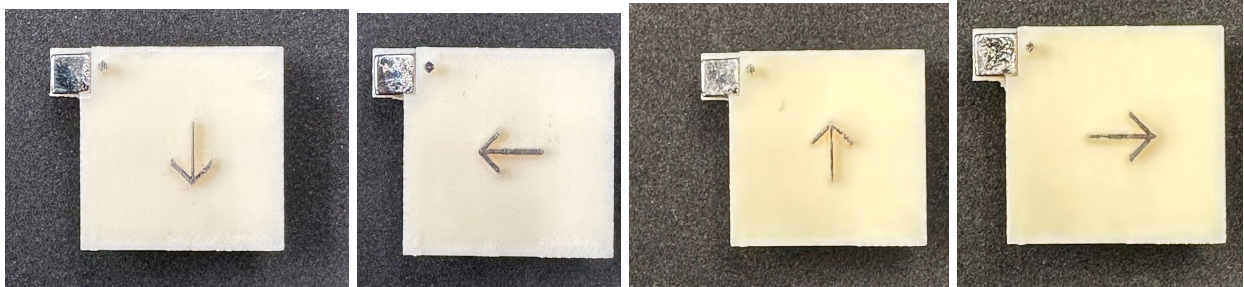


Figure A3. Arrow keys

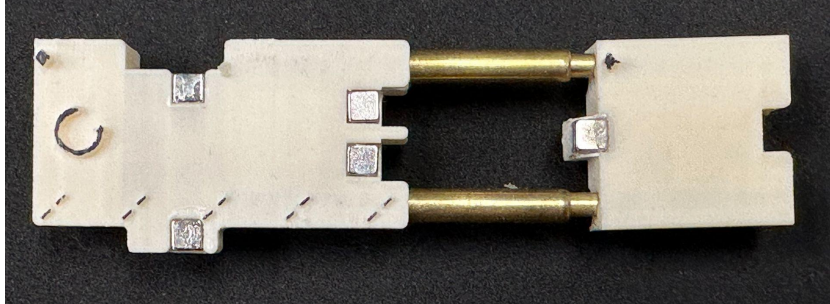


Figure A4. Counterclockwise turn

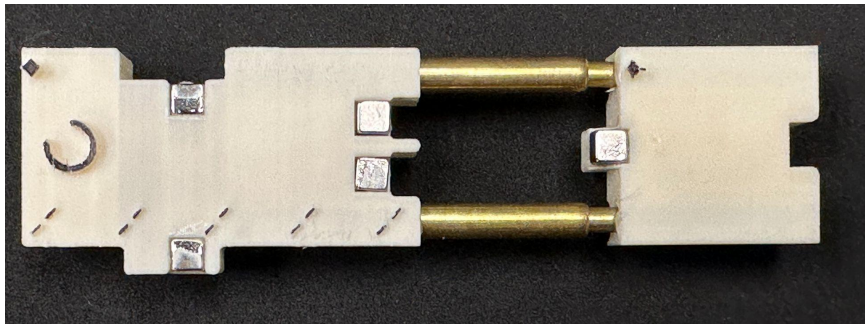


Figure A5. Clockwise turn

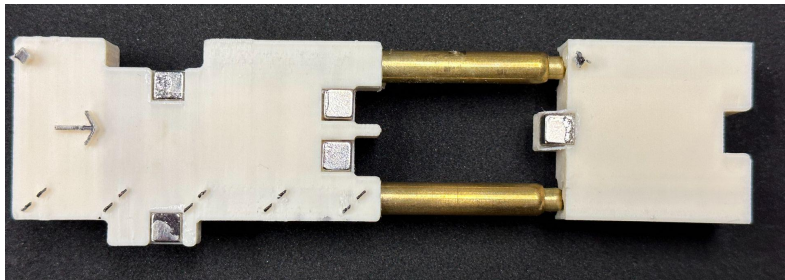


Figure A6. Step

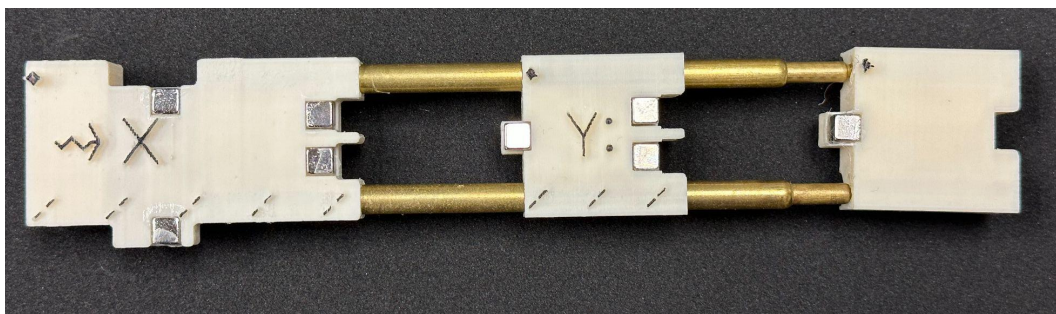


Figure A7. Goto

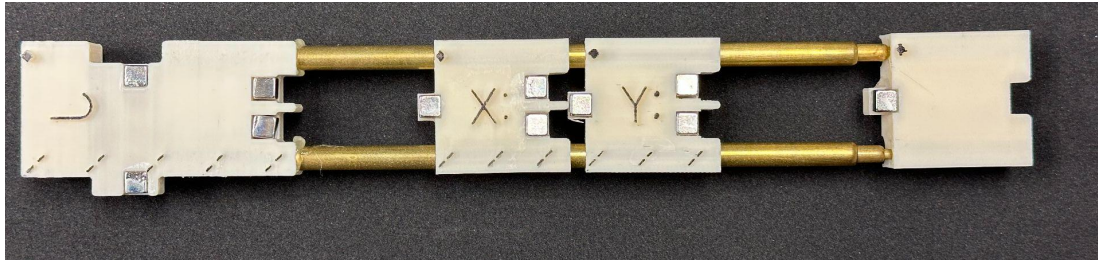


Figure A8. Glide

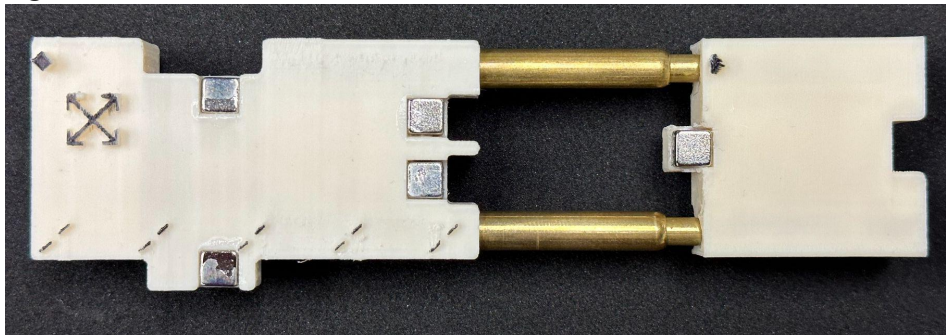


Figure A9. Point

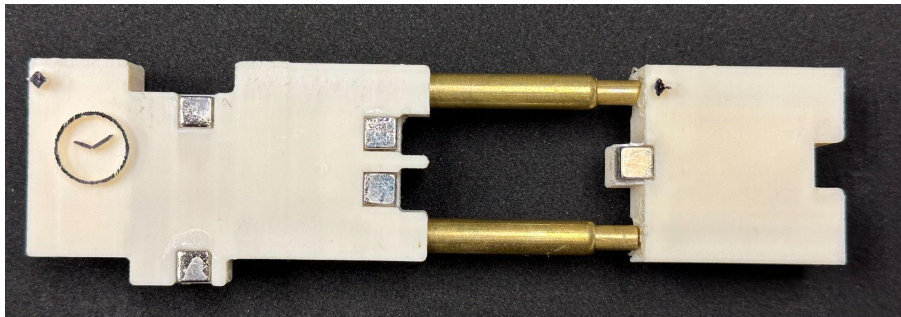


Figure A10. Wait

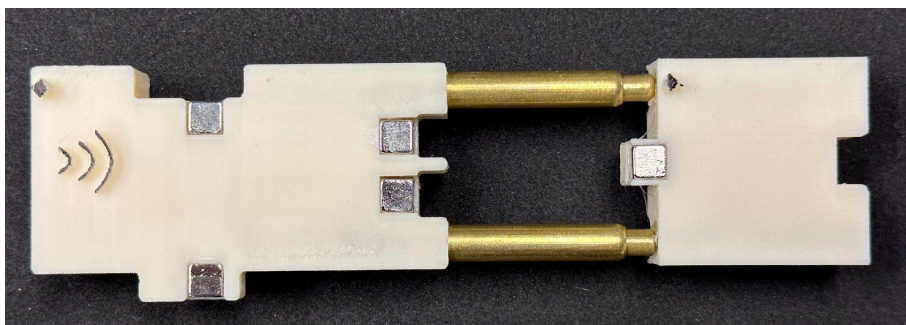


Figure A11. Beep

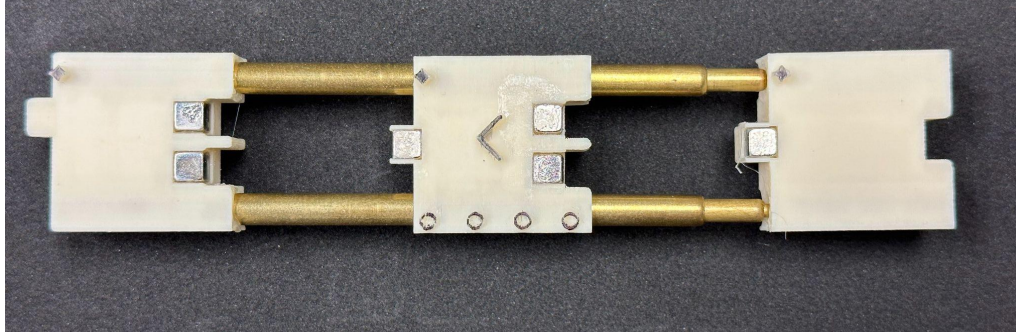


Figure A12. Less Than

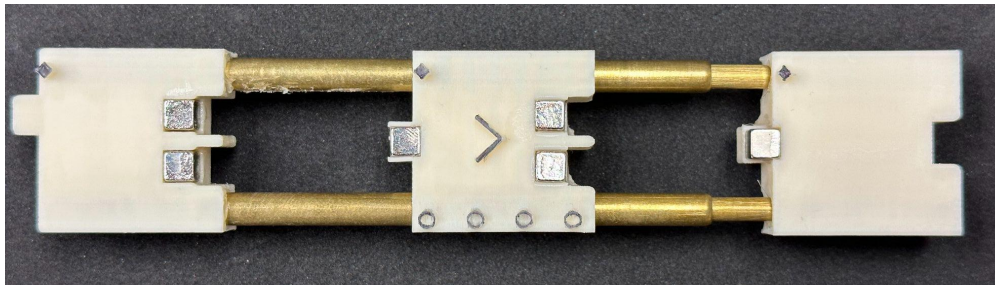


Figure A13. Greater Than

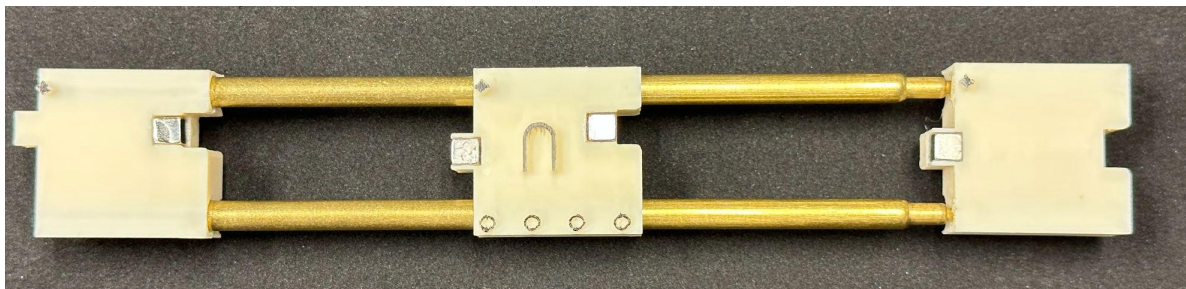


Figure A14. AND

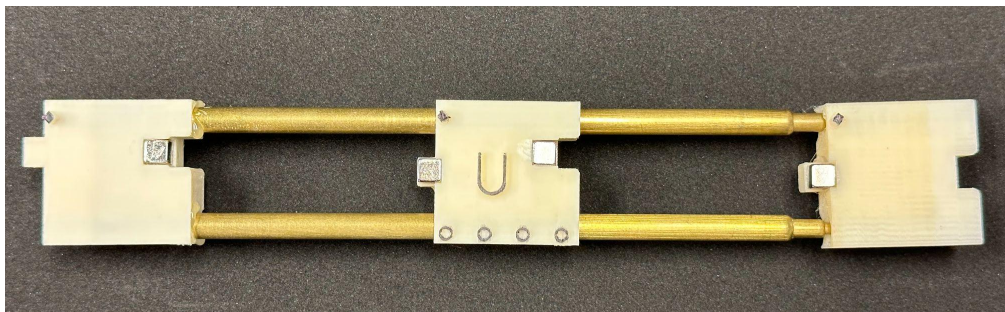


Figure A15. OR

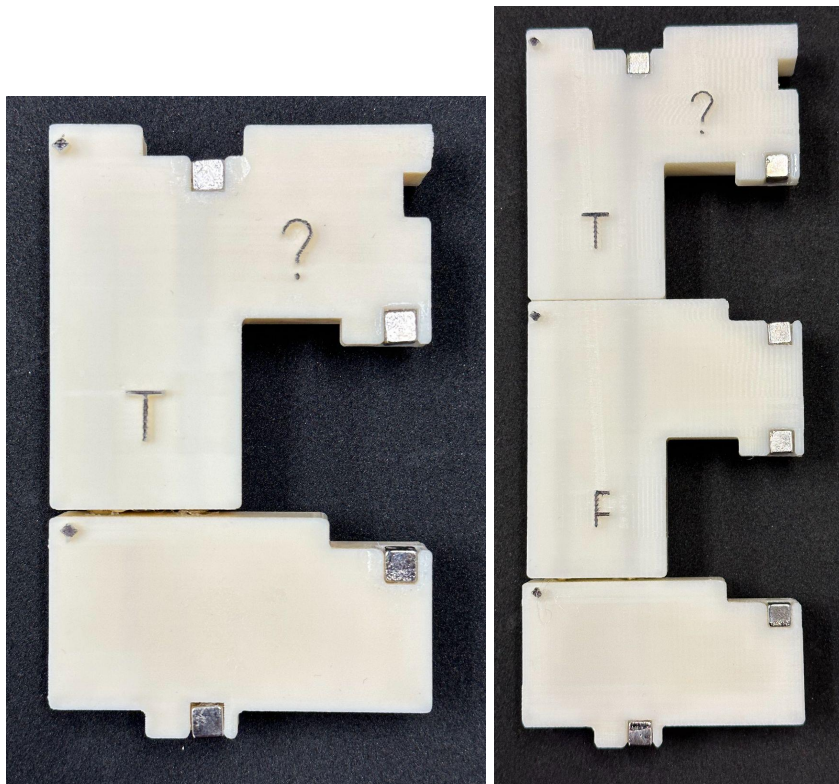


Figure A16. If-then (left) and If-then-else (right)

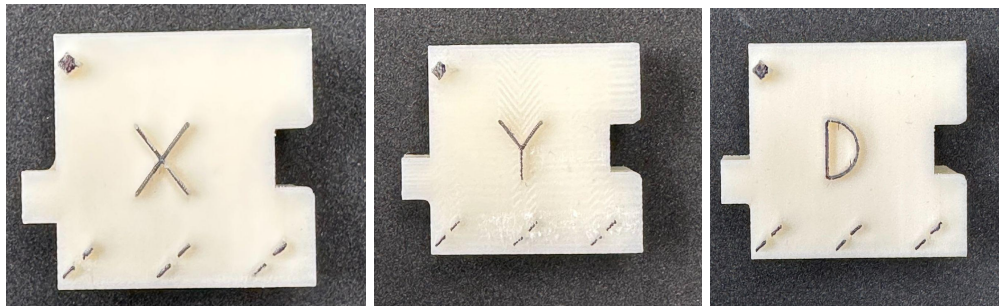


Figure A 17. Variable X (left), Y (middle), D (right)

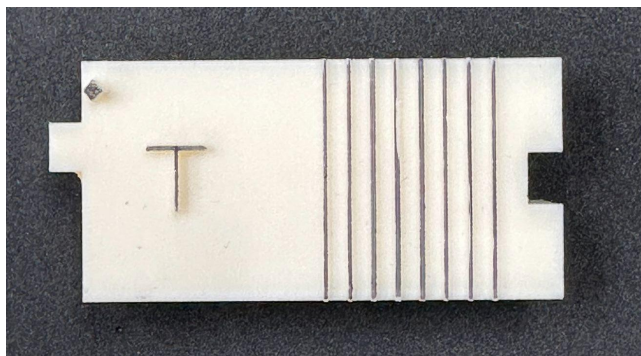


Figure A18. Boolean-texture rough



Figure A19. Boolean-texture smooth

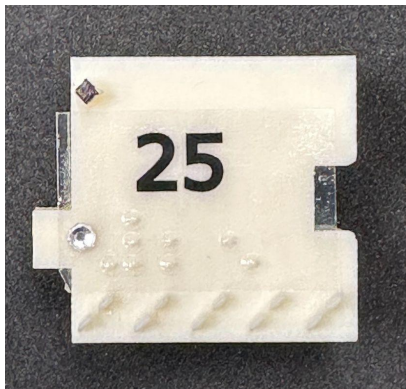


Figure A20. Example of a number block used during test

Appendix B

```
import pandas as pd
import os
from itertools import permutations
from pkg_resources import parse_requirements

file_path = os.path.abspath("s2d2csv.csv")
print(file_path)

df = pd.read_csv(file_path)

# display the first 5 rows of the data frame
df2 = df.apply(pd.to_numeric)

df3 = df2.to_numpy()
a = 0
b = a+1
c = a+2
d = a+3
occur=0
items = [1, 2, 3, 4, 5,6,7,8,9,10]
permu= list(permutations(items, 2))
print(permu[0], type(permu))

# comb2
results = []
for comb in permu:
    val1, val2 = comb
    occur=0
    a=0
    b=a+1
    for x in df3:
        try:
            if (df3[a] == val1 and df3[b]==val2):
                # print(val1,val2)
                # print("yes" , a , df3[a],b,df3[b])
                a=a+1
                b=b+1
                occur=occur+1
                # print(occur)
            else:
                a=a+1
                b=b+1
        except:
            pass
```

```
print(val1,val2,occur)
results.append((val1,val2,occur))
dataResult = pd.DataFrame(results, columns=["val1", "val2","occur"])
dataResult.to_excel("s2d2C2.xlsx", index=False)
```