UNDERSTANDING AND ADDRESSING ACCESSIBILITY BARRIERS FACED BY PEOPLE WITH VISUAL IMPAIRMENTS ON BLOCK-BASED PROGRAMMING ENVIRONMENTS

Aboubakar Mountapmbeme

Dissertation Prepared for the Degree of

DOCTOR OF PHILOSOPHY

UNIVERSITY OF NORTH TEXAS

December 2022

Mountapmbeme, Aboubakar. *Understanding and Addressing Accessibility Barriers Faced by People with Visual Impairments on Block-Based Programming Environments*. Doctor of Philosophy (Computer Science and Engineering), December 2022, 204 pp., 18 tables, 22 figures, 6 appendices, 126 numbered references.

There is an increased use of block-based programming environments in K-12 education and computing outreach activities to introduce novices to programming and computational thinking skills. However, despite their appealing design that allows students to focus on concepts rather than syntax, block-based programming by design is inaccessible to people with visual impairments and people who cannot use the mouse. In addition to this inaccessibility, little is known about the instructional experiences of students with visual impairments on current block-based programming environments. This dissertation addresses this gap by (1) investigating the challenges that students with visual impairments face on current block-based programming environments and (2) exploring ways in which we can use the keyboard and the screen reader to create block-based code. Through formal survey and interview studies with teachers of students with visual impairments and students with visual impairments, we identify several challenges faced by students with visual impairments on block-based programming environments. Using the knowledge of these challenges and building on prior work, we explore how to leverage the keyboard and the screen reader to improve the accessibility of block-based programming environments through a prototype of an accessible block-based programming library. In this dissertation, our empirical evaluations demonstrate that people with visual impairments can effectively and efficiently create, edit, and navigate block-based code using the keyboard and screen reader alone. Addressing the inaccessibility of block-based programming environments would allow students with

visual impairments to participate in programming and computing activities where these systems are used, thus fostering inclusion and promoting diversity.

ACKNOWLEDGEMENT

All thanks and praises to the Almighty God for keeping me alive and in good health throughout my Ph.D. program and for all His countless blessings in my life.

First and foremost, I express my gratitude to Dr. Stephanie Ludi, who accepted to be my Ph.D. advisor and has been the best advisor ever. I'm eternally grateful to Dr. Ludi for her continuous support, advice, and mentorship throughout my Ph.D. program.

I also thank all the members of my dissertation committee, Dr. Bryant, Dr. Do and Dr. Tarau. Thank you so much for your valuable feedback.

I am particularly grateful to my late elder brother and father figure, Assanegou Soule. My brother was the father I never had. He did everything for me to be educated and succeed in life.

I am also grateful to my mom for her endless prayers. I give special thanks to my nephew, Issofa Donmy and his wife, Mariama Donmy, for their continuous support and for always being there for me throughout my program. I also thank all my brothers and sisters for their prayers and encouragements.

I want to also give special thanks to my wife, Aisha, for her countless prayers and support.

Lastly, I thank all my friends in the RISE lab and in the CSE department for their collaborations and fun discussions.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENT iii

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER 1. INTRODUCTION 1
- 1.1 Introduction 1
- 1.2 Research Questions 4
- 1.3 Contributions 5
- 1.4 Structure of this Dissertation 6

CHAPTER 2. BACKGROUND 8
- 2.1 Screen Reader 8
- 2.2 Magnification Software 8
- 2.3 Refreshable Braille Display 9
- 2.4 Keyboard 10

CHAPTER 3. RELATED WORK 11
- 3.1 Introduction 12
- 3.2 Goals and Research Questions 13
- 3.3 Search and Selection Process 15
- 3.4 Analysis 19
- 3.5 Research Question 1 19
- 3.6 Research Question 2 27
- 3.7 Discussion and Opportunities for Future Work 49
- 3.8 Conclusion 52

CHAPTER 4. CHALLENGES FACED BY STUDENTS WITH VISUAL IMPAIRMENTS ON BLOCK-BASED PROGRAMMING ENVIRONMENTS 54
- 4.1 Survey Study with Teachers of Students with Visual Impairments 55
- 4.2 Interview Study with Teachers of Students with Visual Impairments 72
- 4.3 Survey Study with Students with Visual Impairments 97

CHAPTER 5. NAVIGATING BLOCK-BASED PROGRAMS USING ACCESSIBLE BLOCKLY 105
- 5.1 Introduction 105
- 5.2 Related Work 107

5.3 System Design and Implementation 109
5.4 Experiment Design 116
5.5 Evaluation Results 123
5.6 Qualitative Results 128
5.7 Discussion 133
5.8 Limitations 136
5.9 Future Work 136
5.10 Conclusion 137
5.11 Acknowledgment 137

CHAPTER 6. CREATING AND EDITING BLOCK-BASED PROGRAMS USING ACCESSIBLE BLOCKLY 138
6.1 Introduction 138
6.2 Enhanced Accessible Blockly 139
6.3 Experiment Design 142
6.4 Evaluation Results 151
6.5 Qualitative Results 156
6.6 Limitations 165
6.7 Future Work 165
6.8 Conclusion 166

CHAPTER 7. CONCLUSION, MERITS, IMPACTS AND FUTURE WORKS 167
7.1 Merits 168
7.2 Impacts 169
7.3 Future Work 169

APPENDIX A. ACCESSIBILITY CHALLENGES REPORTED BY PAPER 171

APPENDIX B. SURVEY QUESTIONS FOR TEACHERS OF STUDENTS WITH VISUAL IMPAIRMENTS 173

APPENDIX C. INTERVIEW QUESTIONS WITH TEACHERS OF STUDENTS WITH VISUAL IMPAIRMENTS 177

APPENDIX D. SURVEY QUESTIONS FOR STUDENTS WITH VISUAL IMPAIRMENTS 180

APPENDIX E. EVALUATING ACCESSIBLE BLOCKLY FOR CODE NAVIGATION AND COMPREHENSION 185

APPENDIX F. EVALUATING ACCESSIBLE BLOCKLY FOR BLOCK-BASED CODE CREATION AND EDITING 190

REFERENCES 194

LIST OF TABLES

Page

Table 3.1: Search String used to drive the database search 16

Table 3.2: Codes/labels and the category to which they belong 19

Table 3.3: Plugin tools to enhance accessibility in IDEs 28

Table 3.4: Novel accessible programming languages 39

Table 3.5: Accessible programming tool kits 42

Table 4.1: Type of schools in which TVI participants worked 57

Table 4.2: Overview of interview study participants, grades taught and years of experience 75

Table 4.3: Programming languages and environments used by study participants 78

Table 4.4: Summary of student participants 99

Table 5.1: Keyboard navigation shortcuts for Accessible Blockly 113

Table 5.2: Summary of participants' background information 118

Table 5.3: Task completion time 123

Table 5.4: Task score 125

Table 5.5: Number of participants who got full points per task 126

Table 6.1: Updated keyboard shortcuts in Accessible Blockly 139

Table 6.2: Participants details including experience with programming and assistive technologies 143

Table 6.3: Task completion time by participant 151

Table 6.4: Task score by participants 153

LIST OF FIGURES

Page

Figure 1.1: A sample block-based programming environment with the logic category of blocks expanded on the toolbox (Google Blockly). 1

Figure 2.1: A magnified portion of an editor using magnification software[11]. 9

Figure 2.2: A refreshable braille display [13]. 9

Figure 3.1: Scope of literature review. 14

Figure 3.2: Search process. 15

Figure 4.1: Swift Playground coding environment [107] 88

Figure 5.1: Blockly's editor showing the toolbox and a sample program on the workspace 106

Figure 5.2: Example blocks and their connection points 109

Figure 5.3: Components of Accessible Blockly 110

Figure 5.4: The virtual cursors in Accessible Blockly 113

Figure 5.5: Task 1 task completion time by participants for both navigation schemes 124

Figure 5.6: Task 2a task completion time by participants for both navigation schemes 124

Figure 5.7: Average score across participants on the semantically anchored questions 127

Figure 6.1: A sample block-based program to illustrate navigation 140

Figure 6.2: Training task at start and end of training 144

Figure 6.3: Accepted program for Task 1 145

Figure 6.4: Task 2 starting program and the expected answer after completion 145

Figure 6.5: Starting program for Task 3 145

Figure 6.6: Acceptable answers for Task 3 146

Figure 6.7: Examples of partially correct programs for Task 1 149

Figure 6.8: Examples of partially correct programs for Task 3 149

Figure 6.9: Average of each semantically anchored scaled data on participants experience 156

CHAPTER 1

INTRODUCTION

1.1 Introduction

A block-based programming language (BBL) is a type of programming language where programming instructions and syntax are represented by predefined visual elements called blocks [1]. Users typically write programs by interlocking multiple compatible blocks together to create a syntactically correct program. A block-based programming environment (BBPE) is a platform on which users write programs using a BBL. Interaction takes place mainly through drag-and-drop operations using a mouse. Users drag desired blocks from a toolbox that contains predefined blocks and connect them on the workspace (Figure 1.1).

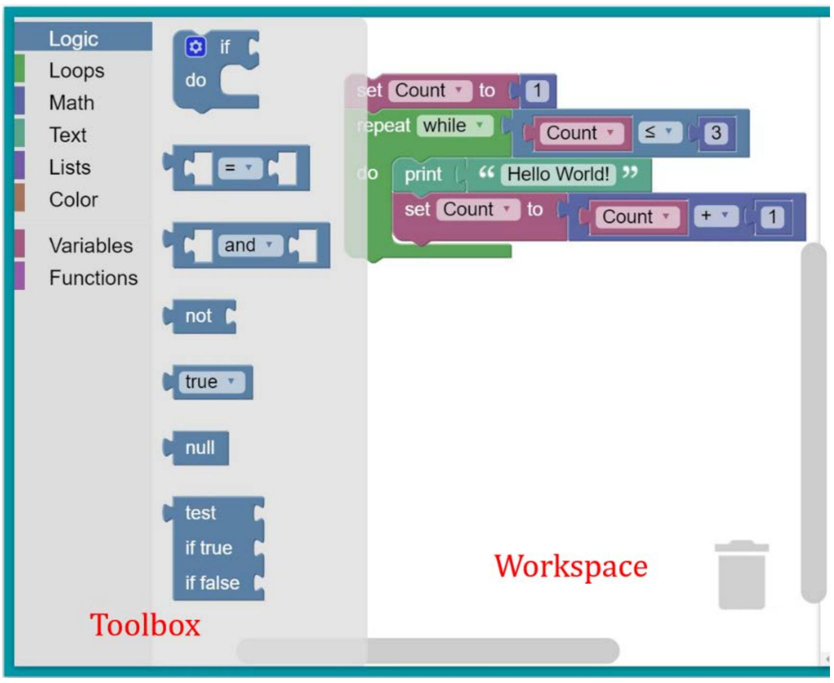


Figure 1.1: A sample block-based programming environment with the logic category of blocks expanded on the toolbox (Google Blockly).

BBLs rely on visual cues such as the shape and color of blocks to convey programming syntax and semantics. This visual and mouse-centric design allows users to focus on concepts and not have to deal with the heavy textual syntax characteristic of text-based languages. This has made BBLs appealing to learners and are increasingly being used in K-12 curricula to introduce novices to programming and computational thinking concepts [1][2][3]. Example mainstream BBPEs include Scratch by MIT[1], MakeCode by Microsoft[2], Snap![3] and AppInventor[4].

With the increased use of BBLs in K-12 curricula, there is a need to add accessibility in order to serve all students. The intrinsic visual and the mouse-centric nature of BBLs make them inaccessible to people who rely on assistive technologies and cannot use the mouse. Thus, excluding them from participating in programming and computational thinking activities where these platforms are used. This is particularly applicable to people who are blind or have low vision. People with vision loss or low vision, hereafter referred to as people with visual impairments, typically interact with computers using assistive technologies such as screen readers, magnification software, braille displays, and keyboards. A screen reader is a software that reads out loud content displayed on the screen to a user. Examples include JAWS[5], VoiceOver[6], and NVDA[7]. A Magnification software allows users to enlarge content or zoom in on the screen. A refreshable braille display is a device that converts on-screen text into braille. Users who rely on screen readers mainly use the keyboard as the primary input device to the computer. By default, BBPEs do not support interaction using these devices. Milne et al. [4] evaluated the most common BBPEs for compatibility with screen readers and

[1] https://scratch.mit.edu/

[2] https://makecode.microbit.org/

[3] https://snap.berkeley.edu/

[4] https://appinventor.mit.edu/

[5] https://www.freedomscientific.com/

[6] https://www.apple.com/accessibility/vision/

[7] https://www.nvaccess.org/

keyboards and found that none of them was accessible when using these assistive technologies.

The inaccessibility of programming tools and materials in many CS curricula is a significant barrier for people with visual impairments to pursue computing-related courses [5]. According to the American Foundation for the Blind[8] (AFB), there are approximately 568 202 US children under the age of 18 with vision difficulty [6]. This only includes those who are blind and those with extremely low vision who cannot use glasses. A number too large not to neglect, given the increased call for equal opportunity and access to education including computer science for everyone [7] [8]. The AFB also reports that there are approximately 63,657 U.S. students in educational settings with vision difficulty. However, it is hard to find the exact percentage of students with visual impairments in computer science or computing-related courses. But the general trend among researchers is that this percentage is low and calls for concern [8], [9]. This low participation in computing and programming is primarily attributed to the inaccessibility of materials in many CS curricula, including inaccessible programming tools and platforms [5]. This concern is even more significant with the increased use of BBLs in many CS curricula and outreach activities intended to attract learners to programming. This is because unlike text-based languages (TBLs) which have some limited accessibility with assistive technologies, BBLs, by default, are not accessible when using assistive technologies such as screen readers and keyboards. Therefore, the increased use of inaccessible BBLs in K-12 curricula and programming outreach activities might exclude more students with visual impairments and any students who cannot use a mouse in general.

[8] https://www.afb.org/

In this dissertation, we investigate and analyze the challenges that students with visual impairments face on current block-based programming environments. We do this through survey studies and interviews with teachers of students with visual impairments (TVIs) and students with visual impairments. We use the findings from these studies to design and improve the accessibility and usability of an accessibility library for block-based programming environments called Accessible Blockly. We also evaluate this library in formal empirical studies with students with visual impairments to demonstrate how the library supports and facilitates code navigation, code construction, code editing, and code comprehension. The accessibility library uses speech through the screen reader to provide feedback/output to a user interacting on the block-based programming environment. For user inputs, the keyboard is leveraged to mimic drag-and-drop-like operations characteristic of block-based programming environments.

The research efforts presented in this dissertation were motivated by the desire to contribute to promoting equity, inclusion, and increasing participation of people with visual impairments in programming, computational thinking activities, and computer science in general.

1.2 Research Questions

We formulated the following four primary research questions (PRQs) to support this dissertation and help guide the research activity.

PRQ1. What block-based programming environments are currently being used by students with visual impairments in educational settings?

PRQ2. What challenges do students with visual impairments face on current BBPEs? And how do they overcome these challenges?

PRQ3. What keyboard interaction strategies allow efficient engagement and drag-and-drop-like operations for code navigation, code construction, and code editing on BBPEs?

PRQ4. How can we leverage the screen reader to provide adequate feedback to the user to improve code navigation, code construction, and code editing on BBPEs?

PRQ1 and PRQ2 were designed to get a sense of the reality of the day-to-day instructional usage of block-based programming environments by students with visual impairments and TVIs. To answer these research questions, we conducted a survey study with 12 teachers of students with visual impairments and another survey study with seven students with visual impairments. To further understand and elicit the findings from the survey, we conducted a follow-up interview with 12 TVIs. The results from these studies show that none of the mainstream BBPEs is accessible to students with visual impairments and that the students use a hybrid environment. The TVIs and students identified several challenges related to code navigation, code editing, and code comprehension, as well as challenges associated with accessing supporting materials. The results also support the claim in this dissertation that there are currently no accessible block-based programming environments being used by TVIs to teach students with visual impairments despite the proliferation of these systems in K-12 education and computing outreach activities.

PRQ3 and PRQ4 were designed to provide empirical data to support the design and improvement of the accessibility library. An initial prototype of the library was evaluated through formal empirical studies with participants with visual impairments. Feedback from these studies was used to improve the library. These studies show that people with visual impairments can efficiently create, edit, navigate, and comprehend block-based code using a screen reader and keyboard alone.

1.3 Contributions

The contributions from this dissertation include:

1. Empirical evidence of the challenges students with visual impairments face on block-based programming and hybrid environments and how the students overcome these challenges.

2. Empirical evidence of how TVIs compensate for the lack of accessible block-based programming environments.

3. Accessible Blockly, an improved version of an accessible block-based programming library for building accessible block-based programming environments with a keyboard and a screen reader alone, as well as data that provides justification.

4. A prototype of the application of the accessibility library and the study results using Blockly as a model for developers that is also usable by persons with visual impairments or their sighted peers.

5. Best practices for incorporating keyboard and speech interactions on block-based programming environments.

The outcome of this research work will have several advantages. (1) Accessibility will be enabled on major BBPEs, thus bringing about equity and promoting inclusion. (2) Novices with visual impairments will be able to participate in programming and computational thinking activities using the same systems and platforms that are found appealing to their sighted peers. (3) Working on the same platform will increase collaboration among learners with visual impairments and their sighted peers. (4) Other people who cannot use the mouse, such as people with situational impairments, shall also benefit from this library.

1.4 Structure of this Dissertation

The remainder of this dissertation includes six chapters. Each chapter except Chapter 7 is based on scholarly articles we published in various academic conferences and journals.

Chapter 2 and Chapter 3 present the background and related work. Chapter 2 gives an overview of the different assistive technologies used by people with visual impairments to interact with computers. Chapter 3 then presents a systematic literature review of related work in making programming and programming environments accessible to people with visual impairments, including block-based and text-based programming. In addition to summarizing the related work, we also identify gaps and opportunities for future work within the current academic discourse.

Chapter 4 addresses PRQ1 and PRQ2 based on three qualitative studies conducted with teachers of students with visual impairments and students with visual impairments. The chapter provides details about the survey and interview studies and how these studies were conducted. The chapter also discusses the approach used to analyze the data gathered from these studies. The chapter then presents the findings and suggestions for improving the accessibility of block-based programming languages for people with visual impairments.

Chapter 5 introduces Accessible Blockly and the initial evaluation of it that addresses parts of PRQ3 and PRQ4. Accessible Blockly was evaluated with 12 users with visual impairments on block-based code navigation. The chapter presents the findings from this study and suggestions to enhance Accessible Blockly.

Chapter 6 discusses the enhancements made to Accessible Blockly after the initial evaluation study. The chapter also discusses an empirical investigation that evaluates Accessible Blockly for block-based code creation and editing with 11 blind programmers.

Lastly, Chapter 7 concludes the dissertation and provides directions for the next steps toward improving the accessibility of block-based programming environments for people with visual impairments.

CHAPTER 2

BACKGROUND

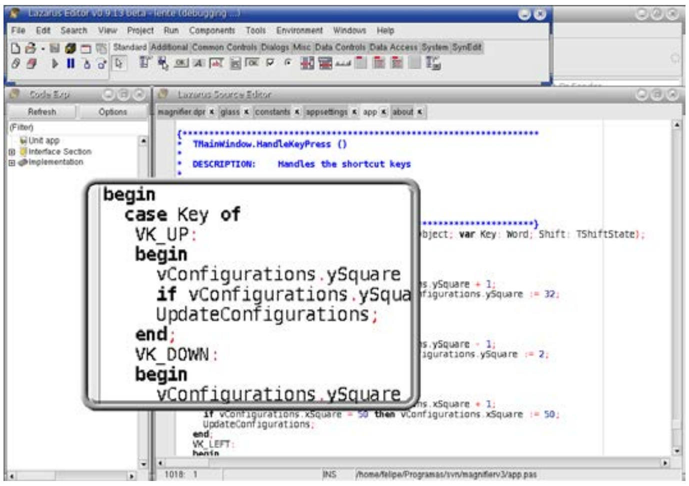
Before discussing the related work, it is worth briefly describing how people with visual impairments interact with computers. People with visual impairments rely on assistive technologies that offer alternative input/output channels when interacting with computers. The reminder of this chapter provides an overview of some of the assistive technologies used by blind people and people with low vision when interaction with computing devices. Knowing what these technologies are and how they operate is essential to understanding the rest of this dissertation.

2.1 Screen Reader

A screen reader is software that reads out load content on a screen. It serves as an alternative output to the visual channel. Screen readers are very popular nowadays and most mainstream operating systems ship with a screen reader. For example, VoiceOver for Apple devices and Narrator that ships with Windows 10. NVDA is a popular open-source screen reader that works with Windows. By default, applications are not screen reader accessible. It is for the application developers to follow and implement established accessibility guidelines for their application to be accessible. An example of such guideline is the Web Content Accessibility Guidelines (WCAG) that sets guidelines for making web applications accessible through assistive technologies including screen readers and the other assistive technologies discussed next. Most people who are blind rely on screen readers to access content on computer and mobile applications.

2.2 Magnification Software

According to the American Foundation for the Blind (AFB), a magnification software is an application that magnifies or enlarges portions of a computer screen to its user [10]. It enlarges both text and graphics on the screen. The location of the cursor determines the portion to be magnified. Figure 2.1 shows a code editor with a section of it magnified[11]. Magnification software are used by people with low vision. It is also common to find people use a combination of a screen reader and a magnification software [10].



```
{*************************************************************
 *  TMainWindow.HandleKeyPress ()
 *
 *  DESCRIPTION:    Handles the shortcut keys
 *
```

```
begin
  case Key of
   VK_UP:
   begin
     vConfigurations.ySquare
     if vConfigurations.ySqua
     UpdateConfigurations;
   end;
   VK_DOWN:
   begin
     vConfigurations.ySquare
```

```
      if vConfigurations.xSquare = 50 then vConfigurations.xSquare := 50;
      UpdateConfigurations;
    end;
    VK_LEFT:
```

```
1018: 1        INS    /home/felipe/Programas/svn/magnifierv3/app.pas
```

Figure 2.1: A magnified portion of an editor using magnification software[11].

2.3 Refreshable Braille Display

A braille display is a tactile device that connects to a computer and converts on-screen text into braille that can be felt on the device (see Figure 2.2).

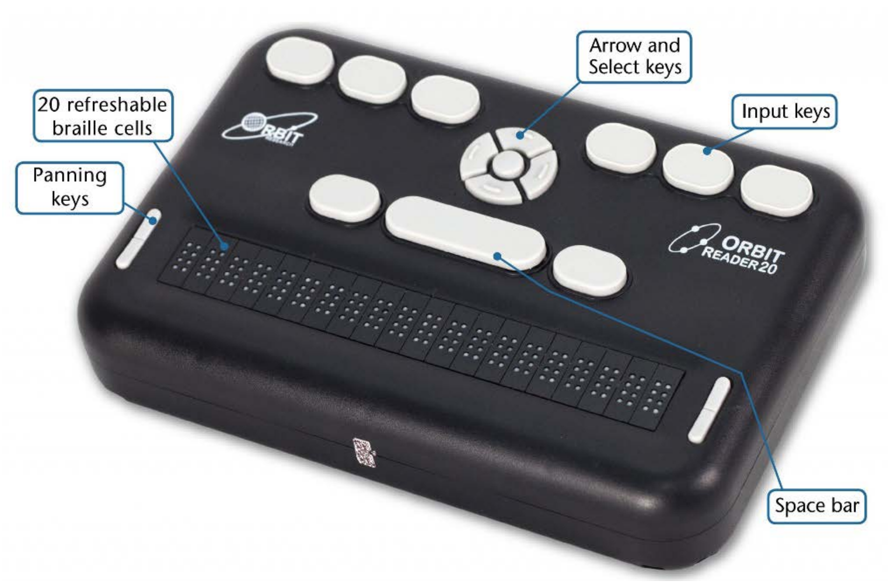


Figure 2.2: A refreshable braille display [13].

It can be described as a tactile monitor. The braille display electronically raises or lowers pins on the device to form different combination of braille text. The display is refreshable, meaning the braille displayed changes as the location of the cursor moves around the screen as desired by the user[12]. Unlike a screen reader that ships with most mainstream operating system or that are available for free download, braille displays are sold and can be expensive. According to the AFB, the price of a braille display ranges from $3500 to $15000 [12].

2.4 Keyboard

The standard keyboard is the main input device used by people with visual impairments to provide input to the computer such as controlling the cursor and typing text.

CHAPTER 3

RELATED WORK*

In this chapter, we present a systematic review of the literature in the accessibility of programming languages and programming environments for people with visual impairments. We conducted this systematic review to identify the accomplishments made so far, the gaps and opportunities for research to serve as basis for my research work. This chapter from a purely theoretical perspective partly answers PRQ1: What block-based programming environments are currently being used by students with visual impairments in educational settings? and PRQ2: What challenges do students with visual impairments face on current BBPEs? And how do they overcome these challenges?

This systematic review focuses on the accessibility of text-based programming and block-based programming alike. This is because as would be seen throughout this chapter, research on the accessibility of block-based programming environments is still at its infancy and can be traced back to 2015 [14]. Understanding the what, the how and the why of accessibility in text-based programming can also serve as basis to address the accessibility of block-based programming environments. We reviewed all accessibility research efforts in making programming and programming environments accessible to people with visual impairments that have occurred over the past two decades. This includes work that seeks to investigate and understand accessibility barriers as well as research efforts that propose solutions to overcome these barriers and make programming friendlier to people with visual impairments. The work in this chapter was done in collaboration with Obianuju Okafor and Stephanie Ludi and is based on work

* Parts of this chapter have been previously published, either in part or in full, from Aboubakar Mountapmbeme, Obianuju Okafor, and Stephanie Ludi. 2022. Addressing Accessibility Barriers in Programming for People with Visual Impairments: A Literature Review. ACM Trans. Access. Comput. 15, 1, Article 7 (March 2022), 26 pages. https://doi.org/10.1145/3507469, with permission from ACM.

published in the ACM journal ACM Transactions on Accessible Computing (TACCESS) [15].

3.1 Introduction

While computer science and programming offers numerous opportunities to people who are blind or people with low vision such as the ability to work independently, current literature shows that programmers with visual impairments face many challenges in their day-to-day activities[16],[17]. These challenges stem from the nature of the tools they work with. These include Integrated Development Environments (IDEs) and programming languages that are mostly designed with sighted people in mind. These heavily visual tools and the intrinsic characteristics of most programming languages impose significant barriers to people with visual impairments working as programmers or software developers. Two percent of nearly 65 thousand developers who took the 2020 Stack Overflow survey [18] reported having a disability of some kind, including visual impairments and blindness, which is considered a low-incidence disability.

Similarly, introductory computer science or programming courses are increasingly being incorporated as part of the K-12 curricula around the world and in the United States in particular. For example, in the United States, initiatives such as CS10K [19], [20], and Code.org [21] are aimed at increasing computer science (CS) education participation nationally. In many of these CS curricula, programming is taught using integrated development environments (IDEs) like Visual Studio and Eclipse. In addition to these traditional programming environments, tools and technologies have been developed to increase the interest and attention of students and cater to novice programmers; they include block-based and hybrid programming environments such as Scratch [22], Blockly [23], and Pencilcode [24]. Unfortunately, most of these technologies have either been reported to be inaccessible to students with visual impairments or present significant barriers to novices with visual impairments learning how to code on these systems. This could discourage students with visual impairments from taking part in computer science and related courses. According to the American Foundation for the Blind (AFB), there are approximately 568, 202 U.S. children under the age of 18 with vision difficulty [6]. The AFB also reports that there are approximately 63,657 U.S. students in educational settings with vision difficulty. However, it is hard to find the exact percentage of students with visual impairments in computer science or computing-related courses. The general trend among researchers is that the percentage is low and calls for concern persist [8], [9].

For the past two decades, researchers have devoted efforts to identify and elaborate the barriers that affect the productivity of programmers with visual impairments and the ability of students with visual impairments to effectively learn programming. Researchers have also proposed solutions to address these barriers. For example, StructJumper [25] is a tool that helps blind programmers overcome Code Navigation and Code Comprehension challenges. Block4All[4] is an accessible block-based programming environment developed with students with visual impairments from the ground up.

3.2 Goals and Research Questions

3.2.1 Goal

The goal of this study is to examine current scholarship on the accessibility of programming languages and programming environments for people with visual impairments. This includes the following subgoals:

- Identify and highlight the barriers that people with visual impairments face while programming or learning how to program in recent academic literature.

- Identify and discuss the various solutions for addressing accessibility barriers as proposed by researchers and practitioners.
- Identify any gap in current discourse on the accessibility of programming to people with visual impairments.

Figure 3.1 shows the main research themes and the scope for this work. The themes include people with visual impairments, programming languages and programming environments. The intersection of these three themes represents the scope of our study. By people with visual impairments, we refer to people who are blind and people with low vision who require some sort of assistive technology to interact with computers. These include professional software developers and students learning how to code. With respect to programming languages and programming environments we discuss text-based programming (TBP), block-based programming (BBP) and tangible programming.

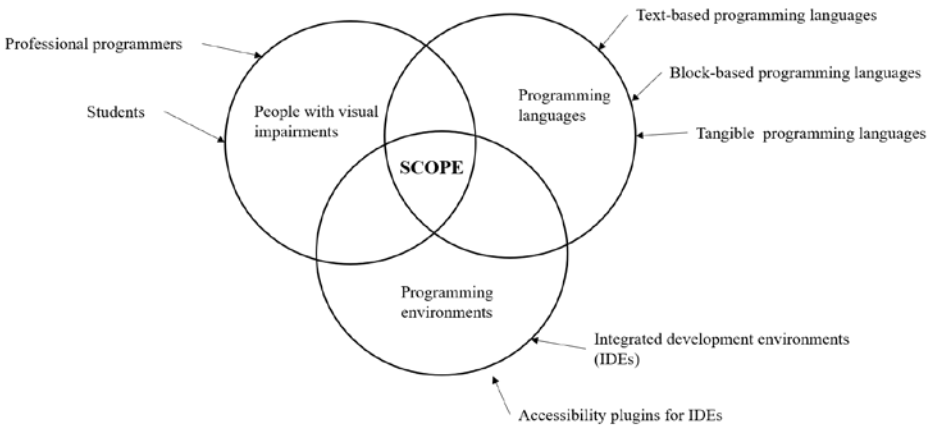

Professional programmers
Students
People with visual impairments
Text-based programming languages
Block-based programming languages
Programming languages
Tangible programming languages
SCOPE
Programming environments
Integrated development environments (IDEs)
Accessibility plugins for IDEs

Figure 3.1: Scope of literature review.

In [26], the authors give a higher level overview current research efforts in this domain by grouping and summarizing each work under the target programming language type including: text-based programming, block-based programming and tangible programming. Our work differs in that we perform an in-depth analysis of each research effort to identify commonalities, differences, and gaps between and within reported challenges and proposed solutions. We identified and grouped specific programming activities and tasks that have been reported to be challenging to people with visual impairments as well as the interventions that have been proposed to address these challenges. We believe a new review and analysis of the status quo is needed given the recent increase in research activities in this domain.

3.2.2 Research Questions

To achieve the goals stated under section 3.2.1, we formulated the following research questions. This allows us to keep focus and guides the search and selection process[27].

RQ1. What are the known accessibility barriers to programming for people with visual impairments identified by researchers and practitioners as represented in the literature?

RQ2. What are the existing and proposed solutions to these barriers?

3.3 Search and Selection Process

3.3.1 Search Process

Our search process comprises an initial database search, a manual search of identified publication venues and a final database search using an extended list of keywords (Figure 3.2). The initial database search confirmed that there are enough papers on the topic and that a new literature review was needed. The manual search of identified venues and the final database search ensured we covered all relevant papers published between 2000 and 2020.

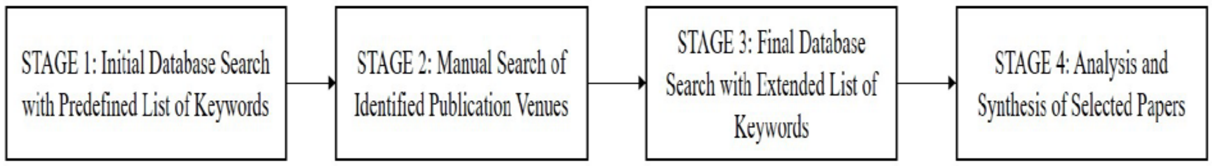

STAGE 1: Initial Database Search with Predefined List of Keywords
STAGE 2: Manual Search of Identified Publication Venues
STAGE 3: Final Database Search with Extended List of Keywords
STAGE 4: Analysis and Synthesis of Selected Papers

Figure 3.2: Search process.

Leveraging our experience in conducting research in this domain, we generated a list of keywords (Table 3.1) to drive the initial database search. The third author has over 10 years of research experience in this field and the other two authors have more than two years of experience. Using the list of keywords, we started the search on Google Scholar before moving on to the following databases: ACM Digital Library, IEEE, and Eric and Wiley. Two authors performed the search independently using the list of keywords. We stored data about each paper found in a common file to avoid storing duplicates. The data stored includes the title of the paper, list of authors, the venue of publication and the publication year. The papers were also stored in a common directory. Backward snowballing where the reference list of a paper is used to find related papers was also used during the search process. The initial search produced a total of 61 publications. Using the inclusion and exclusion criteria discussed in section 3.3.2, a total of 35 papers were retained from the initial database search.

Table 3.1: Search String used to drive the database search

Search Keywords
*Accessibility, **Computer Science Education**, **Computational thinking**, Accessible computing, **Computing Outreach**, **Challenges***
*Visual impairment, **blind students**, **blind programmer**s, **blind people**, **blind person***
*Programming languages, programming environments, tangible programming, block-based programming, **robotics**, **software tools***
*Sound, audio debugging, screen reader, auditory cues, **keyboard navigation***

To ensure that this literature review covers all relevant papers published between 2000 and 2020, we conducted a manual search of identified publication venues. Using the papers found in the initial database search, a list of unique publication venues was generated. For each venue, we manually scanned conference proceedings and journal issues published between 2000 and 2020. The three authors shared the list of venues to expedite the search process. During this process, we looked at the title and abstract of publications to decide if a publication should be included. Most publications in this domain can easily be identified from the title and abstract alone because they are usually explicit about programming and people with visual impairments. All three authors met regularly to discuss papers that were selected and to resolve any doubts. In cases where the title and abstract did not include sufficient information to rule out a paper for inclusion or exclusion, the full paper was read. The manual process produced an additional 48 new papers. Of the 48 papers 25 were excluded after reading the full content of the papers.

Because the manual search produced a sizeable number of papers, we suspected that our initial list of keywords was not sufficiently comprehensive to capture all the relevant papers. We examined the papers that resulted from the manual search process and came up with an additional list of keywords. We added these new keywords to the initial list to produce an extended list of keywords. Table 3.1 shows the final list of keywords with the newly added keywords in bold. Using this extended list of keywords, we performed one last round of database search and found an additional 23 papers. However, only 12 papers met the inclusion criteria.

A total of 70 papers were retained for analysis. These include 35 from the initial database search, 23 from the manual search and 12 from the final database search with an extended list of keywords.

3.3.2 Selection Process

The aim of this literature review is to identify and analyze relevant publications that talk about the accessibility of programming languages and programming environments for people with visual impairments. It focuses on professional and educational programming languages and environments. The next two subsections list the inclusion and exclusion criteria.

3.3.2.1 Inclusion Criteria

Articles published in English between January 2000 and December 2020 and met one of the following criteria were included for analysis.

- The main objective of the paper is to investigate programming accessibility barriers faced by professional programmers with visual impairments.
- The main objective of the paper is to investigate accessibility barriers to students with visual impairments learning how to program. This includes students in K-12, college and above.
- The paper discusses the design of an accessibility tool or plugin to help programmers with visual impairments overcome programming challenges.
- The paper discusses the design of an accessibility tool or plugin to help students with visual impairments overcome challenges when learning how to code.
- The paper discusses the design of an accessible programming language or programming environments for students with visual impairments or professional programmers with visual impairments.
- The paper discusses alternative programming paradigm or environments for people with visual impairment. For example, papers that talk about tangible programming for people with visual impairments.

3.3.2.2 Exclusion Criteria

The following types of papers were excluded.

- The paper meets one of the inclusion criteria but was published outside the target period.
- The paper talks about broadening participation in computer science to include more people with visual impairments without a focus on programing languages or programming environments.
- The paper talks about attracting people with visual impairments to pursue computing careers without a focus on programming languages or programming environments.
- The paper talks about accessibility issues with current CS curricula without a focus on programming languages or programming environments.
- The paper meets the inclusion criteria but is published as a newsletter.

3.4 Analysis

The 70 papers retained were read independently by two of the authors. Thematic analysis was used to examine each paper with the objective of identifying codes or themes. Two of the authors worked independently to identity themes from the papers. The authors met regularly with the third author to reconcile and resolved any conflicts. Once the codes were established, a last round of coding was performed to assigned codes to each paper. A further examination of the themes, via axial coding, revealed that they could be grouped under four different categories. These categories include *programming paradigm, programming tasks and challenges, assistive technology,* and *interaction mechanism*. Table 3.2 presents the list of codes and their categories.

Table 3.2: Codes/labels and the category to which they belong

Main Category	Codes
Programming Paradigm	Text-based, Blocked-based, Tangible, Audio
Programming Tasks and Challenges	Code Comprehension, Code Debugging, Code Editing, Code Navigation, Code Skimming,
Assistive Technology	Screen Reader, Braille Display, Keyboard, Magnification Software
Interaction Mechanism	Input, Output, Speech and Audio Cues

Based on the results of the coding process, the data was organized and sorted to identify similarities, differences, and relationships among the publications. Answers to the research questions were then organized and synthesized following the analysis. The results of our analysis are discussed in the remainder of this paper.

3.5 Research Question 1

What are the known accessibility barriers to programming for people with visual impairments identified by researchers and practitioners as represented in the literature?

Analysis of our corpus reveals that current literature identifies five main challenges or barriers faced by people with visual impairments in programming. These include code navigation, code comprehension, code editing, code debugging and code skimming challenges. Most of the data shows that these challenges are a consequence of the nature of the tools used by people with visual impairments to interact with computers. These barriers affect both text-based and block-based programming. Appendix A summarizes the challenges reported per paper.

3.5.1 Code Navigation and Code Comprehension Challenges

Accessibility of code navigation and code comprehension are studied together as observed from our analysis. Code comprehension challenges are a consequence of barriers to code navigation. Ineffective code navigation leads to improper or inaccurate code comprehension. Analysis of the literature reveals code navigation as the most reported accessibility challenge faced by programmers or students with visual impairments. The root cause of code navigation barriers is the screen reader which constrains users to read code line-by-line. The intrinsic nature of the screen reader to provide information to its users sequentially introduces a number of navigation challenges for blind programmers and students [16], [17], [28]–[30].

Mealin et al. [17] in a study with eight blind programmers reported that one of the navigation challenges is the difficulty to find information within a code base without losing the position of focus. While a sighted programmer can easily scroll up and down to find information without changing the cursor location, this is not possible for a blind programmer who depends on a screen reader. This finding is corroborated by Albusays et al. [16] who studied code navigation challenges with 28 blind professional programmers. In [16] the authors report that because screen readers provide information line-by-line, it is less efficient and difficult for a screen reader user to quickly and easily find specific information within a code base as desired. In addition to the screen reader constraint, the syntactic structure of the code also contributes to the difficulty with moving around in a source code while keeping an awareness of the position of focus for screen reader users [28].

Because screen reader users often lose their location of focus when navigating to find information, it introduces another problem of backtracking [16]. This is the difficulty to quickly and easily return to a previous line of focus when reviewing code or seeking information at another location or file in a lengthy code base. Mealin et al. [17] found out that to overcome the problem with backtracking, blind programmers often rely on temporary text-buffers such as text editors and scratch pads. They write notes and keep important information such as variable and method names in these text buffers. This allows them to quickly reference information without losing their current focus in the main file. This finding is corroborated by other findings in [28] and [31]. Albusays et al. [28] also reported that in addition to text buffers, some blind programmers use a combination of a screen reader and braille display to overcome these challenges. Other blind programmers use the search tool or keyword search to easily locate information within code[16], [17]. Not being able to accurately maintain focus through a screen reader also causes developers to enter code at incorrect or unintended locations [32]. This can cause frustrations as reported by Alotaibi [32].

As stated earlier, Albusays et al. [16] conducted a study focused on challenges of code navigation alone with 28 blind programmers. The authors also reported that navigating into and out of nested structures such as *methods, conditional and control statements* is also challenging for blind programmers. The visual aids such as the alignment of special characters or indentations that mark the start and end of nesting are not easily accessible through a screen reader and makes it difficult for those who rely on these tools to keep track of the level of nesting as they go through source code. Related to the nesting, participants in the study reported being unable to distinguish the level of whitespace through a screen reader in languages such as Python. Huff et al. [31] reported similar findings with participants in their study further describing these navigation tasks as time-consuming due to the constraint by the screen reader to only read code one line at a time.

Other navigation challenges reported in the literature involves navigating the programming environment or accessing various IDE features that are design to increase the productivity of programmers. Smith et al. [30] found that the package explorer provided by Eclipse to allow users easily explore the structure of a program and navigate within this as desired was inaccessible to users who relied on a screen reader. Even in cases where this was accessible, the functionality exposed did not match the logical meaning of the structure presented by the package explorer. For example, users could use the Up-arrow key to move up the tree structure, but the screen reader conveyed no information about the logical meaning and relationship of a node with its siblings or parents. Therefore, they developed a plugin to communicate the program structure and file hierarchy and allow easy navigation for screen reader users. The inaccessibility of IDE features that aid programmer navigate lengthy code bases have also be reported in [16], [31] and [33].

While the above navigation challenges have been reported for text-based programming languages and IDEs, researchers have also reported accessibility challenges on block-based programming environments. Reports on navigation challenges in BBLs are few, and mostly relate to navigating the coding environments with assistive technologies used by people with visual impairments. Ludi et al. [14] point out that block-based programming environments are primarily mouse-driven and do not support a keyboard and screen reader for navigation and interaction. In [34] the authors assessed the accessibility of Scratch, a mainstream block-based programming environment to people with visual impairments in a study with 9 participants with visual impairments. They reported that of the nine participants, only two were able to successfully complete coding tasks on Scratch. These two participants used screen magnifiers, suggesting they had some level of vision. They reported several challenges such as lack of focus navigation which limited participants ability to find different sections of the interface. These are cause by limited feedback from the system as Scratch is incompatible with screen readers making it difficult for the participants to use the interface [34].

Mountapmbeme and Ludi [35] also reported that students with visual impairments face navigation challenges on Swift Playgrounds, an accessible hybrid of text and block-based programming platform that runs on the iPad. The report however offers little insights in the causes of these challenges. We observed that unlike text-based programming environments, little is found in the literature about the navigation challenges in block-based programming and their accessibility in general. This can be attributed to the lack of fully accessible block-based programming environments[36], [4].

As stated above, accessibility of code comprehension has mostly been implicitly studied under code navigation. However, Armaly et al. [37] is the only work found that attempts to study the program comprehension strategies of blind programmers in isolation. They authors compared the ability of blind and sighted programmers to read and summarize source code. The authors found that despite the constraints imposed on the screen reader, blind programmers use similar strategies as sighted programmers to comprehend a piece of code. Blind programmers focus on method signatures and return to re-read the methods signatures more often than the method bodies. Unlike sighted programmers however, blind programmers focus less on method invocations to comprehend code. The authors also report that there are no perceived differences in code summaries generated by blind and sighted programmers. This knowledge can be useful in the design of accessibility tools for blind programmers. One of the limitations of this work is that the length of the source codes used in the study was limited to 22 lines of code. This is very small compared to the actual size of real-world programs.

3.5.2 Code Debugging Challenges

Our analysis reveals that code debugging challenges are understudied. Research into accessibility on debugging tasks has focused more on designing and building solutions (RQ2). However, there are few research work that mention some challenges associated with code debugging or the process of finding errors in source code.

Researchers agree that the main cause of debugging challenges is the heavy visual nature of debugging tools that are incompatible with screen readers [28][33][38]. Modern IDEs come with a set of tools to aid the programmer efficiently debug code. However, most of these powerful debugging tools are designed for people with full sight. In 2009, Stefik et al. [38] pointed out the ineffectiveness of Visual Studio 2005 to aid blind programmers with debugging. They reported that using the debugger with a screen reader does not provide correct information to the user due to the lack of accessibility considerations from the onset. They found that with screen readers, a user has less or no options to use a debugger to investigate the behavior of a program at runtime. These findings are corroborated by Albusays et al. [28] who added that typical debugging tasks such as finding logical errors in code or analyzing program behavior become challenging for programmers with visual impairments debugging lengthy code bases. They often resort to simple debugging techniques such as "print-f" debugging as workarounds to the inaccessibility of debuggers [28] [33]

Potluri et al. [33] found that most of the real-time information provided by IDEs that are useful during debugging are mostly inaccessible to blind developers. For example, the Watch Window, a debugging tool on Visual Studio that allows developers to monitor the behavior of their program at runtime was reported to be difficult to access. They found that useful debugging information such as breakpoints and values of variables and expressions are not accessible to blind developers unless explicit actions are taken by them. Visual debugging cues such as squiggles that alert uses of syntax errors in real-time are also inaccessible to developers with visual impairments who use screen readers [33].

3.5.3 Code Editing Challenges

Unlike code comprehension, navigation and debugging, not much is found in the literature about code editing by people with visual impairments. However, code editing can be a challenge for blind programmers and people with visual impairments. Mealin et al. [17] found that majority of participants in their study had difficulties editing code in-place and instead preferred out-of-context editing. The participants explained that editing code within the original source file made them lose context. Rather, they preferred to copy a piece of code to a temporary buffer where edits are made before copying the newly edited code back to the original source file. This mechanism allows them to control the location of the cursor in the original file while editing code thus, keeping and tracking focus. In [35], the authors cite code editing as one of the most difficult tasks for students with visual impairments to complete on Swift Playgrounds, a hybrid programming environment that runs on the iPad. This difficulty was attributed to the advanced commands required to use VoiceOver, a screen reader that comes with the iPad.

Programmers generally edit code to fix a syntax error or to improve the logic of an existing piece of code. For the first case, IDEs often use syntax coloring such as red squiggles to indicate a syntax error. This allows programmers with sight to easily identify and correct the error. However, using a visual aid to indicate syntax error introduces a barrier for those who rely on screen readers. CodeTalk [33] employs audio cues to proactively inform programmers of syntax error in their code. Upon hearing an error tone, a developer can request a list of errors found in code. The authors only talk about using the tool to find syntax error but not how the code is edited to fix the errors. Stefik et al. [5] briefly discusses how meta-auditory cues in the form of speech can be used to communicate syntax errors or additional information about a line of code to a screen reader user.

3.5.4 Code Skimming Challenges

Our analysis shows that code skimming challenges are understudied and have often been addressed together with code navigation challenges. Tools such as StructJumper [25], CodeMirrorBlocks [39] and CodeTalk[33] discussed under section 3.6 aid blind programmers get a quick overview of a code base. Mealin et al. [17] reported that blind programmers have difficulties getting a high-level overview of their code because screen readers constrain them to read code line-by-line. They found that blind programmers mostly consult API documentation in order to have a higher-level understanding of the code base without delving into the implementation details. Referencing the API documentation gives them a quick overview of the code structure and of the methods available in the code base. This also depends on the availability of good documentation. Additionally, the lack of access to IDE features such as code folding which allows one to collapse/expand a section of code further makes this task difficult [16].

3.6 Research Question 2

What are the existing and proposed solutions to these barriers?

While some researchers investigated challenges faced by people with visual impairments as discussed under section 3.5, others focused on proposing accessible solutions to overcome these barriers. These innovative solutions fall under three broad categories based on the results of our analysis. These include accessibility plugins and tools, novel accessible programming languages developed with people with visual impairments in mind from the ground-up and lastly accessible programming toolkits that make use of multiple interaction modalities designed for novice learners with visual impairments. The following subsections provides more details about how these solutions addressed the challenges presented above with comparisons where appropriate.

3.6.1 Accessible Programming Tools

As discussed in the previous chapter, existing programming tools introduces many challenges for people with visual impairments. As a solution, novel accessible programming tools were developed. These tools either serve as plugins to enhance the accessibility of existing tools, or they are stand-alone tools. Several plugins exist and each address a set of programming challenges faced by people with visual impairments by augmenting the capabilities of existing IDEs such as Eclipse, NetBeans and Visual Studio. This augmentation generally involves accommodating alternative input and output interaction mechanisms for people who rely on assistive technologies. Some of these mechanisms include keyboard shortcuts, tactile objects, audio cues and screen reader support. However, adding alternative input and output mechanism to IDEs as plugins can help but there is also a need for programming tools that incorporate accessibility from their onset. This section presents tools from these two categories and compares them. We present a list of the tools in Table 3.3.

Table 3.3: Plugin tools to enhance accessibility in IDEs

Tool Name	Stand-alone/ Plugin	Date Pub'd	Challenge Addressed	Input Mechanism	Output Mechanism	IDE	User Eval?
JavaSpeak [40]	Stand-alone	2000	Code Navigation, Code Skimming	Keyboard shortcut	Audio cue – speech	N/A	No
Aural Tree Navigator [30] [41]	Plugin	2003	Code Navigation, Code Skimming	Keyboard shortcut	Text-to- speech system	Eclipse	Yes
StructJumper [25]	Plugin	2015	Code Navigation, Code Skimming	Keyboard shortcut	Screen reader	Eclipse	Yes
AudioHighlight [42]	Plugin	2018	Code Navigation, Code Skimming	Keyboard shortcut	Screen reader	Eclipse	Yes
Code Mirror Block [39]	Stand-alone	2019	Code Navigation and Code Editing challenges posed by visual browser-based tools	Keyboard shortcut	Screen reader	N/A	Yes
Tactile Code Skimmer [43]	Plugin	2019	Code Navigation, Code Skimming	Keyboard text input	Tactile object, screen reader	Visual Studio	No
ACONT [44]	Stand-alone	2018	Code Navigation, Code Skimming	Keyboard arrow keys, and 1,2,3 keys	Audio cue-earcons	N/A	Yes
CodeTalk [33]	Plugin	2018	Code Navigation, Code Comprehension, Code Editing, Code Debugging	Keyboard shortcut	Audio cue-earcons, speech	Visual Studio	Yes
Wicked Audio Debugger [45]	Plugin	2007	Code Debugging	Keyboard text input	Audio cue-speech	Visual Studio	Yes
SODBeans [38] [5], [46]–[48]	Plugin	2008	Code Debugging	Keyboard text input	Audio cue - speech, screen reader	Netbeans	Yes

Tool Name	Stand-alone/ Plugin	Date Pub'd	Challenge Addressed	Input Mechanism	Output Mechanism	IDE	User Eval?
Accessible version of Blockly by Ludi et al. [36]	Plugin	2015	Inaccessibility of the BBL Blockly	Keyboard shortcut	Screen reader	Blockly	Yes
Accessible version of Blockly by Caraco et al. [49]	Plugin	2019	Inaccessibility of the BBL Blockly	Touch screen	Screen reader	Blockly	No
BridgIT [50]	Stand-alone	2011	Inaccessibility of the BBL Alice	Keyboard text input	Screen reader	N/A	No
Blocks4All [4], [8], [51]	Stand-alone	2018	Inaccessibility of BBLs	Touch screen	Screen reader	N/A	Yes
Noodle [52]	Stand-alone	2014	Inaccessibility of visual programming systems	Keyboard shortcut	Text-to-speech system	N/A	No

JavaSpeak is one of the earliest programming tools created for people with visual impairment. It is a program editor for novices with visual impairments to learn how to program in Java [40]. It helps students with visual impairment to get an overview of code and to easily navigate through code. It was built with audio output and keyboard shortcuts as its interaction mechanisms. The tool uses aural cues to communicate most of the syntactic and organizational information in a Java program at different levels of granularity to blind students. The smallest level of granularity given by the aural cues is token by token, while the largest is compilation unit. These aural cues can be generated by several aural cue functions and rendered using a speech reader by IBM called ViaVoice. The system also has a navigation subsystem which uses the same function-key mappings as Jaws for windows, to help users to easily navigate and focus on different sections of a program. Using this navigation scheme users can also select fragments of their code for aural rendering. As a future work, the authors planned to extend JavaSpeak to support code debugging.

One of the very first accessibility plugins for people with visual impairments is Aural Tree Navigator created for the Eclipse IDE by Smith et al. [30] [41]. Like JavaSpeak it was implemented to address code navigation and code skimming challenges. Using this tool, blind programmers can easily navigate the hierarchical structure of a program directory via keyboard inputs and speech output. The plugin provides a nonvisual access to the file hierarchy of a large code base through a tree-like structure. Using carefully selected keyboard commands (derived through formative user studies), a user navigates between the nodes of the tree. Each keyboard command on a tree node produces speech output that describes the node, its logical position within the tree and other useful information that a sighted person will normally get by just looking at the package explorer on Eclipse. The authors also produced a set of guidelines the highlights the key features that an accessible tree navigation system must have.

Although the Aural Tree Navigator focused on navigating the program organizational structure (file hierarchy) and not navigating the source code (code structure), it served as a foundation for the development of more plugins that would focus on addressing code navigation challenges. StructJumper [25] was inspired by Aural Tree Navigator [30] [41]. It is an Eclipse plugin that allows blind developers overcome code navigation, code comprehension and code skimming challenges. StructJumper creates a hierarchical tree structure that represents the code structure of a Java class. The tree is a simplified abstract syntax tree (AST) of the source code based on nesting information. Nodes represent abstract constructs such as class, methods, conditional statements etc. It is like the Aural Tree navigator in that it uses keyboard shortcuts for navigating through the tree structure. However, unlike the Aural Tree Navigator that provides a separate speech output, StructJumper was designed to be used with screen readers. It helps blind programmers to navigate and get contextual information about the underlying source code using a keyboard and screen reader. A user can easily switch between a node in the tree, to the actual location of the code representing that node in the text editor and vice versa, without losing focus of their previous location. This minimizes the issues with backtracking mentioned earlier. It also allows users to quickly skim through a source code by browsing through the nodes of the tree structure.

AudioHighlight [42] is a tool that is closely similar to StructJumper [25]. However, unlike StructJumper that addresses a wide array of programming challenges, AudioHighlights focuses on code skimming challenges alone. Also, it is not an IDE plugin like StuctJumper. It was designed specifically to allow blind programmers skim through code that is hosted on the Web. Rather than generating an AST, this tool uses HTML tags to surround structural information such as classes, functions and control flow statements. The type of heading tag (h1, h2, etc.) attached to an element depends on its relative location within the source code. This creates a hierarchical structure of the source code that is accessible through a screen reader due to the compatibility of screen readers with HTML. To evaluate AudioHighlight's ability to read code on the web, the authors conducted a set of code skimming experiments using GitHub as the control. The tasks required participants to read and summarize a source file. The authors found that participants were able to perform tasks easier and faster using AudioHighlight. AudioHighlight was also evaluated against StructJumper. To accomplish this the authors developed a plugin version of AudioHightlight for the Eclipse IDE. Though there were no significant differences in performance and accuracies, the reported results showed that completing tasks with AudioHighlight was relatively faster than with StructJumper and that AudioHighlight has a shallower learning curve.

Code Mirror Block (CMB) is a language independent toolkit that creates accessible, browser-based programming environments [39]. It addresses code navigation and editing challenges faced in most visual browser-based tools. However, in this paper the tool was only evaluated in terms of code navigation. Like StructJumper and related plugins, it creates an AST-based structure of the source code that users can easily access for code navigation purposes using keyboard shortcuts. However, with CMB, the nodes on the tree are rendered as blocks (visual elements) that can be collapsed or expanded as needed to provide contextual information at different levels of detail. Because it runs on the web, the authors use ARIA attributes [53], to easily make the blocks accessible via screen readers. Each node provides a plain language description of the code, which alleviates the users from hearing the spoken raw text of code by screen readers. Another major difference between CMB and other plugins discussed above is that CMB is intended for language developers who want to add accessibility to their platforms. It is not ready-made for the end users. As of the time of publication, the tool was still under development, and the authors stated that they will present another use of the tools in editing code. It will be interesting to see how this tool helps address some of the challenges with code editing for blind programmers as there has been relatively less attention on code editing and its challenges.

Tactile Code Skimmer is Visual Studio Plugin designed for solving code skimming challenges faced by blind developers[43]. However, as the name implies it is a physical device unlike AudioHighlight and StructuJumper. It takes a different approach from the common tree structure-based navigation discussed earlier. The device is made of six horizontal sliders which a user can feel with their hands. The indentation level of a line of code on Visual Studio editor is reflected by the corresponding slider position on the device. Only six lines of code can be mapped at a time. However, blocks of code at the same indentation level are collapsed together onto one slider, allowing the user to quickly skim through a source file. This tool helps programmers who rely on screen readers with code navigation and skimming by circumventing the imposed line-by-line navigation of screen readers. Like StructJumper and AudioHighlight, it also helps users maintain focus on the source file through navigation buttons on the device that allows the user to move the cursor freely on Visual Studio Editor. The authors argue that using this device to communicate information about the structure of the code will remove some of the hearing load imposed by screen readers while communicating such information. However, having an additional device to manipulate with the hands might be intrusive to the normal keyboarding tasks.

The Auditory Code Overview and Navigation tool (ACONT) [44] addresses code skimming and code navigation challenges without creating an underlying AST-structure of source code. The tool uses audio feedback and keyboard shortcuts for navigation to help programmers with visual impairment quickly skim through a class file to get an overview of its structure using a technique called timeline navigation. With this technique users can scroll through a file line-by-line at their own pace. The tool uses the arrow keys to navigate up, down, left, right; the keys 1,2, and 3, to jump to the beginning, middle, and ending of a file. These careful key selections allow users to quickly jump to different locations within a class file. Navigating to a line of code produces sound cues that describe the content of the line. In ACONT, non-speech sounds are used to represent programming constructs, for example, when an if statement is detected on a line a door opening sound is played. The authors argued for the use of non-speech audio cues to provide an overview of source code. To confirm this, they conducted a user study analyzing the performance of three different types of audio cues, speech, non-speech, and spearcons on their ability to accurately and pleasantly present an overview of a class file. Although speech was found to be the most accurate, non-speech sounds was the most preferred as it was less cumbersome and more appealing.

CodeTalk [33] is an accessibility plugin that is similar to StructJumper[25], AudioHightlight[42] and the Aural Tree Navigator[30] as it also creates a tree structure to address code navigation and code comprehension barriers. In addition to the tree structure however, it also provides a summarized list of functions. This list of functions allows a blind developer to quickly and easily navigate from one function to another thus getting a bird's eye view of the source code. Both the tree structure and list of commands can be accessed using keyboard shortcuts. The major difference between CodeTalk and the previously discussed plugins is that it also addresses code debugging challenges. Potluri et al. in their design of CodeTalk incorporated an audio debugger that provides accessible alternatives to use debugging features such as breakpoints, access to variable values as well as alternatives to visual syntax error indicators such as squiggles. These features are provided through audio descriptions. The debugger in CodeTalk provides the developer with the opportunity to mark and track specific locations in code during debugging like how breakpoints are used in visual debuggers, this feature is called talkpoints. The authors only talk about using the tool to find syntax error but not how the code is edited to fix the errors. CodeTalk is a plugin for the Visual Studio IDE version 2015/2017 and supports C# and Python programming languages. CodeTalk was evaluated in a user study and it was found to be useful in debugging.

The Wicked Audio Debugger (WAD) [45] is an accessible debugger similar to CodeTalk [33]. Like CodeTalk, WAD helps in code debugging by providing audio descriptions of programs during execution. WAD was also designed to work with Visual Studio, however, it supports version 2005. Through pilot studies and results from previous work, creators of WAD found out that plain speech worked best for sonifying program construct. Their pilot studies also allowed them to experiment with different speech properties, such as speed, tone, pausing between utterances. The authors found that pausing between utterances increased the accuracy of understanding the sonified program. The result from the pilot study informed design decisions such as carefully choosing words that reduce ambiguity and prepending the nesting level information of a control flow statement before describing the statement itself to help covey nesting information. One of the limitations of WAD is that it does not discuss how the tool can be used to track errors within a program. This is one-way CodeTalk differs. Unlike WAD which only allows a developer to track the execution flow of a program only, CodeTalk offers the option to mark and track specific locations in code during debugging through its *talkpoints* features.

The creators of the plugin WAD discussed in the paragraph above, also created a tool called SODBeans (Sonified Omniscient Debugger for Netbeans) [5], [46]–[48] . It is an audio-based programming environment that uses audio stimuli to communicate information to the users. The initial version called SOD was not created for distribution but rather to demonstrate the use of audio cues to aid with computer program comprehension [54]. SOD was designed for the C programming language [46]. It included a C-compiler, a custom debugging architecture, sound libraries, and a robust system for designing auditory cues. SODBeans was built from SOD; the difference between the two is that SODBeans is designed to be used with Netbeans 6.5. Similar to SOD, SODBeans includes a compiler, custom screen reader, and talking debugger. Stefik et al. evaluated the designed auditory cues and SODBeans during a workshop with twelve students with visual impairments [5]. The workshop was designed to evaluate the effect of these concepts and materials on the programming self-efficacy of the students. After the workshop, students expressed increase in their programming self-efficacy.

Alternative input and output mechanisms such as keyboard shortcuts and audio cues are also useful in making visual programming environments accessible. In [55], Ludi et al. investigated the use of auditory cues in helping with code navigation and code comprehension in the block-based programming environment, Blockly. Three type of audio cues were tested, speech, spearcons and earcons. They found that speech gave the best performance with spearcons sometimes matching speech in performance. Earcons had the worst performance both in comprehension and navigation. Furthermore, Ludi et al. redesigned the Blockly framework to support keyboard navigation and screen reader output while keeping its original visual and mouse-centric nature unchanged [14], [36]. In the new version of Blockly, using keyboard shortcuts and screen readers, users will be able to perform actions such as code creation, editing and navigation. Caraco et at. also discuss a redesign of Blockly to add accessibility for touchscreen devices [49]. They present two alternative protype designs that allow program creation, navigation, editing and reading of blocks using touch gestures and built-in screen readers on tablets. One design uses a hierarchical list representation of blocks on the workspace and the other uses a spatial representation like the original visual representation of BBLs These designs were inspired by Blocks4All [4] discussed in the next paragraph.

The architecture of some visual programming environments made it nearly impossible to incorporate accessibility into them, hence the need for a new system. An example of this can be seen in [50], a novel tool BridgIT was developed after Ludi et al. unsuccessfully attempted to incorporate accessibility into the tool Alice. Alice is a drag and drop programming environment that creates computer animations using 3D models. It is used to teach object-oriented programming through the development of animations. Alice was found to be incompatible with screen readers. BridgIT was built to be screen reader compatible by using a textual object-oriented language, which can be represented with visual elements. Thus, people with visual impairments can use screen readers to read written programs. Similarly, Milne et Ladner implemented a novel touchscreen blocks-based programming environment called Blocks4All that is accessible to children with visual impairments [4], [8], [51]. This tool was created in response to the inaccessibility of BBPEs. Blocks4All was developed as an iOS iPad application, it uses the functionality of Apple's built-in screen reader VoiceOver and zoom capabilities. It is used to control the Wonder Workshop Dash and Dot robots. In their design, blocks can be accessed using the screen reader VoiceOver. The screen reader provides the name, the location and the type of the block, it also gives hints on how to manipulate the blocks. Movement in Blocks4All is designed to mirror the drag and drop method of block-based programming environments. The user first selects a block and then selects a destination. After selecting a block, a list of valid connection points for that block will be added to the

2-D representation of the code in the workspace. The user can then select one of the points to add the block.

A strategy adopted in response to the inaccessibility of visual programming systems is non-visual visual programming. Non-visual visual programming (NVP) suggests a way to make visual programming systems operable by non-sighted users. They are based on the Raman's principle which states that '*any visual information can also be represented in a nonvisual way*." An example of an NVP system is Noodle [52]. Noodle provides nonvisual access to a dataflow programming system, a popular model for visual programming systems. In dataflow programming, programs are created by connecting the input and output of a functional unit to the output and input of other functional units. Noodle combines audio, keyboard commands with speech output to support the creation, execution, and modification of dataflow programs, with an option for visual presentation. Thus, the current version of Noodle has two user interfaces, a visual interface to display diagrams and a non-visual interface that uses keyboard command and audio. The current version only uses four key commands (arrow keys) for navigation while the initial version used 8 keyboard commands. Noodle is implemented in JavaScript as a Web application. The researchers suggest that the same approach used to develop Noodle could be applied to other visual programming systems to increase their accessibility for learners with disabilities; an example of how this can be applied to the BBP application Scratch was given.

3.6.2 Accessible Programming Languages

In addition to enhancing existing programming tools using plugins or creating novel standalone tools, new accessible programming languages have been created to be used in these tools or environments. Table 3.4 presents a summary of these languages.

Table 3.4: Novel accessible programming languages

Name	Language Type	Date Pub'd	Challenge (s) Addressed	IDE	User Eval?
Scripting Language by Siegfried et al. [56]–[60], [56]	Text-based	2002	Inaccessibility of GUI-based applications	Any text-editor, and a console application for compilation	No
GUIDL [61]	Text-based	2012	Inaccessibility of GUI-based applications	N/A	No
Quorum [62]	Text-based	2015	All challenges posed by existing IDEs	Quorum studio, console	Yes
Pseudospatial Blocks [63]	Block-based	2016	Inaccessibility of BBLs	Pseudospatial Blocks	No
APL [64], [65]	Audio-based	2005	Inaccessibility of existing programming languages	APL	Yes

One of the earliest programming languages designed for accessibility is the scripting language by Siegfried et al. [56]–[60], [66]. This language was proposed as a solution to the inaccessibility of a GUI-based visual programming application, Visual Basic (VB). In VB, applications are created by dragging and dropping components or controls (e.g., text boxes, buttons, etc.) on a form or by editing a text file with detailed information. These requirements make it difficult for the blind to create applications. In contrast, this scripting language allows persons with visual impairments to create and modify VB forms without the need for providing intricate details or dragging and dropping components. In scripts, object placement is handled by dividing the screen into three rows and three columns, the preferred location is specified as a property value in the script (e.g., top left, middle right). Scripts can be created and edited using any text editor and compiled using a console application. The file generated after compilation can be imported into the visual basics project. The syntax of the scripting language is designed simple and to look like VB as much as possible.

A language with a similar syntax as the scripting language by Siegfried et al. is GUIDL (Graphical User Interface Description Language) [61]. Like the scripting language it was created to address the inaccessibility of GUI-based applications. GUIDL source code was inspired by the programming language BASIC, which has a similar syntax to the code text portion of Visual Basic. However, unlike the scripting language which can be imported into Visual Basic projects, GUIDL has no direct link to BASIC and cannot be interpreted by BASIC compilers. Using this language, blind programmers will be able to create UI elements using natural language. The GUIDL system is made up of the following elements: GUIDL language, GUIDL source code, and a mediator that translates GUIDL source code to UI code for a given programming language. Syntax of GUIDL language was designed to be simple and easy to use, this enables it to be used not only by programmers but also hobbyists. In the initial prototype only the most used controls (buttons, forms etc.) in applications development was supported. As a future work, GUIDL current control set would be extended to include more UI elements.

The creators of the non-visual visual programming (NVP) tool Noodle [52] discussed in section 3.6.1, also created an NVP language called Pseudospatial Blocks (PB) [63]. As discussed non-visual visual programming address the challenges non-sighted users face in visual programming systems, similar to the last 2 languages discussed. In particular, PB was created in response to the inaccessibility of block-based programming languages. It is a blocks language presentation based on arrow key navigation. In PB, programs are created using keyboard commands and text-to-speech (TTS) output. To create a program, the user selects an insertion point to place a new block, a list of compatible blocks will be presented which the user can then select from. PB is implemented as a Web application that generates a representation of its constructed programs in JSON (JavaScript Object Notation) format. The JSON object is translated to XML, PB then uses the library of the tool Blockly [67], to convert the XML code to JavaScript code. The Blockly library was chosen due to its ability to generate code in various languages based on the XML representation of blocks. Feedback is given to the users using synthetic speech. As at the time of presentation, PB was not ready for testing. They intend to use the feedback received from testing to refine the system.

An audio-based programming language called Audio Programming Language (APL) was presented in [64], [65]. This was the first environment that relied solely on the use of audio for input and output. This language was created in response to the challenges that blind learners face in programming languages based on visual interfaces. APL provides a limited set of commands for writing and executing programs. The user listens to the available commands and makes selections using the keyboard or provide input to the system using voice. APL then translates the constructed program to an actual program. The output of the constructed program can be in form of earcons or text-to-speech. Commands are also available to help the user follow the execution of a program, thus assisting with debugging. One issue with APL is that it is developed to introduce novice blind learners to programming and little is known as to how it can support the transition of these learners from APL to more powerful and mainstream languages.

In contrast to APL, an accessible language that is powerful and widely adopted is the Quorum Language. It is often called an evidence-based language because its design decisions are backed by empirical evidence [62]. It was initially created to help people with visual impairments who use screen readers to learn programming, however, due to its popularity it was extended for use by people with various abilities. It is currently incorporated in K-12 CS curriculums and universities globally [68]. Quorum can be used to create different types of applications, such as 2D/3D games, digital signal processing, Lego robotics and many other applications. The syntax of the language is very straightforward and easy to understand. Quorum Studio, the programming interface for quorum programming language supports keyboard navigation, screen readers, and magnification software.

3.6.3 Accessible Programming Toolkits

The use of programming toolkits is common in introductory programming settings. These toolkits involve playful activities such as toy blocks, games, music, robots. These activities tend to be exciting enough to maintain children engagement as there is constant feedback about the actions being performed. Sadly, many mainstream toolkits for simplifying programming activities in school are inaccessible to visually impaired (VI) children [69]. To address this, researchers have developed novel computing education toolkits suitable for children with visual impairment. Table 3.5 presents these toolkits.

Table 3.5: Accessible programming tool kits

Tool	Date Pub'd	Challenges Addressed	Input Mechanism	Output Mechanism	User Eval ?
Bonk [69]	2018	Inaccessibility of programming through media such as games	Keyboard shortcut, Keyboard text input	Text, screen reader, text-to-speech system	Yes
Story blocks [70], [71]	2017	Inaccessibility of BBLs	Tangible objects	Audio cue - speech	Yes
TIP-Toy [72]	2020	Inaccessibility of BBLs	Tangible objects	Audio cue – earcons	Yes
Music Blocks [73]	2020	Inaccessibility of BBLs	Tangible objects	Audio cue - earcons	Yes
Torino [74]–[77]	2017	Inaccessibility of BBLs	Tangible objects	Audio cue – earcons, speech	Yes
Tangible Programming Tool Prototype by Utreras et Pontelli [34]	2020	Inaccessibility of BBLs	Tangible objects	Audio cue- earcons	Yes

Tool	Date Pub'd	Challenges Addressed	Input Mechanism	Output Mechanism	User Eval ?
CodeRhythm [78],[79]	2020	Challenges of some tangible programming environments	Tangible objects	Audio cue – earcons	Yes
Sparsha [80]	2020	Challenges faced when editing visual layout templates	Tangible objects	Web design layout	No
CodeBox64 [81], [82]	2017	Inaccessibility of BBLs	Tangible objects	Block-based programming	Yes
P-CUBE [83]	2013	Inaccessibility of BBLs	Tangible objects	Robot control	Yes
JBrick [84]	2010	Inaccessibility of robotics programming environments	Keyboard text input, magnification software	Screen readers, refreshable braille displays	Yes
Robbie [85], [86]	2012	Inaccessibility of visual feedback produced by robots	Keyboard text input	Audio cue - speech	Yes
COBRIX [87]	2017	Inaccessibility of existing programming tools	Tangible objects	Robot control	No
Phogo [88]	2017	Inaccessibility of visual programming languages	Keyboard text input	Robot control	Yes
Donnie [89]	2017	Inaccessibility of visual programming languages and robotics toolkit	Logo-based language	Robot control, Audio cue: screen reader, text-to-speech	Yes
TACTOPI [90]	2020	Lack of playfulness and engagement in accessible programming environments	Tangible objects	Robot control, audio cue/sound	No

Kane et al. spoke about the challenges of existing programming toolkits and they presented Bonk as possible solution [69]. Bonk is an accessible programming toolkit that enables novice programmers of all abilities to create accessible and interactive audio games. It was created to address the challenges that blind programmers face in programming tools that involve medias such as 3D-animations, or video games. Bonk offers an abstracted interface for creating accessible games; this helps hide the complexities of its underlying technologies. It supports two forms of input: key presses and text input. The audio output from Bonk programs can be in form of text, screen reader speech, or self-voicing Text-to-Speech (TTS) system (main component). The TTS parameters can be modified such as voice, speed, playback speed, and volume. The TTS output is generated by HTML5 Web Speech API, and the sound effects is by Web Audio API. Bonk tool was implemented using JavaScript, this was done to enable it run on any device with a browser. Programs in Bonk are written in HTML and JavaScript; and can be shared and run through a node.js based web application using a MySQL database.

A common strategy in accessible programming toolkits is the use of tangible objects. In [91], educators report the use of tangible as a best option for children with visual impairments in the absence of accessible block-based programming environments. This programming strategy is called tangible programming. In tangible programming, programs are created by manipulating or arranging physical objects. Akin to Bonk, the output of a lot of tangible programming tools is audio. Some examples include Torino, StoryBlocks, Code Rhythm etc.

In the tool StoryBlocks [70], [71], the tangible objects are in the shape of story characters, actions, or control statements. The blocks are made with low-cost materials. Each block has a visual tag and a tactile symbol. To create a program, blocks are arranged in a designated workspace. To compile the constructed program, a photo of the blocks' arrangement on the workspace is taken and sent to Python-based application on a PC. This application interprets the program using the block's tags, and the corresponding audio story is created. Each line of the story can be read by the screen reader of the device. As a future work, they intend to use StoryBlocks to produce simple audio games.

A tool that is quite similar to StoryBlocks is TIP-Toy [72]. It also makes use of tangible blocks created from passive material, and the programs are compiled using a camera device. However, it outputs music not audio stories, and it is open-source unlike StoryBlocks. Music Blocks [73], is another tangible tool made from a low-cost material, cardboard. It also has in common with TIP-Toy the ability to produce music as output. The prototype consists of blocks containing an NFC sticker, a rack for arranging the blocks, and an android application. Like StoryBlock and TIP-Toy, a device is used to scan constructed programs. However, rather than take a photo, to parse a program, the NFC stickers of each block is scanned through the android application installed on the device.

Torino is a tangible tool that produces both audio stories, music and poetry as output [74]–[77]. In contrast to StoryBlocks, TIP-Toy, and Music Blocks, in Torino, the physical objects being manipulated are instructional beads with embedded electronic. Each bead represents different programming instructions such as play, pause and loop. The play bead instructs the program to play a particular sound. The sound to be played can be adjusted by rotating the dial of the bead. Beads are connected to one another and to a central hub using cables. The hub is connected to a Raspberry Pi device which contains a script that translates constructed programs. Although not open-source, Torino is commercially available under the name Code Jumper [92]. A prototype created by Utreras et Pontelli is similar to Torino [34] in that it also uses music as an output, and it involves objects with embedded electronic. However, the tangible objects in this case are Lego blocks. To create programs, Lego blocks with a male audio jack are connected to female audio jacks placed in a Lego panel. Each Lego block represents a different code construct. Individual blocks can be identified by a 3D printed symbol on top and a Braille label in front. The authors never mentioned how a constructed program is compiled or executed.

In [78],[79], Rong et al. spoke about some of the limitations of existing tangibles such as Storyblocks [70], [71], Torino [74]–[77], and Music Blocks [73]. The authors reported that while using Torino, children with VI encountered challenges when connecting and disconnecting instruction beads to ports. They also reported that students were confused about the right cable direction and the location to plug the wire into sockets. Regarding StoryBlocks, the authors spoke about how the use of a camera to scan the workspace limits the dimension of programs to only the area that can be captured by the camera, and that image detection is also highly dependent on lighting. Additionally, the authors mentioned that Music Block's rack did not intuitively depict programming sequence, and that scanning blocks using a phone is difficult for children with visual impairment. As a possible solution the authors present CodeRhythm, an accessible tangible programming toolkit that creates simple melodies. Some ways CodeRhythm achieves accessibility is by using embedded magnets and conductive tapes to connect blocks together, this addresses the connection burden associated with the use of wires or cables to connect blocks. Not using cables also shortens the program length, making it easier for users to trace programs. Additionally, the magnetic force in blocks helps to repel incompatible blocks. Lastly, CodeRhythm provides multisensory feedback. Individual blocks not only contain tactile info but also the ability to press them and listen to their sound note. This is achieved by installing an Arduino Mini board and speaker module in each block, which enables each block to be controlled independently.

Tangible programming toolkits are not only used to create audio sounds; there are several other ways they can be used. They can be used to design web layouts as seen in the tool Sparsha [80]. Sparsha, uses 3D printed tactile beads to design web layouts. The tactile beads represent different HTML elements. The beads are placed on a board that represents the screen layout. This board contains an Arduino sensing circuit, which can detect the bead type and location. The sensor sends this information to the Web server using an API. This information is translated to HTML layout and the corresponding image is displayed on the client's browser. Another novel way tangible objects can be used, is as an input modality to block-based programming languages. The tool CodeBox64 offers a tactile approach to providing input to Scratch [81], [82]. It is a peripheral device consisting of six buttons. Four of the buttons allow the user to control the mouse cursor, and the other two buttons allow the user to select different menus and drag and join the blocks in the script. As users navigate through the Scratch, CodeBox64 generates auditory cues to indicate which block the users are trying to drag and drop.

Tangible objects can also be used to program robots, this can be seen in P-CUBE [83] [93][94][95]. The tool is made up of a movable robot, a program mat, wooden cubic blocks with radio frequency identification (RFID) tags, and a computer. The robot consists of an Arduino UNO microcontroller board, a wireless SD shield, batteries, and a microSD card. To create a program, RFID readers are placed on the program mat, when the user arranges blocks on the program mat according to the structure of the given algorithm, the RFID readers reads the block's RFID info containing the location of each block and sends this info to a PC which is connected to the mat through a cable. The computer uses this info to generate the program and then transmit it to the mobile robot through the microSD card. The robot then moves according to the generated program. COBRIX is another tangible toolkit that programs robots [87]. It makes use of the LEGO Mindstorms robotics toolkit. Its interface consists of Lego bricks, mat, camera of a computing device. Each brick represents a programming construct. The Lego bricks are placed on the mat. Similar to the StoryBlocks, TIP-Toy, and Music Blocks, using the camera of a mobile device, a user takes a photo of the bricks' arrangement on the mat. The image is translated and then used to provide direction to the Lego Mindstorm robot.

As a future work, they intend to add auditory feedback to COBRIX to help visually impaired users to comprehend the output of their programs.

Akin to tangible objects, robot toolkits are a popular component of accessible programming toolkits. Dorsey et al. conducted a study to evaluate which robot building kits were better suited for engaging students with visual impairment [96]. They found Lego Mindstorms robotics toolkits were the most suitable for students with visual impairments to build and program robots. Originally the graphical user interface used to program robots in this toolkit is NXT-G, however, Ludi et al. found it to be inaccessible [97]. As a solution Ludi et al. implemented the tool JBrick [84], [98], [99], a more accessible text-based programming interface for programming the Lego Mindstorms NXT robots. The language used was in JBrick, is NXC (Not Exactly C). JBrick's user interface is designed to be simple and accessible to programmers with various degrees of vision through features such as compatibility with screen readers, refreshable braille displays and magnification software, ability to change background color, text size and color, etc. JBrick was implemented using Java, this allows it to be cross-platform. Remy et al. also incorporated the Lego Mindstorms NX in their toolkit. Their toolkit consists of a Lego Mindstorms robot and an application called Robbie [85], [86]. Robbie was designed to help visually impaired students in programming robots by providing audio feedback for executed robot code. This tool automatically generates a verbal summary of the actions a Lego Mindstorms robot performed or the actions the robot should have performed, from the log created at runtime. Unlike JBrick, the robotics programming interface Robbie uses is Bricxcc, and the programming language is NQC (Not Quite C).

As an alternative to Lego Mindstorms robotics toolkits, some research proposes the use of robots built from lower-cost materials such as 3D printing, wood and Arduino boards. This can be seen in the tool P-CUBE discussed earlier [83] [93][94][95]. It can also be seen in Phogo [88]. This tool consists of a Python library and a robot called Tortoise. Tortoise was built using 3D printing and Arduino board. The python module consists of pre-built functions that when executed leads to motion in the Tortoise. The computer running the python software communicates with the robot's controller using Bluetooth. Tortoise has a pen for drawing shapes on surfaces where it is moving on. Similarly the authors in this paper [89], present a robot called Donnie. Donie can either be a physical or a virtual robot. Like Phogo, the physical robot is made with cheap and easy to find parts, such as Arduino, Raspberry Pi 2 and 3D printing. Similar to Robbie, this tool provides sound feedback and text-to-speech feedback for visually impaired users. The software used to program Donnie is called GoDonnie, it is a Logo-based programming language built on top of the Player robotic framework, running on Linux operating system. Donnie is the first Arduino-based robot which is compatible with Player robotic framework. Using the Logo-based language, programs are created and translated to commands and sent to the robot by the Player robotic framework. Player receives sensor data from the robot and displays it at the console or in log files. Unlike Phogo that uses Bluetooth, WiFi is used to connect to the physical robot or the virtual simulation environment. Another example of a robot built using 3D printing is TACTOPI [90]. TACTOPI is a tangible game that promotes computational thinking. It entails a physical robot-boat controlled with a 3D printed wheel, and a map with tactile elements and braille label. It is implemented using BBC Micro:bit [100] and not Arduino like Phogo and Donnie. The tool was designed to accommodate different age groups and learning levels.

3.7 Discussion and Opportunities for Future Work

3.7.1 The Need for an Increased Focus on Understudied Challenges

Looking at the data on the accessibility of text-based programming languages alone, the areas of code comprehension, code skimming, code editing and code debugging are understudied. Code comprehension challenges have been addressed as a result of studying code navigation challenges. Although, this is understandable because to comprehend a piece of code you must navigate through it, focus should be given to studying code comprehension challenges explicitly. Understanding how people with visual impairments comprehend code will improve the design of accessibility tools and plugins for people with visual impairments [37]. Similarly, although tools such as WAD[45], SODBeans [46] and CodeTalk [33] have been developed to address the inaccessibility of visual debuggers, our data suggests that there has been less focus on examining how people with visual impairments debug code and the challenges thereof. One study suggests that code debugging challenges are a consequence of code navigation challenges [31]. However, there is little data to support this. Given that there are multiple facets to debugging a computer program, more research work is needed that examines the debugging habits of programmers with visual impairments and the barriers that they face while doing so. The same applies to other understudied challenges.

3.7.2 The Need for an Increased Focus on Accessibility of Block-Based Languages

Most of the papers examined focus on accessibility of text-based programming languages and related IDEs. Only a few were found that talk about the accessibility of block-based languages. However, this does not imply accessibility of block-based languages have been neglected. Our data reveals that research into the accessibility of block-based languages is still at its infancy and ongoing, with the first work dating back to 2015 [14]. We believe ample opportunity exists towards making block-based languages accessible. Given the growing importance of block-based languages and their increase use in K-12 curriculum[1][8], researchers and practitioners should extend more effort in this direction to give students with visual impairments the opportunity to learn and practice programming on the same systems that are found appealing to sighted novices[2]. Opportunities to study specific aspects of block-based programming as it relates to similar challenges noted in text-based programming, as well as unique challenges in block-based languages, provide ample room for impact in the field that can solve persistent accessibility issues across diverse interaction paradigms.

3.7.3 The Potential for Alternative Interaction Modalities for BBLs

As a solution to the inaccessibility of VPLs, in particular BBLs, often times new tools are created which involve the use of manipulation of physical objects for input and audio sounds as output. Not enough research exists where existing BBLs are enhanced for accessibility using alternate input and output interaction mechanisms. The few that exist make use of audio cues [55], touch screen [49], keyboard shortcut and screen readers [36].There are other interaction mechanisms that are yet to be explored such as the use of voice or gesture for input, and tactile objects for output. Individually some of these mechanisms have proven to be beneficial in adding accessibility to text-based IDEs [43], [64], therefore, they could also have the same effect in BBLs. A multi-modal approach i.e. the combination of one or two of the mechanism mentioned above, could also be helpful in incorporating accessibility into existing BBLs or in creating new accessible Block-based tools. Additionally, a lot BBLs are not compatible with screen readers, efforts need to be directed towards creating custom screen readers than will be compatible with BBLs.

3.7.4 Looking Beyond the Experimental Studies

The literature is missing information about the day-to-day use of the accessible solutions that have been proposed so far. Current literature offers little or no insight about the usability of currently tools in non-experimental contexts. It will be interesting to know how the developed solutions influence the habit of programmers and students with visual impairments in their daily activities. Where are these tools used? How many people are aware of the existence of these tools and what impact do these accessibility tools have in the day-to-day programming activities of target users? Current literature alone cannot provide answers to these questions. We noticed that although tools such as StructJumper[25] and CodeTalk[33] have been developed and out there for a while, blind programmers still report code navigation challenges in 2020 [31]. There are possible reasons for this. The first reason is that they might be working with programming languages not supported by current accessibility plugins. The second reason is that they may not be aware of the existence of such tools. In a survey with 15 programmers who were blind, Baker et al. [101] reported that only 4 of the 15 participants indicated using tools and IDEs specifically designed for blind people. The paper does not indicate whether the remaining 9 participants knew about the existence of these tools. Researchers should seek ways to improve current strategies used to disseminate accessibility tools in the communities of programmers and students with visual impairments.

3.8 Conclusion

We have presented a literature review of current academic research efforts to make programming languages and programming environments accessible to people with visual impairments. We reviewed and analyzed literature that contributes to improving the accessibility of existing programming environments and languages for people with visual impairments. Our analysis revealed that researchers have uncovered several barriers that affect professional programmers with visual impairments as well as students with visual impairments learning how to code. These barriers include broad challenges with Code Navigation, Code Comprehension, Code Editing and many others discussed in the paper. At the same time the literature has focused more on studying some of these barriers over others, highlighting the need to divert efforts to understudied areas.

Examination of the literature also revealed that researchers and practitioners have also directed efforts towards creating solutions to address these known accessibility barriers. From creating new accessible programming languages, tools, and toolkits to designing accessibility plugins for existing programming environments, researchers have gone a long way to help make programming and programming activities fully inclusive of people with visual impairments as highlighted in this review.

Despite the laudable research progress, gaps and opportunities for future work still exist as highlighted in the paper. We hope this work provides a comprehensive and consistent description of the status-quo that will serve as guidance for the way forward and at the same time act as an entry point for new researchers joining the field.

CHAPTER 4

CHALLENGES FACED BY STUDENTS WITH VISUAL IMPAIRMENTS ON BLOCK-BASED PROGRAMMING ENVIRONMENTS*

The results of the systematic literature review presented in Chapter 3 served as evidence that the accessibility of block-based programming to people with visual impairments is an understudied area and an important area of research that needs focus. However, this was purely from a theoretical perspective. In this chapter, we present empirical work conducted to directly answers PRQ1: What block-based programming environments are currently being used by students with visual impairments in educational settings? And PRQ2: What challenges do students with visual impairments face on current BBPEs? And how do they overcome these challenges?

We start by presenting findings from a qualitative survey study conducted with 12 teachers to students with visual impairments (TVIs). This is followed by a discussion on a follow-up interview with 12 teachers of students with visual impairments to further elaborate and understand the findings from the initial survey study. Lastly, we present findings from a survey study with seven students with visual impairments to show how the students' account corroborates the report by the teachers regarding the accessibility challenges faced by students with visual impairments on block-based programming environments. The work in this chapter was done in collaboration with Stephanie Ludi.

* Parts of this chapter have been previously published, either in part or in full from Aboubakar Mountapmbeme and Stephanie Ludi. 2020. Investigating Challenges Faced by Learners with Visual Impairments using Block-Based Programming/Hybrid Environments. In The 22nd International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '20). Association for Computing Machinery, New York, NY, USA, Article 73, 1–4. https://doi.org/10.1145/3373625.3417998 with permission from ACM. And from Aboubakar Mountapmbeme and Stephanie Ludi. 2021. How Teachers of the Visually Impaired Compensate with the Absence of Accessible Block-Based Languages. In The 23rd International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '21). Association for Computing Machinery, New York, NY, USA, Article 4, 1–10. https://doi.org/10.1145/3441852.3471221 with permission from ACM.

It is also based on the work we published at the 22nd and 23rd International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS 2020 and ASSETS 2021 respectively).

4.1 Survey Study with Teachers of Students with Visual Impairments

With the emerging efforts to increase accessible versions of block-based programming environments, little is known about the challenges that students with visual impairments face while learning how to code on these systems especially in the classroom. However, a lot of research has been conducted to understand the challenges faced by students with visual impairments when coding using text-based programming languages [26]. The output of these research efforts led to the development of many tools and solutions as discussed in Chapter 3. To provide effective solutions to accessibility challenges on BBPE, an understanding of these challenges is necessary in the first place. With that in mind, we designed a study to investigate the challenges that students with visual impairments face on current BBPEs. Understanding these challenges will inform the design and improvement of Accessible Blockly, the accessible block-based programming library discussed in this dissertation.

To proceed with our investigation, we designed a survey study targeted at teachers of students with visual impairments in order to answer the following two research questions:

RQ1. What are the current BBPEs that are being used by teachers who teach students with visual impairments?

RQ2. What challenges do students with visual impairments face while learning how to code on these systems?

The results of the study show that most of the challenges are technical rather than conceptual and range from limited learners' preparedness on the use of assistive technologies to limited or ineffective support and guidance from the BBPE.

4.1.1 Survey Study Design

We used a qualitative approach to answer these research questions because we believe it will reveal more information about the current status of the reality of the instructional usage and context of BBPE by learners with visual impairments. Accordingly, we designed a survey study aimed at the teachers of children with visual impairments. We chose to address our questionnaire at the teachers because teachers can easily spot challenges since they work with students on regular basis. Additionally, understanding the challenges from the teachers' perspective might reveal more insights than from the students alone. In [4] the authors work with a teacher of children with visual impairments to obtain feedback and elaboration about participants' interactions with Blocks4All in order to inform the design of this system.

We initially came up with a set of questions that covered our research questions. These questions were reviewed before submitting to the UNT Institutional Review Board (IRB) for approval to ensure the wordings were appropriate for the audience and easy to understand and at the same time capable of eliciting the information sought. The survey was hosted on SurveyMonkey for online data collection. The survey consisted of multiple choice, Likert-scale and open-ended questions designed to obtain information at different levels. The complete survey design is shown in Appendix B. The set of questions spanned the following categories:

- *Demographics*: Information such as gender and type of school
- *Teaching background*: Participant's experience in teaching block-based programming. For example, context of teaching, length of exposure, grade/level of students and the types of BBPE used.
- *Assistive tools*: Information about the assistive technologies and platforms used by the students.
- *Challenges and workarounds*: Challenges faced by students as observed by the teachers who teach these students.

4.1.2 Participants

Participants were recruited via mailing lists and (postings in social media groups for teachers of children with visual impairments). Participants had to consent before taking the survey and all questions were optional. No identifying information was requested from participants, except the email address for the the following two reasons. (1) The questionnaire asked respondents to provide their email address if they wanted to be contacted for follow-up studies. (2) Participants could also choose to enter a raffle to win a $50 Amazon gift card by providing their email address (necessary to contact the winner).

A total of 12 people submitted responses to the survey, all residing in the United States. There were 8 women, 3 men and 1 preferred not to answer. All participants were teachers with experience in teaching programming to children with visual impairments. The participants taught in different types of schools (see Table 4.1). All Participants except one had experience with Block-based programming that ranged from less than a year to between 1 and 2 years. The only participant who indicated never having experience with BBPE instead worked with the Quorum programming language, which is a text-based language. This was fine because survey questions had a section dedicated to text-based languages, though our focus in this paper is on BBPE.

Table 4.1: Type of schools in which TVI participants worked

School Type	n
Public School	1
Private School	2
State School for the blind/visually impaired	7
Consultant that works for state agency serving Student with visual impairment	1
State services for the blind/visually impaired related to K-12	1

4.1.3 Results

4.1.3.1 Data Analysis

All the received survey responses were anonymized by removing any identifying information. Two researchers analyzed and coded the open-ended questions. The questions were first analyzed using an initial set of codes based on the literature and one researcher's experience in the field. During the analysis, more codes were revealed from the responses and the two researchers frequently met to discuss and to agree on the codes. Multiple coding passes were run to come up with the final codes that will be discussed in the next section.

4.1.3.2 Findings

In this section we discuss the findings from the survey responses. We start by answering Research Question 1, which was based on the initial set of multiple-choice questions designed to obtain information about the context, technologies, types of BBPEs used in teaching and the educational level or grades of the students taught by the teachers who took part in the survey. Following this will be an analysis and discussion of the open-ended questions that were designed to enquire about and elicit any challenges that the students face while learning how to code on BBPEs.

4.1.3.2.1 Context of Teaching Programming

One of the questions required participants to indicate the context in which they taught programming to children with visual impairments. Five out of the twelve respondents indicated teaching programming in schools. Two of the participants indicated teaching coding to students with visual impairments through short term programs of one to two days or even a week. These are usually in the form of workshops designed to motivate, inspire and increase participation in programming and computing in general to people with visual impairments [102] [98]. One respondent indicated teaching on the Hour of Code activity [103] and in school on an individual and short-term basis. These responses show that people with visual impairments learn programming in a mixed variety of context and settings.

When asked to select the class or level of their students from a list which included Kindergarten-2nd grade, 3rd-5th grade, Middle school and High school, six out of the seven who responded to this question indicated teaching students in middle and high school. Only two participants reported teaching in 3rd-5th grade. Though the number of respondents is relatively small, we can infer here that it is common to find learners with visual impairments in middle and high school who are learning programming.

4.1.3.2.2 Tools and Technologies Used

As part of the survey, we also sought to learn about the types of assistive technologies, programming languages and platforms that teachers used in their classes. When asked about the type of assistive technology used in teaching programming, nine out of twelve participants indicated using both screen readers and magnification software in teaching. Only two participants indicated using braille displays. This shows that screen readers and magnification software are among the primary assistive tools used by novices with visual impairments in learning programming. Therefore, focusing on designing BBPE to be fully accessible through screen readers will have a wider effect on increasing participation in programming by learners with visual impairments.

Another question required participants to select the type of platform used in teaching programming. Teachers had to select among tablet, phone, and desktop or laptop. Six out of the seven who responded to this question indicated teaching programming on tablets. All those who selected tablets indicated using iPads and

VoiceOver in their classes. The most common reason being that iPad comes with Swift Playgrounds[104] which is easily accessible via VoiceOver unlike other common BBPEs like Scratch or Snap!.

4.1.3.2.3 Types of BBPE Used in Teaching

When asked to list the types of block-based programming environments used in teaching, surprisingly, Swift Playground was the only environment indicated by participants who responded to this question. As indicated before, it might be that Swift playground is what they find to be most conveniently accessible. This is supported by the following report from one participant:

> ...we have found that most workarounds for programs such as Blockly or Code Snaps seem cumbersome and we have had the most luck with Swift Playgrounds on the iPad...

Another cited reason is that there are ready-made apps for easily integrating Swift Playgrounds with external devices such as Dash robots and Mindstorms robots. This reason can be compelling since robotic activities are usually at the center of most outreach programs designed to attract people with visual impairments into programming and computing in general [98]. Teachers might not also be aware of other accessible BBPEs such as Blocks4All [4] and Accessible Blockly [36]. If this is the case, it implies efforts are needed towards raising awareness about the availability of these accessible block-based languages. The fact that only Swift Playgrounds is common among learners calls for a need for more diverse and selection of tools to include wider user platforms and use. This finding seems to reveal that our target community is still unaware of the different accessible tools that are designed and developed to assist them. For example, the authors in [101] found that only 4 out of 15 participants in their study indicated using IDEs and editors that were purposely designed with blind programmers in mind from the ground up.

4.1.3.2.4 Challenges Faced by Students with Visual Impairments on BBPEs

To answer Research Question 2, we asked a series of open-ended questions that were designed to elicit the challenges that learners with visual impairments face on current BBPEs. A number of challenges emerged from our coding scheme which we shall elaborate below. The open-ended questions were formulated based on the different types of actions that can be accomplished on a typical BBPE. In a BBPE, students construct programs by selecting blocks from a toolbox which contains a set of predefined blocks. A selected block on the toolbox is then dragged and dropped on a workspace where the program is constructed. Blocks can also be dragged around on the workspace in order to modify an existing program. Some blocks have fields on them that require a user to type in text. For example, an integer block that contains a field to hold the number it represents. Accordingly, four major questions were written in order to inquire about the challenges that students with visual impairment encounter while performing the following actions on a BBPE.

1. Identifying blocks from the toolbox
2. Moving blocks from the toolbox to the workspace
3. Manipulating blocks on the workspace
4. Understanding the relationship between blocks or the logical flow of blocks in an existing program

These actions can seem trivial for a sighted user who uses a mouse to perform drag-and-drop operations while building a program. However, for a user who relies on a screen reader, a number of challenges can occur as reported in this study.

After analyzing the responses, we noticed that most of the reported challenges cuts across all of the four major actions stated above. Therefore, our analysis will be grouped as per the identified themes and not by the four major actions described above. However, we will highlight wherever a challenge affects a particular action more than the others.

4.1.3.2.4.1 Inadequate Screen Reader Skills

Inadequate knowledge about using a screen reader was one of the most common themes that emerged from participants' responses. Because the screen reader is the primary medium of output from a computer for most people with visual impairments, being deficient on how it works can be a significant barrier to correctly interacting and writing programs in a BBPE. This issue was present as a barrier to performing all of the four major actions mentioned earlier. It was mostly associated with the action of identifying blocks from the toolbox. To select a block from a toolbox a user has to browse through a list of blocks sometimes arranged in a hierarchical list of categories. Unlike a sighted user who can quickly identify a block based on its color and shape, a screen reader user needs to listen to an audio description of each block usually in a sequential manner before choosing a block. This can seem daunting and difficult for someone with less experience using a screen reader. For example, one participant clearly reported:

> Most difficulties were with screen readers and dexterity issues. Swift Playgrounds works quite well in identifying blocks using VoiceOver on the iPad.

Because the screen reader is indispensable for people with visual impairments especially those who are blind to interact with a computer, providing adequate training on how to operate it can help reduce some of the barriers in learning block-based programming.

4.1.3.2.4.2 Dexterity Issues

The ability to correctly perform the different gestures that represents different functionalities on a BBPE greatly affects students' capacity to easily build programs. It was reported that students mostly lacked in the gesturing skills necessary for interacting with Swift Playgrounds on iPad. This deficiency in dexterity influenced their ability to accurately perform each of the first three major actions mentioned earlier i.e identifying blocks from the toolbox, moving blocks and manipulating blocks on the workspace. This is further supported by the following response from one participant.

> Students have less experience than we expected on the gestures for using an iPad with Voiceover. We had to spend considerable time with swiping and tapping instruction with some students.

It is only when learning to code takes place on touch screen that dexterity becomes a problem for novices. This will not constitute a barrier if learning takes place on a laptop or desktop computer with a keyboard, although the students will need to learn basic keyboarding skill. In one of our follow-up questions, we sought to understand whether teachers had a preference between teaching block-based programming on touch screen or on desktop with a keyboard. One respondent to this question indicated that they had no preference of one over the other and instead preferred that students should learn both. There is no enough data here to draw a line between learning block-based programming by people with visual impairment on either platform and we leave this as an avenue for future research.

These first two barriers (Knowledge of screen reader and dexterity issues) tells us that sometimes the limitations might just be due students' inexperience in using the technology. We believe these barriers can be addressed through more training, consistent and regular practice by the students. Thus, sufficient training on technology usage is an important step in reducing the accessibility barriers to block-based programming and programming in general.

4.1.3.2.4.3 Editing Programs

Editing an existing program to solve an error once a mistake is realized seems to be the most difficult task students with visual impairments face while working with blocks on the workspace. This challenge is only associated with the third action among the actions mentioned above, i.e. manipulating blocks on the workspace. Out of the 7 participants who responded to this question, 5 highlighted this as a challenge. One of the participants even stressed that this was the largest difficulty as evident below:

> The largest difficulty was editing once a mistake was found. Editing using VoiceOver was difficult for some students–i.e. cut, copy, paste using the rotor.

Further analysis of the participant responses revealed that this difficulty is partly due to the type of commands associated with editing blocks on Swift Playgrounds. These commands are considered advanced for novice users as they require great gesturing skills to perform. This is supported by the previous quote as well as the following answer from another participant.

> Editing blocks of code was difficult for students due to the more advanced screen reader commands involved in cutting, copying and pasting blocks. Few could use the rotor within VoiceOver to do this efficiently.

The VoiceOver rotor is defined "as a context dependent wheel of commands, with only one command in force at any one time. As you spin the wheel, a new command name is spoken and made active" [105]. Using the rotor to execute commands require a series of hand gestures on the touchscreen which can seem difficult for novices.

It should be noted here that the teachers in their responses highlighted that finding the error itself does not constitute a problem for learners with visual impairments. The students are able to understand the logic of their programs and can readily identify errors in logic. However, the difficulty here is executing the commands necessary to effectively move blocks around and solve the identified error. This is supported by the selected response below.

> When the blocks of code were run and errors in logic realized, they were able to see the errors, but the issues with editing the blocks (as mentioned in number 3) were more often the barrier to rewriting correct code blocks.

To our knowledge, this is the first study that identifies editing existing programs as a challenge for learners with visual impairments in the context of block-based programming environments. Previous work has not talked about this and this opens a new path to research because learning how to fix the errors that we identify in our programs is an important part of programming. In [4] the authors discuss building programs and conveying program structure on the workspace, but do not mention about editing existing programs on the workspace.

We also highlight here that all participants to this study indicated teaching programming on touchscreen. Thus, it is not known if editing programs as discussed here as a challenge are present on systems where users primarily rely on the keyboard for interaction.

4.1.3.2.4.4 Navigation Challenges

Navigation also emerged as a challenge and based on our analysis, they were of two types. The first one is navigation with respect to the coding environment. This can be defined as the ability to successfully go from one component of the BBPE to another. For example, moving from the toolbox to the workspace. Right now, it is not clear whether this is a challenge on its own, but as elaborated under section 4.1.3.2.4.5 (ineffective tool feedback and Orientation), this can be due to the limited feedback from the system that affects the user’s ability to easily locate and navigate to different components on the screen. It can also be due to the learner’s screen reading skills as explicitly stated by one respondent.

Navigation with screen reader, they need to refine their screen reading skills.

Navigation in programming as a challenge to people with visual impairments have been reported before and mostly in the context of text-based programming [16], [25], [106]. This is usually referred to as code navigation, which is the second type of navigation challenge that emerged from our study. In the context of BBL, code navigation involves going through the blocks of a program usually block by block. There are various reasons for navigating a program such as to read and understand the program as a whole, to obtain specific information, to locate an insertion point or to go to an error location. For a screen reader user, this means shifting focus from one block to another while paying attention to information about the structure of the program such as changes in nesting levels.

It emerged that students have difficulties with navigation on the workspace as explicitly stated by one of the respondents as shown below.

> Challenge with navigating the workspace and moving the blocks to the correct location to solve an error.

Challenges due to code navigation is a research topic by itself and has been shown to be associated with different number of coding activities [16]. Though our study was not designed to capture code navigation challenges in detail, but the general challenges faced by students in BBPE, the fact that it emerged from our analysis tells us that it is an area of concern that has to be investigated in the context of block-based programming.

4.1.3.2.4.5 Ineffective Tool Feedback

It could also be drawn from the pool of responses that the coding environment provided limited feedback to the students as they interacted with the system. This insufficient feedback often leads to alignment problems. For example, teachers reported students having difficulties knowing their precise location on the touchscreen with respect to the components of the BBPE. Block-based programming environments provide a lot of visual feedback to sighted users as they interact with the system. Sighted users can easily locate the placement of the toolbox and the workspace or a specific connection between two blocks just by looking.

They also receive continuous visual feedback as they move blocks from one location to another until the final spot. This makes it easy for sighted novices to easily orient themselves within the coding environment. However, things are different for novices with visual impairments who do not have access to this rich visual feedback. The current alternative feedback mechanism, mostly sound and audio in this case seems to be limited or ineffective. This lack of adequate feedback from the system affects students' ability in carrying out the first three major actions stated above.

Because there is insufficient feedback from the system, students have difficulties navigating around the BBPE. This affected the students' ability to easily locate the toolbox and consequently select blocks as reported in the study. Analysis of the responses also revealed that students faced difficulties moving blocks from the toolbox to the correct spot on the workspace due to the lack of continuous feedback. Sighted users always have that continuous visual feedback as they move blocks from the toolbox to the workspace. This is not the case for users who rely on the screen reader. Lastly, similar to moving blocks from the toolbox, the system did not provide enough feedback to the students while they manipulated blocks on the workspace.

We also observed that ineffective tool feedback affects the students' ability to select the right command for a given action. It was reported that students often confused the correct command or shortcut associated with performing a particular action on the screen. For example, one teacher reported the following:

> Student understanding of what is actually happening in the coding environment vs. using a shortcut to make the correct choice.

As illustrated above, it could be inferred from the responses that users who rely on screen readers do not receive sufficient feedback from the tools or system to easily know what is happening on the coding environment and thus act accordingly. Though confusion of commands can also be associated with the fact that students are mostly novices, we believe insufficient feedback from the system plays a role in amplifying this difficulty.

4.1.3.2.4.6 Software Glitches

Occasional software glitches emerged as a frequent obstacle to the students while they interacted with the programming environment. Though this is not a challenge from the students' perspective per se, it showed up a lot in the pool of responses that we thought it is worth mentioning. Software glitches was mentioned by four out of the eight participants who responded to the open-ended questions, and it appeared as a challenge across all the four major actions stated above. It was recurrent in the responses of two of the four participants who stated it as a challenge. This shows that it is a common technical issue that teachers see as an obstacle to the successful learning of programming by learners with visual impairments. Example responses highlighting this issue follow.

> ...gaps in knowledge of screen reader, occasional software glitches in the programming and difficulties editing blocks of text with cut, copy, paste within the screen reader environment…

> Occasionally there were glitches in the screen reader that did not work fluidly within the Swift Playgrounds environment.

As illustrated above, this was explicitly cited by teachers as a challenge faced by students with visual impairments. However, this might be an issue with Swift Playgrounds and VoiceOver because all the participants to the study indicated teaching programming on Swift Playgrounds on iPad using VoiceOver. Thus, this cannot be generalizable to other BBPEs. However, occasional technical failures in the software system can make learning how to program challenging for learners with visual impairments.

4.1.3.2.4.7 Inadequate Accessible Learning Material

Another common theme that emerged was the absence of materials in accessible form. Without enough accessible material, learning block-based programming becomes a challenge for novices with visual impairments. This is one of the reasons why teachers in our survey reported using only Swift Playgrounds to teach programming to students with visual impairments. Swift Playgrounds provides accessible tactile graphics for each coding puzzle that can be used by students with low vision or without vision to have a better understanding of what is happening in the coding environment. Participants to the survey reported that these maps are very useful in supporting students with visual impairments while they learn coding on Swift Playgrounds. In the absence of these accessible instructional support, it will be difficult for students to fully have a grasp about what is happening within the coding environment. This is evident from the following selected responses by two participants in our study.

> It helps that we have tactile maps of the different puzzles that are found on screen so they can have a tangle [sic – meant tangible] understanding of what needs to be done.
>
> The tactile maps of the Worlds that can be embossed from Swift Playgrounds were essential to helping our students who were braille users to understand the layout of the worlds and plan and correct their blocks of code.

Because block-based programming environments communicate much of the information in visual form to its users, having tangible versions of samples as support material can help students with visual impairment better understand what is happening within coding environment. Absence of accessible information is one of the most commonly cited reasons as to why there is less participation in computing by people with visual impairments [5]. A number of efforts are going on to provide accessible curriculum and teaching materials in order to address this issue [5], [9].

4.1.4 Conclusion and Future Work

We have presented a study designed to investigate the challenges faced by students with visual impairments on block-based programming environments while they learn how to code. Our survey study was addressed at the teachers of students with visual impairments partly because they work with different students on regular basis and can easily spot challenges and also because teachers have often been requested in the literature to provide elaborate explanation about the difficulties faced by their students. Firstly, our findings indicated that the few accessible block-based programming environments that have been developed in recent years are not common in the educational space. Swift Playgrounds is the only accessible block-based programming environment that is cited as being used by teachers to teach children with visual impairments. Other mainstream BBPEs such as Scratch, Snap! and Blockly are found to be inaccessible with screen readers.

Secondly, the challenges identified through our study show that these difficulties are technical rather than conceptual. The study revealed that students are usually unprepared in terms of the skills required to easily interact with the technology. These includes inadequate skills in using a screen reader and dexterity issues in regard to the gestures required to interact with Swift Playgrounds on iPad. Our findings also indicated that challenges in editing programs is one of the major difficulties faced by the students. Though few research work has studied the challenges faced by students with visual impairments on block-based programming languages, this is the first time that challenges in editing programs is reported as a barrier in block-based programming for students with visual impairments. Students were reported to have difficulties with editing programs especially because of the advanced gestures associated with the commands required for this action.

Our study also indicated that navigation was a challenge for students. The teachers reported that students had difficulties navigating the BBPE as a whole as well as difficulties with navigating code on the workspace. We also found out that students received limited or ineffective feedback to guide their actions successfully on the system. Other difficulties resulted from occasional software failures which the teachers perceived as a barrier for students with visual impairments. Lastly, the study revealed that inadequate accessible learning material is also a barrier to students with visual impairments learning how to code on block-based programming environments. This corroborates with previous findings that inaccessible learning materials is a main reason for low participation of people with visual impairment in computing in general.

There are however some limitations to our study. Firstly, all the participants reported teaching programming using Swift Playgrounds on iPad. Therefore, the challenges reported in this study mostly reflect what happens on touchscreen-oriented block-based programming environments. We are still unsure whether these are generalizable to environments where interaction primarily takes place via a keyboard. Secondly, because only Swift Playgrounds have been cited in the study, some of the challenges such as the occasional software glitches may not apply to other BBPEs. Another limitation of the study is that the account is based solely upon the teachers' observations. It will be interesting to have an account from the students' perspective as well. Lastly, the geographical location and the number of participants to the study is relatively small to arrive at universal conclusions.

Despite these limitations, the study highlights important information for researchers and developers who are currently working to create accessible block-based programming environments or trying to enable accessibility into existing ones. As mentioned in the introduction, the study was design to inform our ongoing larger effort into the design of accessible block-based programming environments for children with visual impairments. Areas of future work with respect to this study include: conducting an elaborate investigation of the challenges related to code editing and code navigation that were identified in this study, investigate whether these findings are generalizable to systems where users primarily interact via a keyboard and lastly conducting focused group studies with teachers in order to understand how they usually tackle these challenges when encountered.

4.2 Interview Study with Teachers of Students with Visual Impairments

Research on the accessibility of block-based programming languages (BBLs) for students with visual impairments originated within the past six years [14]. These research efforts have led to the development of a few accessible block-based programming environments (BBPEs) such as Blocks4All [4] or hybrid environments such as Swift Playgrounds [107]. Other researchers have chosen a different path by developing accessible tangible block-based programming toolkits for students with visual impairments [76][77][71]. Six years later, the literature says little about the learning processes of students with visual impairments on these systems.

While some work has been done to explore the learning process of students with visual impairments in computer science [108], existing scholarship looks at computer programming in general without a focus on block-based programming or hybrid environments. Other researchers work with teachers to explore the coding practices of students with visual impairments in order to inform the design of accessible programming environments [91]. There is a gap of knowledge between these types of isolated evaluations or studies and the reality of the instructional usage of block-based programming and hybrid environments in schools or in outreach activities. The survey study we conducted with TVIs and reported in the previous section was an attempt to fill out this gap [35]. The results of the survey study showed that none of the mainstream BBPEs was being used by TVIs to teach students with visual impairments. Rather TVIs used Swift Playgrounds, a hybrid environment that runs on the iPad. Additionally, they reported that students face challenges on Swift Playgrounds including difficulties navigating and editing code, and dexterity issues associated with the use of VoiceOver, a screen reader on the iPad. These findings however leave so many questions unanswered. For example, it is not clearly known why the TVIs choose the hybrid environment and the reasons behind the reported challenges are unexplained.

To fill this gap of knowledge about the reality of the instructional usage of block-based programming and hybrid environments in schools we asked the following research questions.

RQ1. What are the causes of the accessibility challenges reported in survey study with TVIs? How does the experience differ with the level of students' vision?

RQ2. What do teachers use in the absence of accessible BBLs? Why do teachers prefer Swift Playgrounds as reported in the survey with TVIs?

To answer these research questions, we conducted a qualitative interview study with twelve teachers of students with visual impairments (TVIs). First, the results of the study confirmed the challenges reported by the TVIs in the survey study. In addition to identifying these challenges, we found that teachers employ a trial-and-error like strategy to find accessible solutions around the inaccessibility of common block-based programming environments. Issues with the curriculum and/or syllabus was also highlighted by the teachers confirming findings from prior research [108]. We also found that the current situation imposed by the COVID-19 pandemic during which classes are conducted remotely was one of the reasons why teachers transitioned to using Swift Playgrounds. Teachers expressed an overwhelming interest in having a fully accessible block-based programming environment and a curriculum tailored to the needs and level of their students.

In this section, we provide: (1) an in-depth analysis of the causes of the accessibility barriers and the issues impacting the instructional usage of block-based programming and hybrid environments by students with visual impairments through the lens of TVIs; (2) An analysis of how teachers deal with the lack of fully accessible block-based programming environments; and (3) suggestions on opportunities to improve the learning process for student with visual impairments learning how to code and on designing accessible on-screen block-based programming languages.

4.2.1 Methods

We conducted a qualitative study consisting of semi-structured interviews with teachers of students with visual impairments (TVIs). The goal of this study was to better understand the reality of the instructional usage of block-based programming and hybrid environments across the United States of America (USA), and to answer questions that automatically arose from the previous survey study with TVIs, and how TVIs deal with these challenges. This study was approved by the IRB.

4.2.2 Participants

Twelve teachers of students with visual impairments (TVIs) took part in the interview study. Five participants identified themselves as male and 7 participants identified themselves as female. All participants resided in the United States. To take part in the study, you had to be a TVI with experience in teaching programming or computer sciences courses to K-12 students with visual impairments. Preference was given to people who had experience teaching block-based programming or a hybrid. Participants were recruited via snowball sampling and posting in social media groups and listservs for TVIs. Each participant was compensated with a $50 Amazon gift card after the interview session. Table 4.2 shows the data about the participants.

Table 4.2: Overview of interview study participants, grades taught and years of experience

ID	Gender	Grades Taught	Yrs Teaching Exp	Comments
P1	F	K-12	13	
P2	F	3rd, 5th and above	5	
P3	M	9th and 12th	15	
P4	F	9th and 10th	18	
P5	F	9th to 12th	17	
P6	F	K-12	4.5	
P7	M	3rd to 12th	25	
P8	M	7th to 12th	30	Identified as a blind software engineer teaching homeschooled blind students
P9	M	2nd to 12th	10	
P10	F	6th to 12th	16	
P11	M	4th to 12th	5	Identified as blind
P12	F	9th to 12th	15	

Semi-structured interview questions were designed to gather information about the participants background and experience in teaching programming to K-12 students, the tools and technologies used to teach students, perceived challenges and workarounds to address the challenges if any, the perceived difference with the students' level of vision and finally suggestions on making block-based programming fully accessible and functional for students with visual impairments. The complete interview questions are shown in Appendix C.

Interviews were conducted remotely via Zoom. Participants each signed an informed consent form before joining the session. Each session typically lasted 45 to 60 minutes and participants had the option to opt out of the session at any time. All twelve participants completed the full session of the interviews. The Stephanie Ludi and I, together, conducted the interview for P1 and P2. The remaining interviews were conducted by the me. Participants P1 and P2 requested to be interviewed together in the same session. P8 and P11 stated that they were blind. P8 reported he was a software engineer teaching homeschooled blind students as part time. Data was collected and transcribed in real-time. Participants responses to the questions were written down as they were being interviewed.

4.2.3 Analysis

All the transcripts from the interviews were cleaned after each interview session. Thematic analysis was used to analyze the transcripts in multiple phases. Using the affinity diagramming technique, we went through each transcript highlighting important concepts or phenomena. Each highlighted excerpt was assigned a code. We met weekly to discuss the codes. Excerpts tagged with the same code were grouped together. This first phase produced 37 codes. We met with Stephanie Ludi to discuss the codes and associated excerpts. This led to the elimination of 9 codes. Of these 9 codes, 6 were merged with other codes and the rest were discarded. Another phase of inductive coding led to the categorization of the remaining 28 codes into 9 high level themes including students' background, teachers CS background, syllabus/lesson plans, reason for using swift Playgrounds among others. These themes discussed in the next section highlight issues and phenomena relating to the practice of teaching programming to students with visual impairments as identified by TVIs.

4.2.4 Findings

In this section we discuss the findings structured according to the two research questions introduced earlier. Analysis of our data revealed and confirmed all the prior challenges reported in the previous survey study including *limited screen reader skills, dexterity issues, difficulties editing programs, navigation challenges, poor system feedback* and *the lack of accessible learning materials*. In addition to confirming these challenges, the data from the interview with TVIs offers insights into the causes of these barriers and on issues surrounding the learning process of students with visual impairments learning how to code in educational settings. Before addressing the research questions, we present the programming environments used by TVIs to teach students in the next subsection.

4.2.4.1 Programming Environments Used by TVIs

Table 4.3 shows the block-based programming environments, hybrids and tangible alternatives participants reported using to teach students with visual impairments how to code. In addition to these systems, other participants reported using text-based languages and environments such as Java, Python, JavaScript, HTML, OpenScad and CodeHS. However, there was not enough data to talk about these text-based languages because they were mostly cited by only one participant. We also wanted to maintain a consistent scope and focus on block-based programming and hybrid environments. We included Quorum, an accessible text-based language in this table because many participants talked about it and because one popular accessible CS curriculum uses Quorum as the main programming language [109].

Table 4.3: Programming languages and environments used by study participants

Coding Environment	Type	Accessibility	Participants who have used
Blocks4All	Block-based	Accessible	P1, P2
Scratch	Block-based	Not accessible	P5, P8, P11
Swift Playgrounds	Hybrid block/text	Accessible	P1, P2, P3, P4, P6, P7, P8, P9, P11
CodeQuest	Block-based like	Accessible	P1, P2, P6
Edison	Block-based	Not accessible	P12
Quorum	Text-base	Accessible	P1, P2, P3, P6, P8, P11
Code.org	Mixture of block and text	Not accessible	P1, P2, P3, P6, P11
Code Jumper	Tangible	Accessible	P1, P6
CodeSnaps	Tangible block-based	Not accessible	P6, P7
Snap Circuits with Snapino	Tangible	Partially accessible	P6

4.2.4.2 Causes of Accessibility Barriers and Issues Surrounding the Effective use BBLs or Hybrid Environments

4.2.4.2.1 Student Background and/or Experience with Technology

Almost all participants (n=9) agreed that the student background or experience with technology was one of the primary reasons they faced challenges while learning how to code. This reason applies when students are learning coding on block-based or text-based programming environments. The codes that emerged under this theme include *limited prior exposure to basic computing skills* and *limited orientation and mobility (O&M) skills.*

Eight participants explained that students have limited prior exposure to basic computing skills before they start learning how to code. This lack of experience usually translates into dexterity issues and other challenges identified in the previous survey study. These basic skills include screen reader skills for screen reader users, swiping, tapping and drag-and-drop skills for those who use tablets, and keyboarding skills. P1 who used Swift Playgrounds with students stated:

> …students don't have the VoiceOver skills. I had to introduce swiping and tapping skills to the students.

VoiceOver is a screen reader that comes with an iPad. P2 highlighted a similar issue when working with students on Quorum by noting that:

> With Quorum you can do a lot, but the problem is that the braille users don't have the keyboarding skills…

This finding is consistent with prior work [108]. However, our qualitative data offers more insight about how this lack of precursor skills differs among students, how TVIs usually mitigate this issue and how it affects the learning process.

Half of the participants (n=6) further explained that students who had more experience with these skills were better and faster in completing tasks than those who did not as illustrated by this quote from P3.

Those experienced with the keyboard and screen reader where faster and better.

P8 thinks that this lack of experience in the basic skills and tools is the main issue or source of challenges for students with visual impairments learning how to code. P8 explained in the following words.

> One of the big things people don't understand is that accessible programming is not the issue. Main issue is students don't know their tools. I end up teaching the tools more than teaching programming. Once they learn their tool, they can do pretty much any task that you show them. Make sure kids are learning good keyboarding keys…

These data suggest that ensuring students have good familiarity with their tools and basic computing skills before they start learning how to code is an important step in addressing the challenges students with visual impairments face in the learning process. In fact, this is one of the approaches teachers take with the students as illustrated in the above quotes by P1 and P8. This, however, influences the learning process. It tends to slow down the learning process because of the time spent to introduce and teach these skills as illustrated by P7:

> Students lack typing skills when they come to class. I teach how to navigate using the keyboard, … Learning tools makes it slower. If they don't know what hashtag is, we have to stop and teach.

An option to help students learn the basic skills without slowing down the learning process once they start coding might be to introduce these skills earlier as suggested by P1 and P2.

> We are trying to push that keyboarding skills be introduced early.

Some participants (n=3) believe that this deficiency in basic computing skills is because students have less exposure to learn or practice the skills from home. P8 stated how he encourages parents to help with this.

> I also encourage parents to allow students use the computer like playing games to help them improve their skills.

The second code that emerged under this theme was *limited O&M skills*. Half of the participants explained that students, especially blind students were deficient in O&M skills. This in turn made it challenging for them to understand what was happening in the coding environment, what their code was doing and the behavior of their program after execution. To understand why this deficiency causes challenges for students, we need to look at the type of programming activities that the teachers use with the students. Teachers explained that they prefer robotics activities or tangible programming with students with visual impairments because their output is easily accessible, and students find them to be fun. This corroborates prior research [91][4]. In addition to robotics activities, the output in some block-based programming environments usually involves moving an avatar in some sort of grid space. An example is the puzzles in Swift Playgrounds. Being successful at this type of activities requires some knowledge of spatial navigation. The following quote by P11 demonstrates how this deficiency in O&M skills affects students learning how to code on Swift Playgrounds, an accessible hybrid environment:

> With Swift Playgrounds, students have a hard time with orientation like left, right, north and south even when they understand the technology.

Like with the basic computing skills, participants also explained how they go about to mitigate this deficiency in O&M skills that affects students' ability to learn and practice coding. Essentially it involves having the students mimic what their code does with their bodies before implementing the code. P1 and P2 reported:

> We introduce concepts to students by having them execute instructions with their bodies first. For example, turning right vs turning left.

P7 also finds this to be useful as illustrated by the following quote:

> Once they learn the concepts, they can follow and say what the program is about. One kid will be the robot and the other kid will look at the code and they walk the code and mimic the robot together. I use this method to teach. Kids will then have an idea of what their code is doing based on the instructions that is executed by the other kid say go left, go right.

It should be noted that according to the data, students who have good O&M skills perform better than those without. The participants also reported that deficiency in O&M skills affected blind students the most. Pires et al. [91] investigates how to make spatial activities in programming environments accessible to students with visual impairments.

4.2.4.2.2 Teachers Background/Limited CS Knowledge

One of the themes that emerged from our qualitative data suggests that the teacher's domain knowledge affects how well the students learn coding irrespective of the type of coding environment used. This has to do with the *teacher's inexperience in programming* and/or *inexperience in the use of assistive technologies.* Data from five participants suggests the learning process is difficult and sometimes frustrating to students when the teacher assigned to teach has a poor CS background. P4 talking about the issues with teaching students with visual impairments on Swift Playgrounds recalled a moment when someone who lacked programming skills was assigned to teach students:

> One of the issues is having people who don't understand what programming is before working with students. The student was frustrated because the person who took over didn't understand programming … They have same challenges that other people face when they are being helped by someone who don't understand the program. Their teachers are not well trained.

P10 took an example of herself who was assigned to teach programming to blind kids while she had a poor CS background. The lack of CS background by some teachers can prevent students from learning coding on an accessible coding environment when one is available. P1 and P2 explained why Quorum being an accessible programming language is not considered as an alternative in schools by some teachers because of their limited CS knowledge.

> Quorum isn't easy for teachers to work on by themselves. Teachers don't know the syntax, so it is not a solution in schools… They don't have a CS background. … It is overwhelming for teachers to consider alternative language like quorum because they lack a CS background.

In other instances, teachers have a good CS background but are *deficient in the use of assistive technologies*. This is mostly the case when the students with visual impairments seat in mainstream settings with their sighted peers. Some participants (n=3) reported that part of their work involves seating with blind and visually impaired kids to help them access coding environments and coding materials with their assistive technologies. And in the process, they also help the students with the coding tasks when necessary.

Our qualitative data also suggests how teachers try to manage themselves out of this situation through self-study. Eight participants mentioned they learn programming and CS concepts on their own to help their students. These results in some teachers having to do extra work as illustrated by this quote from P3: “I had to do extra work. I learned the Quorum in addition to Code.org.”

As stated earlier, teachers would *have someone work alongside the students* if the teachers are deficient in the use of assistive technologies. From this qualitative data, it can be inferred that part of facilitating the learning process of coding by students with visual impairments involves training qualified personnel with the right CS background. This is a concern that other researchers have been focusing on as part of broadening participation in computer science to include more people with visual impairments [110].

4.2.4.2.3 Limited or Too Complex Curricula/Lesson Plans

Nine participants also expressed concerns about the curricula or lesson plans that they use to teach students. The data suggests that the *lesson plans are either limited or too complex* for students with visual impairments. This applies to multiple curricula or programming environments that come with lesson plans for kids. Teachers not only find the available materials to be inaccessible, but there is also a lack of material with lesson plans tailored towards the needs and abilities of students with visual impairments. P6 noted:

> There are not many accessible materials. Swift Playgrounds, Sphero, what are the next steps? No clear lesson plans. I teach about 40 lessons. Quorum is too high level and Code Jumper is too basic.

As another example, some teachers (n=3) not only find Code.org to be inaccessible to students with visual impairments, but also that some of the activities do not match the needs and abilities of students with visual impairments. This confirms findings from prior research [110]. Another example from our data is Swift Playgrounds. Although Swift Playgrounds is accessible [35] and comes with well-designed lesson plans and supporting materials such as tactile maps, four participants described the lessons plans as complex for students to follow at some point. P6 who has tried multiple programming environments with her students described Swift Playgrounds as follows:

> Students have more success with Swift Playgrounds. However, Swift Playgrounds get complex quickly. It will be great to have something like Swift Playgrounds junior that is slow pace and simple.

This complexity of lesson plans seems to affect blind students more than students with low vision as illustrated by the following quote from P1 and P2 when talking about Swift Playgrounds:

> [Swift Playgrounds] gets a little complicated as it goes along, especially if you don't have the vision. The level of difficulty increases fast.

On the other hand, some materials are described as *too basic by the teachers*. Examples are the CodeQuest App and Code Jumper which are both accessible to students with visual impairments. P1 and P2 for example reported:

> With CodeQuest app, we move at slower paste, slow progress but only basic concepts are offered. Only loops, no conditional statements.

The absence of an accessible (mainstream) curriculum with activities tailored to the needs of students with visual impairments have teachers trying multiple materials. This not only makes it challenging for the students in the learning process but also introduces extra work for the teachers. There has been an attempt to create an accessible CS curriculum for students with visual impairments using the Quorum programming language [110]. However, as our data suggest, TVIs find this curriculum too advance for the students, or as stated earlier, the limited CS background of some teachers make it difficult for them to consider Quorum as an alternative.

The qualitative interview data also suggests how teachers try to mitigate the challenges that stems from the complexity of the lesson plans. They personally modify the lessons plans by *introducing more scaffolding activities,* or by *providing alternate* *instructions* (n=6) and by *repeating instructions multiple times* (n=2). P7 stated that he designed a custom program to teach kids programming.

> I use a self-made program designed for the kids using physical programming materials.

The data from three participants also suggests that the presence of *instructions in alternative format* such as *the tactile maps* that come with Swift Playgrounds and Code Jumper also helps minimize the issues with the complexity of the lesson plans and activities.

4.2.4.3 How Teachers Compensate with the Lack of Accessible BBLs

All participants in our study expressed a lot of *interests in having accessible BBLs* for their students. They would like their students to learn programming on the same environments with similar activities that sighted kids find appealing and engaging [2]. P1 and P2 reported:

> Ultimately we will prefer to have BBLs because all the schools are using it...Have stuffs that all the other kids are using accessible for all kids.

However, the current shortage of accessible BBLs have teachers looking for alternate ways to compensate for this and enable their students with visual impairments to learn programming in a fun and engaging way. This section describes the themes that emerged when looking at our data from this angle.

4.2.4.3.1 Tangible Blocks used as an Alternative to BBLs

Data from half of the participants (n=6) indicates that they *use tangible programming toolkits as alternatives* to on-screen BBLs. The reason is that student can feel and easily manipulate the tangible blocks or artifacts with their hands. These include toolkits such as Code Jumper [92], an accessible physical programming environment originally developed by Microsoft. P6 who used Code Jumper with her students noted:

> Code Jumper has very well-organized lessons. Good for students with lower cognitive levels … Because it is like toys, is hands-on, actually teaches real CS concepts that students can feel.

Other participants reported using QR printed card blocks from CodeSnaps [111] to have students physically program the Sphero robot. One participant designed and printed 3D blocks which the students assembled physically with their hands before scanning the assembled program with an iPad app (CodeSnaps). These participants find the physical activities to be an easy method for introducing programming concepts to students with visual impairments as suggested by the following quote from P7:

> …the block-based programs are nice and easy introduction for the students. That's why I like the QR printed physical blocks. It takes less time to introduce BVI [Blind and Visually Impaired] students to programming.

These findings provide support for multiple research work in the literature that have looked at designing accessible tangible programming environments for students with visual impairments as alternatives to the inaccessible on-screen BBLs [71][77][76]. Pires et al. [91] in an exploratory study to design an accessible programming environment for students with visual impairments found that tangible programming and physical activities could enable a faster learning process and improve concept acquisition by students with visual impairments. However, situations like the one imposed by the COVID-19 pandemic where teachers have to teach remotely do not favor the use of tangible programming environments. Multiple participants stated *interrupting the use of tangible programming or physical activities because classes had gone remote due to the lookdown imposed by the pandemic.* Teaching remotely made it impractical for teachers to use tangibles because it would be difficult to readily assist their students on the physical activities. P1 and P2 reported:

> One TVI brought Code Jumper and worked one-on-one with students. The TVI suggested having Code Jumper for everyone. But it is not good given the pandemic since it is tangible.

4.2.4.3.2 Programming Environments with Haptic Manipulatives (robots) and/or Sound Feedback are Attractive

Qualitative data from seven participants suggests that TVIs *prefer programming environments or toolkits that involves programming robots*. Because robots can be touched and/or heard they find them easily accessible and engaging for their students, especially the blind students. For example, P12 reported:

> Totally blind kids love hearing the robots move. When they start, they have to understand that you have to drag the blocks.

The feedback from the robot is a great way for students to feel and appreciate the output of their program. P1 and P2 talking about Blocks4All that comes with the robot reported:

> Students used the iPad to control the DASH robot … You have great feedback using the robot … They used DASH with Blocks4All. Kids are relatively independent using DASH with Blocks4All. It is really great.

This finding confirms previous findings by Pires et al. [91] who worked with TVIs to explore accessible programming environments for students with visual impairments. Robotic activities in general have been found to be fun and engaging for students with visual impairments [98]. Our data also suggests teachers sometimes need to do some modifications to the activities to augment the feedback from the robot. P6 for example explained how she had to place plastic cups along the path followed by the Sphero robot. When the robot knocks down a cup on its path, the resulting sound served as feedback to students who are totally blind. Pires et al. [91] suggested that designers should incorporate more sound feedback and feedforward mechanisms on robots in order to increase the amount of information received by students during the activities.

Our qualitative data also suggests that second to environments with robotic activities, TVIs also *prefer virtual environments that provide sufficient auditory feedback about the output of a program*. This is the case with environments that involve controlling a virtual avatar. An approach followed by Quorum and Swift Playgrounds where auditory feedback is given about the on-screen avatar. P11 while discussing Swift Playgrounds noted:

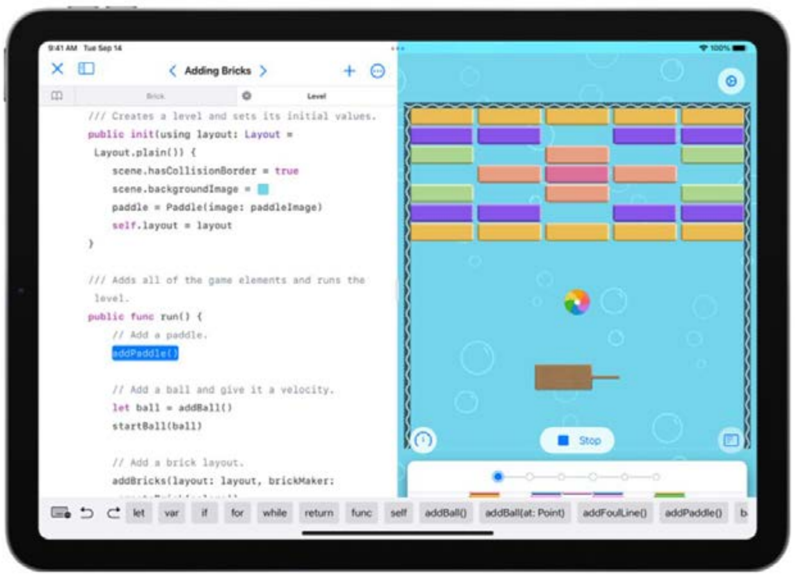
> With Swift Playgrounds you get audible feedback about the character moving. With other environments you don't get any feedback.

P3 also praised the auditory feedback provided by Quorum:

> They [students] got enough feedback. Auditory feedback describes the turtle moving. Quorum tells them what the turtle is doing.

4.2.4.4 Swift Playgrounds

A subset of questions in our semi-structured interviews focused on Swift Playgrounds [107]. Swift Playgrounds (SP) is a hybrid of block-based and text-based programming developed by Apple. Figure 4.1 is a screenshot of Swift Playgrounds with a toolbox of predefined blocks at the bottom and a text editor on the left.



```
/// Creates a level and sets its initial values.
public init(using layout: Layout =
 Layout.plain()) {
    scene.hasCollisionBorder = true
    scene.backgroundImage = 
    paddle = Paddle(image: paddleImage)
    self.layout = layout
}

/// Adds all of the game elements and runs the
 level.
public func run() {
    // Add a paddle.
    addPaddle()

    // Add a ball and give it a velocity.
    let ball = addBall()
    startBall(ball)

    // Add a brick layout.
    addBricks(layout: layout, brickMaker:
```

Figure 4.1: Swift Playground coding environment [107]

The initial survey study with TVIs revealed that Swift Playgrounds was the only accessible hybrid block-based programming environments used by TVIs. However, the data from the survey failed to give insights about the preference of Swift Playgrounds by TVIs. In addition, the TVIs in the survey reported some challenges related to the use of

Swift Playgrounds with little insights about the causes of those challenges. This section tries to fill this gap of knowledge by discussing reasons why teachers prefer Swift Playgrounds and by elaborating among others why students find editing an existing program challenging on Swift Playgrounds.

Analysis of our qualitative data reveals four reasons why TVIs choose to use Swift Playgrounds with their students. The first reason is that *Swift Playgrounds is the only BBL (hybrid) that they find accessible*. When asked why they use Swift Playgrounds, half of the participants (n=6) suggested that SP was what they find accessible as illustrated by this quote from P11:

> I emailed schools for the blind to find stuffs that are accessible for everyone. They suggested Swift Playgrounds as what they find accessible.

The participants elaborated that Swift Playgrounds is well integrated with VoiceOver, a screen reader that ships with the iPad. This facilitates interactions for blind students who have experience using VoiceOver with other applications. In addition to being accessible, the materials that come with Swift Playgrounds lessons are accessible. Three participants praised the tactile maps that come with Swift Playgrounds. These tactile maps allow blind students to have a better understanding of the behavior of Swift puzzles.

The second reason for using Swift Playgrounds was because *classes have gone remote due to the COVID-19 pandemic* (n=2 participants). Two of our participants who were using tangible and physical activities to teach their students before the pandemic indicated they switched to Swift Playgrounds when schools closed, and lessons were being taught remotely. Situations like this further stresses the need of fully accessible on-screen BBLs. P6 reported: "I transitioned to Swift Playgrounds because I'm working remotely."

The third reason for choosing Swift Playgrounds is that the *iPad is easy to handle* (n=3 participants). The participants explained that students have experience with the iPad because they have used other applications on the iPad and are comfortable interacting with Swift Playgrounds. One last reason for using Swift Playgrounds as revealed by our qualitative data is that *Swift Playgrounds comes with well-organized lessons* (n=2 participants). Although as earlier explained, some TVIs find these lesson plans complex for the students at some point in the learning process.

Despite being accessible with well-structured lesson plans, students with visual impairments face a number of challenges on Swift Playgrounds. One of the main challenges the participants reported was *difficulties editing programs.* This confirms the findings from the survey study with the TVIs. The qualitative data from the interview, however, also offer insights into the type of editing tasks and suggests why editing an existing program can be challenging for some students with visual impairments. The interview data revealed that the difficulty with editing has to do with *adding or deleting a specific character on the workspace*. The participants explained that students struggle to add or delete a character on the workspace when they notice a mistake, for example, a wrongly spelled function name.

This challenge with editing has to do with the *VoiceOver rotor* as suggested by the data. The rotor is an advanced option on VoiceOver that requires two fingers to operate [112]. Half of the participants (n=6) explained that using the rotor requires complex interactions which is somewhat difficult for the students to perform. P1 and P2 explained:

> Even in short programs editing is difficult. Even for students with strong VoiceOver skills … Many were not competent with the rotor. It is easy to add commands by swiping, finding commands or to double tap. Removing was difficult. To delete a letter, they struggled.

P4 also reported regarding the rotor:

> …Finger dexterity, they can't get the motion correctly. It is too much for the rotor, not keeping it simple.

The interview data also revealed that the *feedback from Swift Playgrounds is inadequate*, especially when an error occurs on the system. Three participants reported that Swift Playgrounds does not convey enough information to the user about what is happening on the coding environment as they build their programs. This appears to affect blind students the most. P6 noted:

> To add code there is a bit of delay when you press the code and it populates. With vision it is easy to see and wait. But for students with no vision it is difficult. They don't know what is going on especially when they are to add codes consecutively.

4.2.5 Discussion

Research on making programming and programming environments accessible to people with visual impairments have been going on for many years, especially with respect to text-based programming. Recent efforts are focusing on the accessibility of block-based programming environments because of their increase use in K-12 curriculum to introduce novices to programming [14][8][36][4]. In this section, we have presented some insights about the reality of the instructional usage of block-based programming, hybrid environments and tangible alternatives by teachers of students with visual impairments. TVIs still struggle to find accessible materials for their students, existing lesson plans are perceived as complex for the students' level and TVIs find themselves trying different programming environments to facilitate the learning process for their students. In the following subsection we discuss some strategies for mitigating these issues and for improving the overall learning process for students with visual impairments learning how to program.

4.2.5.1 Training TVIs and Teachers in Mainstream Classes

Based on the analysis of our qualitative data, we noticed that TVIs who are assigned to teach students with visual impairments sometimes have a limited CS background. Therefore, we believe the research and educational community can improve the learning process of students with visual impairments by offering training in computer science and programming related courses to TVIs. These training can be in the form of workshops or short-term programs. One participant in our study mentioned that she learned a lot about programming by attending workshops. The participant cited the Quorum conference as a place where she learned about Quorum and decided to introduce that at her school after the conference. Similar activities can be organized to train TVIs on accessible CS tools and programming environments. AccessCSforAll [109] organizes professional development workshops to train teachers on an accessible version of AP computer science principles (CSP) designed for student with disabilities in general. Stefik et al. [110] organized a one week workshop to train TVIs on this accessible version of AP CSP.

On the other hand, our data also suggests that teachers in mainstream classes who have students with visual impairments in their classrooms might also need training on the use of assistive technologies such as screen readers and braille displays. Some TVIs in our study mentioned that they are often called to assist blind or visually impaired students taking programming courses in mainstream classes. This assistance involves helping students with the use of their assistive technology. Teachers in mainstream classes if trained on the basics of assistive technologies can readily assist students with visual impairments sitting in their classes, thus facilitating the learning process for these students. This training can also be in the form or workshops or short-term courses.

4.2.5.2 Teaching the Students Precursor Skills and O&M Skills

The findings also suggests that students with visual impairments face challenges learning how to code because they are deficient in the use of their tools. This include skills such as keyboarding, screen reader usage, and touch screen interactions. Some participants mentioned that students have no difficulties understanding the concepts, the problem is their use of screen readers. Making sure students have a mastery of these skills before they start learning how to code will positively improve their learning experience. Researchers and practitioners can work together to organize special workshops or summer camps to train students on the proper use of their assistive technologies. Many workshops and summer camps have been organized to attract students with visual impairments in computing [98] [5][102]. However, these workshops usually focus on computing activities with less emphasis on the use of assistive technologies.

The results of this study also suggests one other method that can be used to help students acquire the precursor skills. One teacher mentioned that he encourages parents to allow their children use the iPad at home to play games because it would help them develop good screen reader skills. By building on this, the research community can produce guidelines or a set of resources on how students with visual impairments can improve their basic computing skills at home. These resources can be shared with parents as guidelines for them to help their children at home.

We have also identified O&M skills as an important precursor skill that impacts the learning process of students with visual impairments learning how to code. Students with good O&M skills tend to understand faster and perform better than those deficient in O&M skills. This is because the type of programming activities and tasks commonly used to introduce novices to programming require basic knowledge of spatial navigation. Examples are robotics activities and tasks that require writing code to move an on-screen avatar on a grid space. Therefore, introducing O&M skills to students prior to teaching coding would reduce the challenges faced by students with visual impairments learning how to code. This is only applicable in situations where learning takes place through activities that inherently require spatial navigation concepts such as with robotic activities. Researchers and designers can also investigate innovative ways to reinforce students understanding of these spatial and directionality concepts while learning how to code. For example, in [91] the authors use tangible blocks with 3D arrow symbols to convey the direction the robot is going to take once that block of code is executed. This design helped the students understand right and left concepts [91].

4.2.5.3 Developing a Curriculum that Meets the Needs of Students

Based on the responses TVIs find the current lesson plans that come with various programming environments designed for kids either limited or complex for students with visual impairments to follow at some point. This implies that current curricula need to be revised not only to incorporate accessibility but to ensure that the lesson plans and the flow process provide a good learning curve for students with visual impairments. The only accessible CS curricula for students with visual impairments and students with disabilities in general include Exploring Computer Science by Ludi et al. [113][114] and the accessible AP CSP curriculum designed by AccessCSforAll [109] . These are notable efforts towards making computer science and programming accessible to students with visual impairments. The AP CSP curriculum is more popular given the connection to the Advanced Placement (AP) Program. The AccessCSforAll curriculum uses Quorum as the main programming language. Our data suggests that TVI find quorum advanced for their students or that TVIs lack the necessary CS background to consider Quorum as an alternative. TVIs in our study also expressed a lot of enthusiasm in having their students use the same programming environments and similar activities used by sighted kids. Researchers, practitioners, and educators can come together to consider ways in which to improve current curricula to include more programming tools and activities tailored to the needs of students with visual impairments and TVIs. Such efforts will require huge resources and involvement of TVIs and students with visual impairments as well.

4.2.5.4 Building Accessible On-Screen BBLs

The findings also shows that accessible on-screen BBLs are still uncommon in classroom settings with students with visual impairments. TVIs resorted to tangible alternatives which, however, do not provide a continuous learning process in certain situations such as with the current COVID-19 pandemic where lessons are conducted remotely. Apart from Blocks4All, a touchscreen based accessible BBLs which itself requires a DASH robot to operate, no other accessible on-screen BBL was mentioned. We believe we are not far away from designing and development fully accessible on-screen BBLs. Ludi et al. [36] designed an accessible version of Blockly that supports keyboard input and screen reader output. Our data also suggests that TVIs find programming environments with activities that control an on-screen avatar attractive to students with visual impairments provided the system conveys enough auditory feedback about the behavior of the program. Researchers and developers can build on these efforts to provide a fully accessible on-screen block-based programming environments for students with visual impairments. Such a system will allow students with visual impairments learn programming and computational thinking activities on the same systems that are found appealing to sighted novices [2]. These types of system will also provide a continuous learning process for the students and TVIs in situations where classes need to be conducted remotely. All participants in our interview study expressed interests to have an accessible on-screen BBL although some argued that BBLs do not prepare students for a professional career because they are not used in the industry.

4.2.5.5 Limitations

This report focuses solely on the accounts of TVIs. The opinion of curriculum designers and accessibility experts are not included. In addition, the number of participants in the study is relatively small to generalize these finding to all TVIs, existing curricula and current programming environments. The study discusses the issues associated with existing lesson plans at a highly level. A rigorous study of each existing programming environments and associated lesson plans is needed to highlight the specific issues that make the system limited or complex for the students. Last but not the least, an account of the students is needed to offer a complete picture of the accessibility barriers and the causes of these barriers.

4.2.5.6 Conclusion and Future Works

Through the lens of TVIs we have presented an account of the issues TVIs and students with visual impairments face in the day-to-day process of learning programming on existing programming environments and curricula. Our study revealed that TVIs find existing programming environments and curricula complex or limited for students with visual impairments. TVIs find themselves trying several programming environments designed for kids to find something tailored to the needs of their students. We also discussed Swift Playgrounds that seems to be the current best alternative to inaccessible BBLs and presented some issues surrounding the use of Swift Playgrounds. Our study revealed that the student lack of precursor skills is one of the main causes of accessibility challenges faced by students with visual impairments in the learning process. We terminated by offering some suggestions on how to improve current curricula and programming environments including efforts to train teachers with the required set of skills needed to provide a smooth learning process for students with visual impairments in computer science and computing related courses.

As future work, we plan to address some of the limitations of our study such as getting an account of the students to have a full picture of the issues and barriers affecting the learning process of students with visual impairments learning and practicing coding. We also plan to explore possible ways to design and develop a fully accessible on-screen BBL. We believe this type of system is needed to provide more alternatives for students with visual impairments to learn and practice programming and computational thinking concepts in a fun and engaging way.

4.3 Survey Study with Students with Visual Impairments

While the previous two sections in this chapter report on empirical studies with TVIs who identified a number of challenges faced by students with visual impairments on block-based programming and hybrid environments, one of the limitations of these studies as discussed is that they focus on the perspective of the teachers alone. We decided address this limitation by conducting a survey study targeted at students with visual impairments. The objective of this survey was to answer the following research question: Are the students reporting the same challenges as reported by TVIs?

To answer this research question, we conducted a survey study targeted at students with visual impairments learning how to code. Seven students with visual impairments shared their experiences on the use block-based programming and hybrid environments. Analysis of the data reported by the students corroborates the accounts of the teachers. In the following subsections we present an analysis of this report and how the voice of the students validates the account of the TVIs.

4.3.1 Methods

We conducted a survey study designed to get an account of students with visual impairments regarding the challenges that they face learning coding on block-based programming and hybrid environments. The survey was designed to be answered by the students alongside their parents after signing the consent form. The survey contained questions to capture the students experience and difficulties encountered while coding on a block-based programming or hybrid environment as well as about the students experience with assistive technologies and use of computers. We speculated that many of the respondents to the survey might not actually have heard of or used a block-based programming environment before and included a group of questions to elicit information about student experience on text-based programming. This design also allowed us to have a comparative view of the experiences of students on block-based programming and hybrid environments. This survey consisted of multiple-choice questions, Likert scale questions and open-ended questions (see Appendix D for complete questions). These questions were designed following the interview with the teachers. The questions were carefully curated to help and guide students into answering the questions with ease and with few words as possible. This study was approved by our institution's IRB.

4.3.2 Participants

We received seventeen submissions for the student survey. After filtering and removing submissions that were deemed incomplete, we retained a total of seven submissions for further processing. Four of these seven participants identified as males, two as females and one preferred not to answer. Details about these seven participants are shown in Table 4.4. All participants indicated having prior exposure to computing and programming. However, only four of the seven indicated that they had heard of block-based programming before and out of the four, only three had actually done block-based programming. The participants who had not done block-based programming shared their experiences on text-based programming as supported by the questionnaire design.

Table 4.4: Summary of student participants

Student ID	Gender	Age	Type of school	Type of Assistive technology used	Done BBP before?
S1	F	14	State School for the blind	Magnification software	Yes
S2	M	17	Private School	Screen reader, braille display	Yes
S3	M	42	Public School	Screen reader, braille display	Yes
S4	M	19	State School for the blind		No
S5	M	27	State school for the blind	Screen reader, braille display	No
S6	N/A	19	Public School	Screen reader	No
S7	F	40	Home Schooled	Screen reader, braille display	No

4.3.3 Analysis

The responses to the multiple-choice questions and Likert scale questions were tabulated. The responses to the open-ended questions were grouped together and used for extracting themes. Because the students were less verbose and direct in their responses, it was easy to categorize the response and compared them to the themes that emerged from the teachers interviews as this required fewer rounds of inductive coding steps.

4.3.4 Findings

The results of the student survey confirmed what the TVIs reported regarding the challenges encountered by students with visual impairments on block-based programming and hybrid environments as discussed in the previous sections.

4.3.4.1 Challenges from the Perspective of the Students

The survey started by asking students information regarding their background and level of skills including keyboarding skills, proficiency at their assistive technology and O&M skills. All the seven students rated themselves on average as highly skilled at using the keyboard or touchscreen with prior experience using computers, interacting with website and smartphone apps. All the students also reported to be highly skilled with their respective assistive technologies. However, when it came to rating their O&M skills, the seven students gave an average rating of moderately skilled, suggesting a deficiency in O&M skills as reported by the TVIs.

Further analysis of the students' data shows that their responses substantiate the report of the teachers. The three students (S1, S2 and S3) who used block-based programming stated that they used Swift Playgrounds and Scratch. S2 and S3 indicated using block-based programming environment for less than one year. S1 on the other had stated they used block-based programming for more than 1 year. S2 and S3 used Swift Playgrounds. When asked to rate how accessible the block-based programming environment they used was on scale of 1 to 5, with 5 being fully accessible, S2 and S3 indicated 2 and 5 respectively for Swift Playgrounds. S1, who reported they used magnification software to interact with computers indicated using Scratch to learn how to code. S1 however, rated Scratch as being less accessible with a rating of 3.

The questions also asked students to rate the level of difficulty of the five main actions that characterize a block-based programming environments, namely identifying a block on the toolbox, taking a block from the toolbox to the workspace, placing a block on the workspace with existing blocks, understanding how blocks are connected to one another on a program and editing an existing program on the workspace. The ratings were on a scale of 1 to 5, with 1 being very difficult and 5 being very easy. The students were also asked in open-ended questions to share any challenges that they encounter while accomplishing such actions.

For identifying blocks on the toolbox, S1 said it was very easy. S2 neither easy nor difficult and S3 very difficult. Looking at the level of vision, S1 has some vision and uses a magnification software. However, S2 and S3 completely rely on screen readers, suggesting this task is difficult for blind students. This corroborates with the teachers' report that blind students suffer more from the challenges than students with low vision. S1 and S2 indicated it was very easy to move a block from the toolbox to the workspace and this did not matter whether there were existing blocks on the workspace or not. On editing an existing program, S1 and S2 indicated it was neither difficult nor easy. However, the students reported facing some challenges with editing and this confirms the report by the teachers. S1 says trying to figure out how the program should be while editing is a challenge. On the other hand, S2 points to the VoiceOver rotor as being tricky to use to edit a program such as deleting a line: "Deleting a line and replacing it is tricky with voiceover, but doable."

The TVIs in their report repeatedly mentioned that the interactions required to use the VoiceOver rotor to edit a program was complex for the students to perform. The student might know what they need to change on a program, but performing the required action seems to be what makes the task difficult.

Most of the challenges reported by the students in the open-ended questions relates to editing. S2 who is blind stated that it was difficult to find the right spot because they explore the interface by touch. S1 although using a magnification software also reported that it was challenging to drag and drop blocks where they should go. This suggests that finding the correct location for inserting a new block is a major challenge that students face while editing or building programs. Although they reported difficulties with finding the right spot to add a new block, S2 stated that there was no difficulty with adding a new block at the end of an existing program, suggesting the difficulty is with inserting a new block between two different blocks, making editing existing program challenging as reported by the TVIs.

The students were also asked to indicate the action they take when they encounter a challenge. All the three students reported they sought help from an instructor with S2 indicating that they first try to address the challenge themselves before reaching out to the instructor. This suggests that training TVIs to be well competent with computer science knowledge as well as with the use of assistive technologies will improve the experience of students with visual impairments learning how to code on block-based programming environments and on any programming platform in general.

Although four out of the seven submissions that were analyzed indicated they had heard of block-based programming before the survey study, only 3 had actually tried block-based code. The remaining four students indicated they used text-based programming. This indicates that block-based programming is still not widely known among students with visual impairments despite their increased use in K-12 education. This can partly be attributed to the lack of accessible block-based programming environments. The text-based programming languages indicated by the students include Python, C#, Java and JavaScript. These students reported they coded on simple text editors including Notepad++, Emacs and custom editors. On average, the students found these editors to be accessible. Only S5 who worked with Python indicated using an advanced editor, namely VS Code. S5 however reported that VS Code was only partially accessible through their assistive technology and that it was missing features such as syntax highlighting via speech and indentation reporting. S4 indicated they used Minecraft [115], an online video gaming platform that also teaches coding in a fun way. The programming languages used in Minecraft include Java and Python. S4, however rated Minecraft as not accessible using their assistive technologies.

4.3.5 Discussion

Block-based programming is still not widely known by students with visual impairments despite the presence of a few accessible ones. Only three out of the seven students in our study had tried block-based programming before. In this section we discuss some strategies for mitigating these issues and for improving the overall learning process for students with visual impairments learning how to code.

4.3.5.1 Training the Students on O&M Skills

Interviews with TVIs also identified O&M skills as an important precursor skill that impacts the learning process of students with visual impairments learning how to code. Our survey study with the students shows that some are deficient in O&M skills. Although all the students rated themselves as highly skilled with keyboarding and touchscreen interactions, it was the opposite with O&M skills. All the seven students except one reported that they were moderately skilled when it came to orientation and mobility. This corroborates the teachers report and further suggests that deficiency in O&M skills affects the students learning experience. Therefore, we believe training the students to be good at O&M skills is an important step to improve the learning experience of students with visual impairments learning how to code.

4.3.5.2 Designing Simple Interaction Patterns

The data from the students and the TVIs suggest that sometimes the challenges are due to the limited affordances offered by the system. This is the case with the VoiceOver rotor that is required to edit programs on Swift Playgrounds. The students and the teachers both identified the VoiceOver rotor as being too complex to use when editing programs. The report from the TVIs suggests that students have difficulties figuring out how to manipulate the rotor. Therefore, designing systems to offer enough affordance regardless of abilities is essential in improving the experiences of students with visual impairments on block-based programming environments and hybrid environments.

4.3.5.3 Limitations

This report is based on a survey study alone. Follow up interview with the students would be required to offer a more detailed understanding of their experiences.

4.3.5.4 Conclusion and Future Works

We have presented an account of students with visual impairments regarding their experiences on block-based programming and hybrid environments. The reports shows that block-based programming is not widely known among students with visual impairments. This report also shows that the challenges reported by the students corroborates the accounts given by the TVIs. As future work, we plan to conduct follow-up interviews with students with visual impairments to gain a deeper understanding of their experiences as well as suggestion on how to improve their experience on these systems.

CHAPTER 5

NAVIGATING BLOCK-BASED PROGRAMS USING ACCESSIBLE BLOCKLY*

In this chapter, we introduce Accessible Blockly and an initial evaluation of it with respect to code navigation and code comprehension. Accessible Blockly is a continuation of the work done by Stephanie Ludi and her previous team. We used this initial version of Accessible Blockly as a starting point for designing and implementing an accessible block-based programming library that addresses some of the challenges discussed in Chapter 4 such as code navigation, code comprehension and code editing challenges.

After resolving multiple bugs on the initial version of Accessible Blockly, we conducted an initial formative study to address PRQ3: What keyboard interaction strategies allow efficient engagement and drag-and-drop-like operations for code navigation, code construction, and code editing on BBPEs? And PRQ4: How can we leverage the screen reader to provide adequate feedback to the user to improve code navigation, code construction, and code editing on BBPEs?

The results and feedback from the evaluation study presented in this chapter was used to make enhancements to Accessible Blockly. The work in this chapter is based on our work accepted for publication at the 24th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS 2022). The work was done in collaboration with Stephanie Ludi and Obianuju Okafor.

5.1 Introduction

Blockly [23][116][117] is the de-facto library on top of which most if not all

* Parts of this chapter have been previously published, either in part or in full, from Aboubakar Mountapmbeme, Obianuju Okafor, and Stephanie Ludi. 2022. Accessible Blockly: An Accessible Block-Based Programming Library for People with Visual Impairments. In The 24th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '22). Association for Computing Machinery, New York, NY, USA, (Accepted), with permission from ACM

mainstream block-based programming environments are built. Popular BBPEs such as Microsoft MakeCode, Scratch 2.0, AppInventor, and many others are built using the Blockly Framework. The Blockly library itself is not accessible to people with visual impairments, and as a result, all the mainstream BBPEs that rely on Blockly are not accessible [118]. Recent efforts by the Blockly team, including collaboration with our research group, have led to the addition of a keyboard interaction mechanism to Blockly. Figure 5.1 shows the demo editor for Blockly.

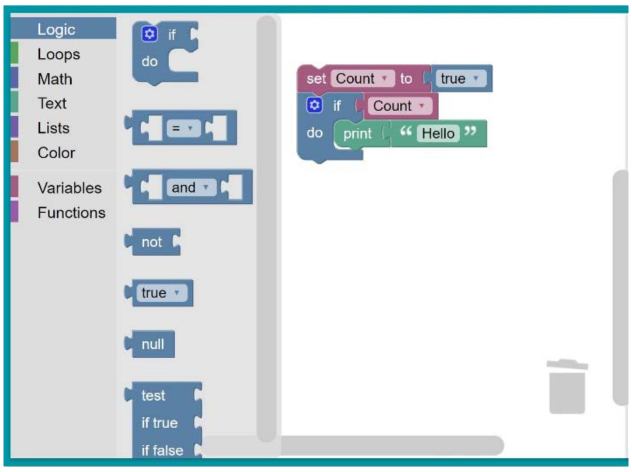


Figure 5.1: Blockly's editor showing the toolbox and a sample program on the workspace

In the following sections, we present the design and implementation of Accessible Blockly; a block-based programming library augmented with keyboard and screen reader interaction mechanisms. Accessible Blockly maintains the same visual and mouse-based interaction as the original Blockly and uses WAI-ARIA guidelines [53] to add screen reader and keyboard interactions. Accessible Blockly is a natural extension to Blockly and can be used in similar ways as Blockly. We believe having one system accessible to all would allow sighted and students with visual impairments to work on the same platforms, foster collaboration, and remove feelings of exclusion in settings where these systems are used. Accessible Blockly was designed with two main goals.

1. Enable keyboard interaction as an alternative to the mouse for navigating, creating, and editing block-based code.

2. Leverage the screen reader as an alternative output channel on Accessible Blockly.

We formulated the following research question to initially evaluate Accessible Blockly for block-based code navigation and comprehension: How well does the keyboard and screen reader navigation strategy in Accessible Blockly aid people with visual impairments navigate and understand block-based code?

The work presented in this chapter includes a prototype of an accessibility library, an analysis and results of the evaluation of the library for navigating block-based code, and finally, reflections on how to improve the accessibility of block-based programming environments.

5.2 Related Work

Although we have discussed the related work in detail in Chapter 3, in this section, we briefly talk about research work that are closed related to our research. Research on the accessibility of block-based programming to people with visual impairments is a growing field and dates back to 2015. In [14], the author points out that block-based programming is inaccessible to people with visual impairments mainly because of its visual and mouse-centric design. The author discusses design considerations to improve the accessibility of BBPEs through keyboard and screen reader interactions. The authors in [8] further present the need to make these systems accessible to people with visual and motor impairments, given their increased popularity in introductory computer science courses.

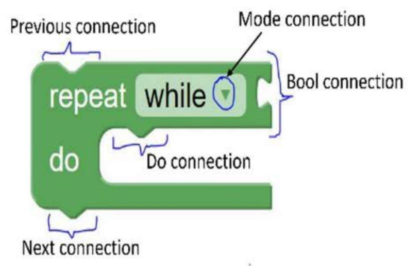
Blocks4All[4] is an accessible touchscreen-based block-based programming environment designed with students with visual impairment in mind. It runs on the iPad and allows users to build and run block-based code using touchscreen interactions and a screen reader. Blocks4All was designed to address the accessibility barriers in block-based programming environments, including accessing the output, accessing elements in the coding environments, conveying program structure and type information, and moving blocks [4][35]. In [51], the authors discuss the extension of Blocks4All to support functions and variables. Our work differs from this in that while Blocks4All is a complete coding environment where users can build and run block-based code, Accessible Blockly as a natural extension to Blockly is a library that allows people to develop an accessible block-based programming environment. It is an extension of the Blockly library that adds keyboard and screen reader interactions. One of the primary objectives of developing Accessible Blockly is to provide a mechanism for making existing mainstream BBPEs that run on Blockly accessible.

Das et al. [119] present an accessible solution that uses Blockly's Keybaord interaction mechanism and custom speech rather than a screen reader. The authors generate custom speech that describes the blocks in Blockly. Our work defers from this in two ways: 1) Accessible Blockly supports screen reader output, and 2) we offer a novel keyboard interaction mechanism different from Blockly's keyboard interaction scheme.

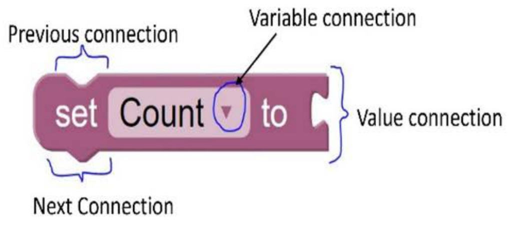
The keyboard and the screen reader have been used as alternative interaction mechanisms for people with visual impairments in text-based programming environments. Baker et al. developed Structjumper [25], which helps blind programmers navigate code in Eclipse integrated development environment (IDE) using keyboard shortcuts and screen reader output. CodeMirror-Blocks [39] is similar to Structjumper and allows blind programmers to edit and navigate text-based programs. It also offers a block view of text-based code where code sections are converted into visual elements to facilitate navigation. The bulk of research in improving the accessibility of programming environments for people with visual impairments has primarily focused on text-based programming [15]. In [15], the authors discuss the evolution of research in this field and present a summary of all the accessibility tools that have so far been developed.

5.3 System Design and Implementation

Before getting into the details of the design, it is important to briefly describe the morphology of a typical block and how block-based programs are constructed. Blocks are special visual elements that represent a particular programming construct. The shape of the block depends on the type of programming construct that it represents and on its intended functionalities. We can generally identify two types of blocks based on shape: non-container blocks and container blocks. As their name implies, container blocks are blocks that can accommodate other blocks within them. Examples include blocks that represent control flow statements such as the *"If"* blocks and *"repeat"* blocks (Figure 5.2 and Figure 5.4a). A block will have connection points (Figure 5.2). Connection points on a block allow other blocks to be attached to it or they enable the block to connect to other compatible blocks. The type and number of connection points also depend on the intended functionality of a block. A typical block-based program grows vertically downwards and left to right. Users select blocks from the toolbox and connect them on a workspace to form a program (Figure 5.1). Constructing block-based programs typically occurs via drag-and-drop operations using the mouse.



(a) Example container block and its connection points



(b) Example non-container block and its connection points

Figure 5.2: Example blocks and their connection points

Designing accessible BBPEs requires providing access to the rich visual information characteristic of BBPEs and also enabling an alternative interaction mechanism other than the mouse. We established five primary conditions while designing Accessible Blockly:

1. It should be usable by both students with visual impairments and sighted students alike. This includes collaboration between the two.
2. It should communicate information that is otherwise available visually in an alternate format such as speech and audio.
3. It should support keyboard interactions as an alternative to the mouse.
4. The library should be easy to integrate into existing mainstream block-based programming environments to enable accessibility on these systems.
5. It should be easy to learn and use.

Figure 5.3 shows the main components in Accessible Blockly.

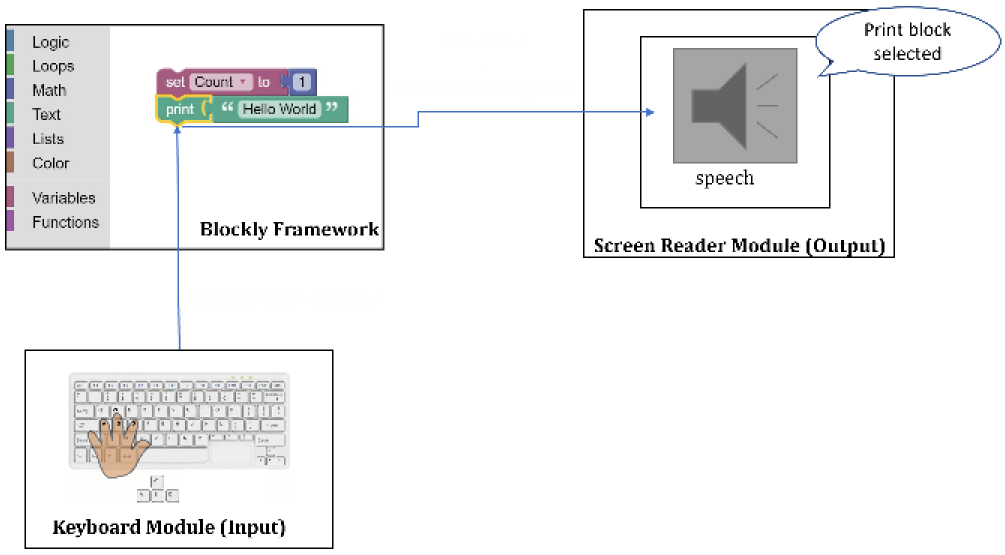


Figure 5.3: Components of Accessible Blockly

5.3.1 Blockly Library

The first component is the Blockly library [23], on top of which accessibility is being added. As stated earlier, Blockly is an open-source library developed by Google for creating block-based programming environments. Blockly became popular in 2016 after former US president Barack Obama introduced CS for All using the platform to code[120]. Blockly is built using JavaScript and runs on the web. This makes it a good candidate for use since Accessibility guidelines for Web Applications are well established through the W3C WAI group [121]. In addition, interactive coding platforms that use Blockly run on the web. With Blockly being open-source and widely adopted, anyone is free to add to or extend the Blockly framework. This also makes it suitable for our case. Another reason for using Blockly is that it already supports complex syntax such as functions and variables. Therefore, adding accessibility would allow users to have access to these rich sets of features by default. In [118], the authors report that TVIs find current accessible programming environments to be too basic to use at some point because they only contain basic coding concepts.

5.3.2 Keyboard Module

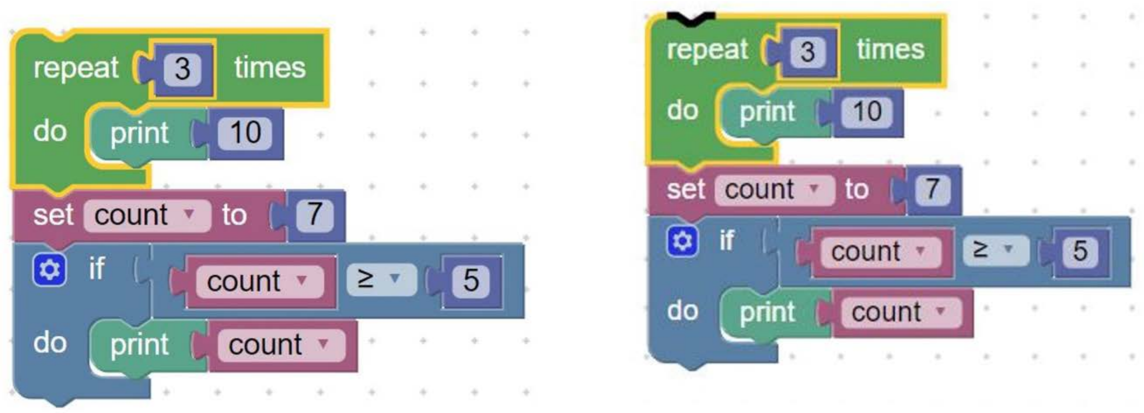
The second component is the Keyboard Module. Blockly, in its initial design, only supported mouse interactions. The Keyboard is an alternative to the mouse for people with visual impairments and temporary disabilities. We designed Accessible Blockly to support keyboard interactions with the following principal guidelines.

- Allow a user to browse block categories and individual blocks in the toolbox using keyboard inputs.
- Enable drag-and-drop-like operations via keystrokes alone.
- Mimic spatial navigation by allowing a user to navigate a program top-down, left-right, and transverse in and out of nested or container blocks.

We believe adhering to these principles is essential to produce one platform that is usable by people with visual impairments and sighted people alike. We defined a virtual cursor on the programming environment controlled by keystrokes. This cursor can be on the toolbox or the workspace at any time. When the focus is on the toolbox, a user can open a flyout category by using keystrokes, move the cursor between categories, and explore the blocks within a category in the same way they would use a screen reader and a keyboard to access HTML menus on the Web.

The keyboard module also allows users to perform drag-and-drop-like interactions from the toolbox to the workspace. Prior research [4] and our experimental work [36] have shown that there can be more than one way to perform drag-and-drop-like operations with a keyboard. We designed the drag-and-drop-like functions to closely mimic the actions performed by a user using the mouse. To move a block from the toolbox to the workspace, a user first selects a location on the workspace where they want to attach a new block using the virtual cursor. Next, they go to the toolbox and choose the desired block to be added, and then with a final keystroke, the selected block is placed at the previously determined location on the workspace. This method is commonly referred to as the location-first select, then block select method[4]. To improve the user experience and minimize confusion, once a user selects a connection point on the workspace and goes to the toolbox, blocks that are incompatible with the selected connection point are marked as disabled and communicated to the user via the speech output.

The keyboard module operates in two modes: Navigate mode and Edit mode. Navigate mode allows a user to move the virtual cursor through the blocks or categories of blocks on the toolbox or to navigate the blocks that make up a program on the workspace. The navigation allows a user to explore a program on the workspace spatially. The virtual cursor determines the current location of focus of the user. The cursor can be moved vertically downwards or upwards across blocks. It can also be moved horizontally between blocks connected at the same level. Figure 5.4a shows the cursor as a yellow outline around the *"repeat 3 times"* block. The cursor can also be caused to move directly into the body of a container block, such as an *"if"* block. This allows a user to continue exploration within the body of the container block.



a) The virtual cursor in navigate mode as a yellow outline around a block

b) The virtual cursor in edit mode as a dark outline around the connection point of a selected block.

Figure 5.4: The virtual cursors in Accessible Blockly

Edit mode is used to construct a new program or modify an existing program on a workspace. In this mode, the virtual cursor is used to select a connection point of a block on the workspace. The virtual cursor only moves between the connection points of the highlighted block. It allows a user to circle through the connection points and select a connection point at which to attach a new block. Upon choosing a connection point, the user goes back to the toolbox and selects a compatible block that is automatically snapped at the connection point on the workspace. The cursor is taken back to the workspace onto the newly added block. Figure 5.4b shows the virtual cursor as a black outline on the top connection of the *"repeat 3 times"* block.

Table 5.1: Keyboard navigation shortcuts for Accessible Blockly

Key	Action
W	Go vertically up to the previous block. Go to the previous connection point (in Edit mode).
A	Go horizontally left to the block on the left. Go out of the body of a container block to the container block.
S	Go vertically down to the next block. Go to the next connection point (in Edit mode)
D	Go into the body of the container block to the first child block
E	Toggle between Edit Mode and Navigate Mode

Key	Action
F	Go horizontally right to the first inline block or first block that is a property of the current block.
J	Jump to the first block on the workspace
R	Repeat current focus or location
C	Open the toolbox
ENTER	Add selected block to workspace
Delete	Delete selected block
Ctrl + Z	Undo Delete

The Keyboard Module uses the WASD keys as the primary keys for interaction (Table 5.1). These keys were selected because the WASD keys are common among gamers for navigation and can allow users to interact with the system one-handed. We chose to use keys other than the arrow keys for navigation to avoid conflicts with existing screen reader functionalities and other applications.

As an example of using the WASD keys to navigate the program, consider Figure 5.4a, and suppose the virtual cursor is on the *"if"* block. Pressing W would take the virtual cursor up to the "set count to 7" group of blocks. Pressing S at this location would take the cursor back down to the *"if"* block. Now if we are still on the *"if"* block, pressing F takes the cursor inline or horizontally right to the *"count >= 5"* block. The F-key causes the virtual cursor to move to the first property block of the current block. In Blockly terminology, these blocks are referred to as value blocks. These are blocks that cannot start a statement on their own and must exist as properties to other blocks. Another example of using the F-key is if we are on the *"repeat 3 times"* blocks as shown in Figure 5.4a. Pressing F would take the virtual cursor to the number block *"3."* Pressing A while at the number block *"3"* would move the cursor back to the "repeat 3 times" block. Pressing D while the virtual cursor is on the *"repeat 3 times"* block would cause the virtual cursor to move into the body of this container block to the *"print 10"* block. Pressing A causes the virtual cursor to move out back to the *"repeat 3 times"* block.

5.3.3 Screen Reader Module

This component was designed to serve as an alternative output channel to the visual channel. One of the ways in which people with visual impairments interact with computers is through screen readers. A screen reader is a program that reads out loud content on the screen. The screen reader module converts and conveys information otherwise only accessible visually into verbal descriptions. By default, blocks which are visual elements are not accessible through the screen reader. Unlike HTML images tag, for example, which can easily be made accessibly by adding the alt text, blocks are JavaScript drawings created at run time. Currently, there is no standard mechanism like the alt text to make blocks easily accessible through the screen reader. When the virtual cursor is focused on a block, the screen reader will output verbal descriptions based on what the block represents. Text descriptions for each block were carefully designed based on previous experiments [55] and using previously established guidelines for describing computer programs in speech or audio form [122]. Navigating the toolbox also allows the screen reader to read out the categories of blocks and each block within a category. As the virtual cursor moves from one location to another in the coding environment, the screen reader outputs a corresponding speech that describes the focused element. For example, if the focus is on the *"repeat block"* in Figure 5.4a, the user will hear *"repeat three times block."* Another example is if the focus is on the connection point shown in Figure 5.4b, the user will hear *"top connection."*

The speech for each block is designed to closely convey what the block represents. An aria-label text template is defined for each block type. The text label is updated at run time to reflect the actual state of the block. For example, there is an aria-label text *“repeat N times block”* for the repeat block. For the example in Figure 5.4a, the final text will be *“repeat 3 times block”* where N is replaced by 3. The screen reader will then output this information to the user. Aria-labels at run time can also come from a combination of two templates. Consider the *“set count to 7”* group of blocks in Figure 5.4. The resulting aria-label *“set count to 7”* comes from the template “set VARIABLE to block” and “NUMBER block.” “VARIABLE” is replaced by *“count”* and “NUMBER” by *“seven.”*

Accessible Blockly has been tested with NVDA, VoiceOver, and JAWS screen readers. With NVDA and JAWS, it works in focus mode or pass-key through mode. One of the design constraints was to maintain compatibility with existing applications while adding accessibility to Blockly. Having the screen reader work in these modes minimizes the chances of key conflicts between Accessible Blockly functions and other screen reader or browser functionalities.

Blocky is mainly based on JavaScript. Our Keyboard and Screen Reader modules are also implemented in JavaScript. The system listens for keypresses and calls special event handling functions to execute the requested actions and generate the speech output. Our code is open source. The repository alongside the wiki are located at [123]. This repository was forked from the official Blockly repository.

5.4 Experiment Design

To answer the research question, we conducted a study in which 12 blind programmers accomplished navigation tasks using Accessible Blockly and Blockly’s alternate navigation scheme as control. Our institution’s IRB approved the study. This evaluation is the first phase in our evaluation series and involves navigation tasks only. We decided to focus on the navigation task because:

1. This is the first attempt to design an accessible block-based programming library that uses the keyboard and screen reader for interaction. The Blockly's alternate navigation scheme originated from work by the Blockly team with a collaboration from our research group.
2. Navigation is crucial in writing code or building block-based programs. Therefore, we wanted to fully evaluate the tool for this activity before considering other coding tasks. Accessibility of code navigation is commonly assessed in isolation[25], [39].

Blockly[23] currently does not support screen readers. It only offers the alternate keyboard interaction mechanism. To use the Blockly's alternate keyboard navigation as control in our experiment, we replicated this navigation scheme in our codebase since this already supports screen reader interactions. Unlike Accessible Blockly, which offers two navigation modes as discussed above, Blockly's alternate keyboard navigation, referred to as the "Default Cursor" on the official website, has only one mode of interaction. In this single-mode interaction, the user uses the WASD keys to move the virtual cursor between blocks and connection points of blocks alike. Detail description of Blockly's alternate keyboard navigation scheme can be found at [124]. This work will act as one that evaluates both navigation schemes through a formal user study. We, however, chose to use this as a control because it is the navigation scheme that is shipped with the official Blockly library.

5.4.1 Participants

A total of 12 people took part in the evaluation study. There were ten males and two females. Participants were recruited via mailing lists, snowballing, and personal contacts. Participants had varying levels of experience with programming ranging from professional programmers to novices just learning how to code. Of the 12 participants, ten reported being blind, and the remaining two said they had no useful vision and relied on screen readers to interact with computers. Two of the participants were regular braille users and the rest used screen readers daily.

Table 5.2: Summary of participants' background information

	Gender	Age	Location	Programming experience	Screen Reader used	Screen Reader proficiency	Keyboard Proficiency
P1	Male	51	US	More than 3 years	NVDA	Expert	Expert
P2	Male	51	US	2 to 3 years	NVDA	Advanced	Advanced
P3	Male	40	Europe	More than 3 years	NVDA	Expert	Advanced
P4	Female	56	US	More than 3 years	NVDA	Expert	Expert
P5	Male	28	India	More than 3 years	NVDA	Advanced	Expert
P6	Male	57	US	More than 3 years	NVDA	Moderate	Advanced
P7	Male	26	Middle East	More than 3 years	NVDA	Advanced	Expert
P8	Female	40	Europe	less than 1 year	NVDA	Expert	Expert
P9	Male	22	India	less than 1 year	NVDA	Expert	Expert
P10	Male	29	Europe	More than 3 years	NVDA	Expert	Expert
P11	Male	60	US	More than 3 years	JAWS	Expert	Expert
P12	Male	35	Hong Kong	2 to 3 years	NVDA	Advanced	Advanced

The average age of the participants was 41 (SD = 13). Only one participant (P1) out of the twelve had heard of block-based programming before the study. P1 had tried Scratch, Blockly and MakeCode and reported these to be only partially accessible. Every participant signed an informed consent form before the study and received $50 Amazon gift card as compensation after the study session. Table 5.2 shows the summary of participants demographic data.

5.4.2 Set-Up

All study sessions were conducted remotely via Zoom. Participants were encouraged to use their regular screen reader settings and other settings related to their daily computer usage. Participants shared their screen and computer audio throughout the study session, and the researchers watched as they completed the tasks. Sessions were recorded for later analysis and insights.

5.4.3 Procedure

Participants were asked to provide demographic information in a pre-study survey hosted on SurveyMonkey. This survey asked participants to provide information regarding their level of vision, programming experience, experience with block-based programming, and with screen readers.

The study was organized in two phases based on the type of navigation tool used, i.e., Accessible Blockly vs. Blockly's alternate keyboard navigation. Each phase included a training task, two exercise tasks, and qualitative questions regarding the navigation tool used in that phase. After completing both phases, participants were asked open-ended questions regarding their overall experience using both navigation schemes. Before starting each session, the participants were briefly introduced to block-based programming. Participants were briefed on the morphology of a block and how block-based programs are built. Participants were also informed about the usefulness of block-based programming and the importance of making block-based programming accessible to all.

An identical program was used for training purposes in each phase. This program contained non-container blocks with assignment operations and a container block (see Appendix E for details). During training, participants were taught how to navigate a program using the keyboard shortcuts for each navigation scheme. Participants executed the keyboard commands following the instructions from the researchers. Participants were given time to familiarize themselves with the navigation schemes after the instruction stage. Once participants felt comfortable using the navigation scheme in each phase, we moved on to complete the exercise tasks.

We designed two sets of similar programs for use during the exercise tasks. Each group comprised of one simple block-based program without any container block. This program performed some basic arithmetic operations and updated the value of a named variable at the end of the program. The second program in each set had two container blocks: an *"if"* block and a *"repeat"* block. The program also included two *print* blocks. The participants completed the tasks on one set of programs using Accessible Blockly and on another set of programs using Blockly's alternate keyboard navigation scheme. Appendix E contains details about the study tasks. The link to the study tasks can be found at [125].

An effective navigation scheme should allow a user to get to desired locations on the code while building an understanding of the code and an awareness of their location within the program. This should also offer an acceptable navigation time. The two exercise tasks were designed to evaluate Accessible Blockly on these criteria. For each set of programs, the two exercise tasks included:

1. Determine the final value of a variable: Navigate the program and determine the final value of the variable "X." Here, X had different names on each set of programs, and the arithmetic operations were different on each set.
2. a) What is the expected output? Navigate the program and say what you believe will be printed on the screen if this program was executed. This program contained *print* blocks within container blocks.

   b) Conditions: how will the program's output in (2a) above change if the initial value of a variable in the program was set to a different value?

Task 2b was designed to test how well participants understood what the code was doing after navigating the program in Task 2a. The question asked participants to state how the program's output in Task 2a changes when the initial value of a variable used on the *"if"* block was set to a new value. The modification was such that it would negate the value of the Boolean condition and thus alter the program's output.

Participants were given 10 minutes to complete each task, except for Task 2b, a follow-up question to Task 2a. A timer was started after the researcher had finished reading out the question to the participant and when the participant indicated they were ready to start. The timer was stopped the moment the participant said they found an answer. All answers were recorded. Participants were allowed to take down notes as they completed the tasks. Eleven participants used NVDA, and one participant used JAWS during the study sessions.

All tasks were scored. Task 1 was scored on a scale of 0-1. A participant received 1 if they could determine the correct value of the named variable. Task 2a was scored on a scale of 0-2. A participant received full credit if they determined the correct output of the program. They received 1 if their answer was incomplete but partially correct. Task 2b was scored on a scale of 0-1. A participant received 1 if they correctly answered the follow-up question and 0 otherwise. For the timed tasks, participants received a 0 if they could not provide an answer within the allotted time.

At the end of each phase, participants were asked to rate their experience using the navigation scheme in that phase on a 5-point semantically anchored scale. We asked the following three questions. Similar questions have been used in studies evaluating accessible navigation tools in text-based programming [25][39].

1. On a scale of 1 to 5, 1 being very hard and 5 being very easy, rate how easy the tasks were to complete.
2. On a scale of 1 to 5, 1 being very frustrating and 5 being not frustrating at all, rate how frustrating the tasks were to complete.
3. On a scale of 1 to 5, 1 being I had no idea where I was and 5 being I always knew where I was, rate how well you know where you were in the code while completing the tasks.

After a participant had finished both phases of the study, they were asked to share their experiences with the two navigation schemes and state if they had a preference for any and, if so, why. Participants were also asked to share suggestions regarding improving the accessibility of block-based programming languages.

5.4.4 Design and Analysis

We designed a 2x2 within-subjects factorial design experiment with factors of the program sets and the navigation scheme used. Our experiment was modeled following similar experiments that evaluate accessible navigation tools for people with visual impairments [25][39]. Each participant completed two tasks using each navigation scheme. The order of the navigation scheme encountered first was counterbalanced between participants. However, each navigation scheme was fixed to a particular set of programs because the programs were designed to be very similar. How this design affects the study results are discussed under limitations. Each participant completed four tasks for the entire experiment for a total of 48 tasks. A single factor ANOVA was used to analyze tasks completion time. Descriptive statistics were used to analyze the semantically anchored scale data.

5.5 Evaluation Results

We analyzed the results and compared the effectiveness of the two navigation schemes based on task completion time, the correctness of answers to the tasks, and the participants' experience. Overall, participants were faster when completing the tasks with Accessible Blocky. Participants also scored higher on average with Accessible Blockly than with the alternate navigation scheme. Although most participants found both navigation schemes equally accessible, they tended to prefer accessible Blockly because they found it to be easy and less frustrating to use.

5.5.1 Task Completion Time

Participants were faster when completing the tasks using the Accessible Blockly navigation scheme, with an average task complement time of 1 minute 30 seconds against 2 minutes 55 seconds for the alternate navigation scheme (Table 5.3). This difference was statistically significant (P-value 0.0035). We note that fixing one navigation scheme to a particular set of programs throughout the study could also have an influence on the time difference and is addressed as a limitation. Participants completed Task 1 with Accessible Blocky in an average of 1m30s vs. 2m44s for the alternate navigation scheme. The same pattern was observed on Task 2a with an average task completion time of 1m30s for Accessible Blockly against 3m5s for the alternate navigation scheme.

Table 5.3: Task completion time

	Accessible Blockly Nav		Blockly Alternate Nav	
	Mean	SD	Mean	SD
Task 1 -Time	1m30s	59s	2m44s	1m43s
Task 2a - Time	1m30s	37s	3m5s	2m30s
Avg. Time	1m30s	48s	2m55s	2m6s

All participants completed each task within the allotted 10 minutes. Longer task completion times came from participants who had less experience with programming.

Figure 5.5 and Figure 5.6 compare the task completion time on both navigation schemes by participants for Task 1 and Task 2a respectively.

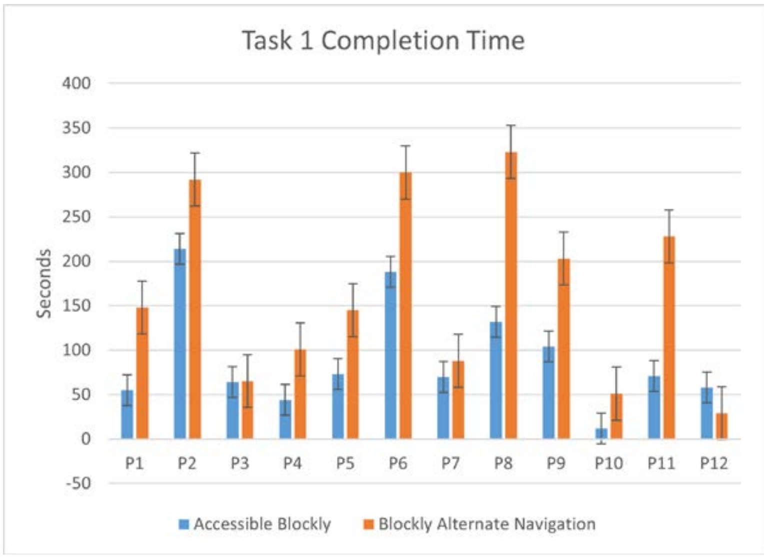


Figure 5.5: Task 1 task completion time by participants for both navigation schemes

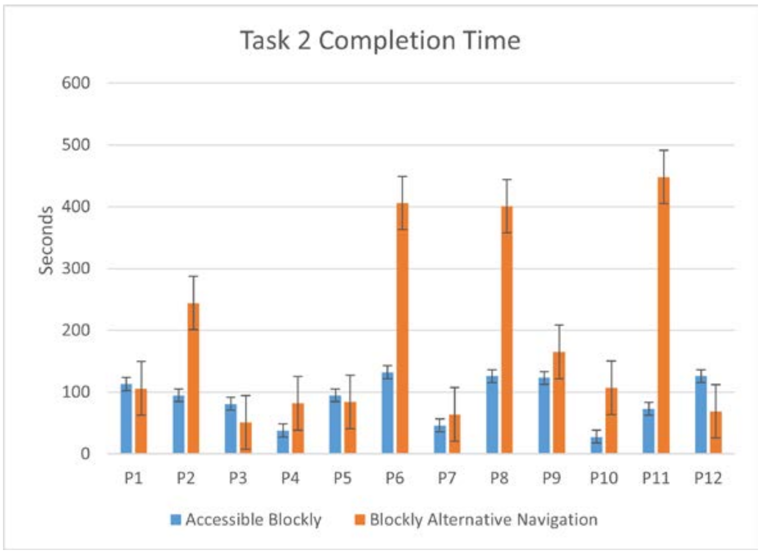


Figure 5.6: Task 2a task completion time by participants for both navigation schemes

5.5.2 Task Score

The maximum score possible for completing tasks correctly with each navigation scheme was four. As shown in Table 5.4, participants were more successful in completing the tasks accurately with Accessible Blockly (avg. score = 3.17) than with the alternate navigation scheme (avg. score = 2.75). Looking at the score by tasks, participants scored higher on Task 1 when using Accessible Blockly (avg. 0.83) than with the alternate navigation scheme (avg. 0.67). For both navigation schemes, some participants felt intimidated by the mathematical expressions in Task 1 and did not take time to properly evaluate the arithmetic expressions despite correctly navigating the programs.

Table 5.4: Task score

	Accessible Blockly Nav		**Blockly Alternate Nav**	
	Mean	**SD**	**Mean**	**SD**
Task1 - Score	0.83	0.39	0.67	0.49
Task 2a - Score	1.58	0.67	1.33	0.65
Task 2b Score	0.75	0.45	0.75*	0.45
Avg. Score	3.17	1.19	2.75	1.22

The performance in Task 2a was similar. On average, participants scored higher with Accessible Blockly (avg. 1.58) than with the alternate navigation scheme (avg. 1.33). Eight participants scored full points on Task 2a using Accessible Blockly against five for the alternate navigation scheme (Table 5.5). Participants who did not get the full points in Task 2a when using either navigation scheme forgot to explore the body of a container block in the program. Some participants attributed the miss to their unfamiliarity with the jargon on block-based programming. While it was easier to determine whether a block was a container or non-container block with Accessible Blockly, it was not the same with the alternate navigation scheme. With the alternate navigation scheme, some participants got confused by the connection points, which made them lose track of the level of nesting in the code. Because to go one level into the body of the container block using the alternate navigation, you must go through the connection points, it was difficult for participants to keep track of their level of nesting at the same time while remembering the location of the virtual cursor through the connection points. Participants who got partial points in Task 2a thought one of the container blocks was nested inside the other, which was not the case. Again, participants with experience in programming did not have this confusion.

Table 5.5: Number of participants who got full points per task

	Accessible Blockly Nav	Blockly Alternate Nav
Task 1	10	8
Task 2a	8	5
Task 2b	9	9*

*After they were given hints and completed Task 2a

For both navigation schemes, participants who did not get Task 1 or Task 2a correct after the timer was stopped were given hints to complete the task. All participants who failed at the first attempt successfully completed the tasks with recommendations. However, this was not accounted for in the task score and task completion time above. For Task 2b, participants on average scored equally using Accessible Blockly (Avg. 0.75) and the alternate navigation scheme (avg. 0.75). This suggests participants had a good understanding of the programs they navigated. However, more participants successfully completed Task 2a on first trail with the timer on when using Accessible Blockly than with the alternate navigation scheme. The three participants (P2, P8, and P11) who did not respond correctly to Task 2b while using either navigation scheme reported they had limited programming experience. In addition, P8 and P11 were seasoned braille users and mentioned having a hard time keeping up with the screen reader output. These could also have influenced their performance at the tasks.

While observing participants complete the tasks, we noticed that all participants could explore all the blocks within the programs and within the allotted time. No participant gave up in the middle of completing a task. For each navigation scheme, all participants could explore all the blocks on the programs for each task. We noticed excitement on the part of some participants, which made them impatient to reflect on their answers properly. In this precipitation to give an answer, some participants made mistakes. This was more apparent with Task 1 which involved arithmetic operations.

5.5.3 Participant Experience

Participants also shared their experience regarding the level of difficulty of the tasks, their degree of frustration while completing the task, and about their awareness of their location on the program while completing the tasks. A higher number is better as it indicated they found the tasks easy, not frustrating, and were more aware of their location within the program. Overall, participants felt Accessible Blockly was easier and less frustrating to use. Participants explained that it required few keystrokes to get to a desired location, and the fact that they did not have to deal with the connection points while navigating made their experience less frustrating. The participants also thought they were more aware of their location within the programs while navigating with Accessible Blockly.

The largest difference in the semantically anchored scale data between the navigation schemes was on the level of difficulty. Participants overall rated Accessible Blockly to be easier to use (avg. 4.33) than the alternate navigation scheme (avg. 3.75) (Figure 5.7). We note that all participants found both navigation schemes equally accessible and usable for navigating block-based programs.

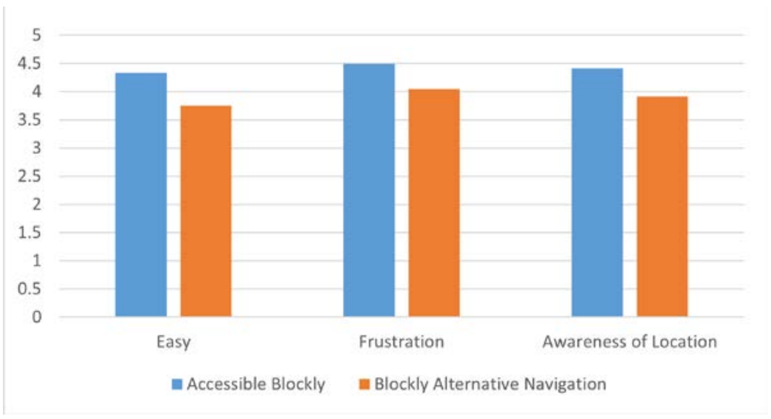


Figure 5.7: Average score across participants on the semantically anchored questions

5.6 Qualitative Results

After participants had completed the tasks with Accessible Blockly and the alternate navigation scheme, we asked participants open-ended questions regarding their experience learning and using both navigation schemes. Participants were also asked whether they preferred any of the two navigation schemes. Participants were also allowed to provide suggestions regarding improving the usability of the navigation schemes and the accessibility of block-based programming in general.

Multiple steps of inductive analysis were applied to the transcripts from each session. Codes were identified from each transcript through multiple coding rounds. Using the affinity diagramming technique, these codes were grouped into themes. Several themes were identified and are discussed below.

5.6.1 Easy to Use and to Learn

Analyses of the open-ended questions revealed that participants found Accessible Blockly easy to use and learn. Nine out of the 12 participants explicitly stated that accessible Blockly was easier to use to navigate the programs. The participants attributed this ease of use to the fact that you do not have to worry about the connection points while navigating the program with accessible Blockly:

> I prefer the navigation two [Accessible Blockly]. Because I think it's easiest to navigate because you don't have to worry about the connection. So, I prefer the second one, phase two one." (P3)

> Obviously, this [Accessible Blockly] is quite a bit more some, like simplified because you don't have all the connection points to worry about so this is just simpler. (P10)

On the other hand, five participants found the alternate navigation scheme challenging and less intuitive. Participants explained that navigating the connection points at the same time as blocks made learning and using Blockly's alternate navigation scheme less intuitive.

> It [Blockly’s alternate navigation scheme] is easy not very easy. I mean here, the training right, how to use this navigation is much more important because it's not very intuitive. It's a bit difficult. So yeah, a lot of things to kind of understand properly. (P9)

Other participants thought the alternate navigation scheme has a steep learning curve and that its ease of use could improve with experience. Because each block has its morphology and type of connection points, it can be overwhelming for the users at the beginning.

> I guess my umm I really feel like I wanna make it complicated but that’s because I’m just learning it [Blockly’s alternate navigation scheme]. I think you know I think after time this would be easier like I said because it actually goes left and right and I would be able to explain it better to other people you know like what I was doing. In one sense it’s more complicated to learn, I think. (P1)

5.6.2 Shorter Navigation Time

The quantitative results already showed that Accessible Blockly was faster for navigating the programs. Additionally, six participants also mentioned in the open-ended discussions that they found Accessible Blockly to be faster when navigating the programs. The participants said they could quickly get to their target on the block-based code because they had fewer keystrokes to execute. This is because Accessible Blockly in navigation mode does not go through the connection points, saving extra keystrokes from the user.

> … I mean the other thing that I thought was different I thought the second one [Accessible Blockly] felt like it had a little less steps may be cause like you know when going to the hollow block before we did D and for this particular question there was one like one of 2 choices I didn’t’ really have to go deeper so in that aspect it was kind of like less layers to go into in the second session whereas in the first session [Blockly alternate navigation scheme] when we did the hollow blocks it seemed like there was more like cause I had to go into the do ummm connection and had to go in there and then come back out and come back out. (P2)

In addition to being faster to navigate, participants thought accessible Blockly provided quick review of the programs.

> Phase 2 (Accessible Blockly) in terms of reviewing a program is easier because it allows you to navigate the programs quickly. You don't have to go into the details of the connection points as with the Phase 1 [Blockly alternate navigation scheme]. (P6)

On the other hand, participants thought using the alternate navigation scheme to read the programs made it longer to navigate the code. This longer navigation time is also evident in the quantitative data presented above.

> Getting into the different connections made it a bit longer than the first one, phase one [Accessible Blockly]. (P3)

5.6.3 Less Cognitively Demanding

While reflecting on their experiences, participants also found that in contrast to Accessible Blockly, the alternate navigation scheme was more cognitively demanding. 6 out of the 12 participants explained that using the alternate navigation scheme to read the programs increased their memory constraints and required extra attention.

> …umm I think it could umm I mean I know when I was doing it, it [Blockly alternate navigation] required a little bit more time to go in there and then I had to focus a little bit more on where am I at, coming back out you know like like ummm It felt like the second one [Accessible blockly] I didn't have to worry, … as soon as I was in, I was back out. Whereas the first time [alternate navigation scheme] I was in then I was in then I was like okay when I came out where am I at when am I gonna come out again. (P2)

Having to remember the connection points while navigating to get to a target location was mostly cited as the reason for the increased cognitive load. Participants felt that the alternate navigation scheme exposes too much information during navigation, making it more challenging to explore the code. Exposing connection points during navigation increased navigation time and had participants retain more in their memory.

> In the phase one [Blockly's alternate navigation scheme] you have to keep track of the connection points which you don't need if you are not editing the program. I don't know how it might be in terms of editing but the Phase 2 [Accessible Blockly] was straight forward. (P6)

Although some participants made remarks regarding the increased memory requirements, they thought the detailed information provided could be more helpful when navigating complex programs. P5, while providing his rating for the level of frustration with the alternate navigation scheme noted:

> I gave 4.5 because I had to process bit more of an information but I believe like to thoroughly understand a complex program I might need all this information. (P5)

On the other hand, five participants thought it was the opposite when using Accessible Blockly because they did not have to worry about the connection points when reading the programs.

> Because it [Accessible Blockly] just gives you uhh enough information you want to understand the program. uhh it did not give you more details which can make things a bit uhhh like which can increase the cognitive load. The things you need to intake are more in the second phase [Blockly alternate navigation scheme]. So, in first phase [Accessible Blockly] like the cognitive load was very much less, and so as frustration and understanding would be a bit more. So, for a novice or for uhh like student uhhmm learning first phase was very much good. (P5)

One participant mentioned it was easier to memorize the navigation keys with Accessible Blockly than with the alternate navigation scheme.

> I have a preference for navigation scheme applied to phase one [Accessible Blockly], because that's much more easier. Okay, it's only few keys. So that's much more easier to memorize because being a blind individual, always you need to memorize key shortcuts. So it's, I mean, it should be as simple as that. So, in that case, I have a preference with phase one [Accessible Blockly]. (P9)

5.6.4 Suitable for Novices

As participants thought Accessible Blockly was straightforward, faster, and less cognitively demanding, they also stated that it seemed more suitable for novices and appropriate for simple programs. P5 explicitly mentions this in the previous paragraph. One of the primary objectives of block-based programming is to make learning programming by novices easier by removing the heavy syntax imposed by text-based languages[2] [91]. Looking at it from this perspective further strengthens the observations made by participants. We also observed from the qualitative data that participants with less programming experience performed better with Accessible Blockly than with the alternate navigation scheme, suggesting it was easier for them to learn and use.

On the other hand, participants who had more experience with programming thought the alternate navigation scheme was more appropriate for navigating complex programs because it provided more information which would give them more control especially when it comes to editing.

> So, for reading the program it was easier with the way it worked in phase one [Accessible Blockly], umm I would imagine when it came to editing you would need more control like what you have in Phase 2 [Blockly's alternate navigation scheme]. (P3)

5.6.5 Participant Preference

Participants were explicitly asked if they had a preference for one of the navigation schemes over the other and their responses were interesting. Five participants explicitly stated that they preferred Accessible Blockly over Blockly's alternate navigation scheme. Three participants said they liked both navigation schemes citing various use cases in which they find both to be valuable and appropriate. The most common reason for preferring Accessible Blockly was its ease of use and design to favor novices. Blockly's alternate navigation scheme on the other hand was mostly preferred by experienced programmers who stated it was because it provided more information that might be valuable when navigating complex programs. One participant, P10, stated they preferred the alternate navigation scheme over accessible Blockly. What was more intriguing was the three participants who said they liked none of the navigation schemes. P12 explicitly stated this: "I don't have a preference for any. I feel they end up trying to map the WASD keys to the Arrow keys." (P12)

Overall, we observed that both navigation schemes are equally usable and accessible for navigating and understanding block-based programs, with Accessible Blockly being preferred over the alternate navigation scheme because of its ease of use, faster navigation time, and its tendency to favor novices.

5.7 Discussion

5.7.1 Participants with Programming Experience and Experience Using Screen Readers Perform Better

The individual reports and quantitative data show that participants with good programming experience performed best, having the fastest task completion time and highest score with both navigation schemes. These participants also found the tasks to be easier to complete. This was the contrast with those who had little or no programming experience. The participants with limited programming experience scored low on the tasks. When watching the video recordings of participants who scored low on the tasks, we noticed that most missed exploring the body of the container blocks or control flow blocks in Task 2. Five participants missed exploring the body of container blocks in Task 2a, and this was more common when they completed the task with the alternate navigation scheme. When asked why they missed, some cited their limited programming knowledge explaining that they did not know these blocks had blocks nested within them. This was the case for P2 and P8.

It could also be that these participants had not formed an accurate mental model of block-based code and their conceptualization of block-based code was still influenced by their text-based program reading strategy. At least the discussion with three participants suggested this. P4, who missed exploring the container blocks in both Phase 1 and Phase 2 said her familiarity with procedural languages influenced her thought process.

> Yeah, I think it’s just my training as you know umm because normally in a procedural language the thing to be repeated would be on the next line and I just haven’t gotten used to looking inside the block for what should be repeated. (P4)

We believe that in addition to adding accessibility to block-based programming, training students with visual impairment to form an accurate mental model of how block-based code works would be essential in increasing their success in using block-based programming environments. Training students to use screen readers and other accessibility technologies would also improve their performance at block-based coding[35].

5.7.2 Enthusiasm Regarding both Methods and on the Accessibility of BBPEs in General

We observed that almost all participants (9 out of 12) were enthusiastic as they used both navigation methods to complete tasks and welcomed the idea of adding accessibility to block-based programming environments. Participants displayed positive emotions and frequently smiled as they completed the tasks and reflected upon their experience. Most participants expressed interest in using the navigation schemes, and some even requested more complex programs as they were curious and eager to see how they work on larger programs. Overall, it was a feeling of enthusiasm, satisfaction, and enjoyment. For the participants who had not heard of block-based programming before, it felt like a new and exciting experience. The only participant familiar with the accessibility issues in mainstream BBPEs [35], [118] stated they were happy to see efforts being made to add accessibility to these systems. P1, who also teaches students with visual impairments how to code and has tried some mainstream BBPEs stated:

> I really like this though you guys have done […]. This is much better than the Google outline method that they had. Umm what was it? I don’t know how many years ago? Ummm. (P1)

P1 was referring to a text-based version of Blockly that was designed by the Blockly team to be accessible to people with visual impairments. The problem with this was that it was a separate system and thus not suitable in situations where students with visual impairments and sighted students work together.

5.7.3 Suggestions to Improve the Accessibility of Block-Based Code

As discussed in the previous section, all participants in the study expressed enthusiasm and gave feedback regarding how we could improve the navigation strategy and the accessibility of block-based code in general. We examined all the suggestions made and noticed that they were similar across participants. Most of the recommendations had to do with the feedback provided by the screen readers. All participants found the keyboard navigation strategies accessible.

However, Participants thought the verbal descriptions provided by the screen reader could be improved and enriched with more feedback regarding the actions taking place on the coding environment. Participants suggested providing hints about container blocks. Participants thought letting the screen reader alert them that a block is a container block would improve their success at navigation and reading block-based code.

Participants also requested a quick review mode that would allow one to listen to the description of the whole program or part of a program without navigating the program. This would give the user a head start before they begin exploring the individual blocks on the program.

We note here that adequate feedback from the programming environment is crucial to an effective and fully accessible programming environment for people with visual impairments. Therefore, researchers and practitioners should explore how to increase the amount of feedback a user can obtain from such interactive visual environments through alternate interaction channels. Sound and auditory cues have been used for similar purposes in text-based programming environments [122] [126] [44].

5.8 Limitations

One obvious limitation of this work is the number of participants who took part in the study. Due to the difficulty in recruiting participants, we could only evaluate Accessible Blockly with 12 participants. Although 12 is high when compared to similar studies [4] [25].

As stated before, each navigation scheme was fixed to one set of programs throughout the study. This might have also influenced the quantitative results and their statistical significance. However, the programs were designed to be very similar in length and content and we believe this configuration had negligible influence on the results. Additionally, during the open-ended discussions most participants felt Accessible Blockly was faster for navigation.

5.9 Future Work

This study mainly evaluated Accessibly Blockly for navigating and reading block-based code. This was the first study in our series of evaluation studies. Part of the next steps involves looking at how Accessible Blockly allows people with visual impairments to create and edit block-based code. Participants were very eager about this and almost all of them wanted to create block-based code.

Updating the screen reader output by improving the verbal descriptions and increasing the amount and type of information a user would get through the screen reader is also part of our future work. Participants gave suggestions regarding the generated speech, and we believe we can address these in our future iterations. Audio cues have been used to increase the amount of feedback and information that users get when navigating text-based programs [122] [44]. We plan to explore these mechanisms and other ways to improve the feedback generated by the screen reader.

5.10 Conclusion

We have presented a prototype of an accessible block-based programming library that uses the keyboard and screen reader for interaction. This library is a modification of the Blockly library used to build most mainstream block-based programming environments. The library was designed to improve the accessibility of block-based programming environments for people with visual impairments. Using the library, a user can navigate, create, and edit block-based code using the screen reader and keyboard only. This paper also reported a user study evaluating the library on code navigation tasks. The results show that people can effectively navigate block-based code with a keyboard and screen reader. Participants in the study found Accessible Blockly to be fast, easy to use, and less frustrating. They also showed enthusiasm about this initiative to improve the accessibility of block-based programming environments. Participants also expressed interest in future studies to evaluate the library for code creation and editing.

5.11 Acknowledgment

We thank all participants who took part in the study and Google's Blockly team for their collaboration.

CHAPTER 6

CREATING AND EDITING BLOCK-BASED PROGRAMS USING ACCESSIBLE BLOCKLY

In this chapter, we present the enhanced Accessible Blockly and an evaluation of it for block-based code creation and editing. This chapter addresses the remaining parts of PRQ3: What keyboard interaction strategies allow efficient engagement and drag-and-drop-like operations for code navigation, code construction, and code editing on BBPEs? And PRQ4: How can we leverage the screen reader to provide adequate feedback to the user to improve code navigation, code construction, and code editing on BBPEs?

The work in this chapter was done in collaboration with Stephanie Ludi and Natalio Castaneda.

6.1 Introduction

Accessible Blockly was enhanced based on the feedback from the initial evaluation study presented in Chapter 5. The system was then evaluated in an empirical study with 11 blind programmers who used the keyboard and screen reader to create and edit block-based code. The evaluation results show that blind programmers can efficiently create and edit block-based code using the keyboard and the screen reader alone. The participants in the study also described Accessible Blockly as being accessible and easy to use without any accessibility concerns. The participants, however, gave suggestions for improvements as they were excited about the project and eager to see it grow with more features that would improve the overall experience of blind programmers.

The remainder of this chapter discusses the enhancements made, the empirical study design, the results of the study, and reflections on improving the design and the accessibility of block-based programming environments for people with visual impairments.

6.2 Enhanced Accessible Blockly

Based on the qualitative results from the empirical study discussed in Chapter 5, some updates were made to Accessible Blockly to improve the user experience. The updates involved resolving bugs, adding new keyboard shortcuts, and optimizing the screen reader speech to provide more contextual information. The bugs mainly were unexpected behaviors from existing features. The issues tracker on the GitHub repository was used to efficiently track and resolve bugs.

6.2.1 Keyboard Module

In Table 6.1, the rows in bold represent the new keyboard shortcuts that were added to Accessible Blockly. The I-key causes the virtual cursor to move to the right between value blocks that are siblings of the same parent block. The Shift-I key would cause the virtual cursor to move in the reverse direction. For example, as shown in Figure 6.1, if the virtual cursor is on the variable block *"count"* attached to the comparison block, pressing I would move the virtual cursor to the number block "5" to the right. Pressing Shift + I would cause the virtual cursor to move back to the "count" block. If no more blocks are at the extremes, the screen reader would announce that to the user.

Table 6.1: Updated keyboard shortcuts in Accessible Blockly

Key	Action
W	Go vertically up to the previous block. Go to the previous connection point (in Edit mode).
A	Go horizontally left to the block on the left. Go out of the body of a container block to the container block.
S	Go vertically down to the next block. Go to the next connection point (in Edit mode)
D	Go into the body of the container block to the first child block
E	Toggle between Edit Mode and Navigate Mode
F	Go horizontally right to the first inline block or first block that is a property of the current block.
J	Jump to the first block on the workspace

Key	Action
R	Repeat current focus or location
C	Open the toolbox
ENTER	Add selected block to workspace
Delete	Delete selected block
Ctrl + Z	Undo Delete
I	**Traverse left to next sibling of current block if available**
Shift + I	**Traverse right to previous sibling of current block if available**
ESC	**Close the toolbox and return to workspace**

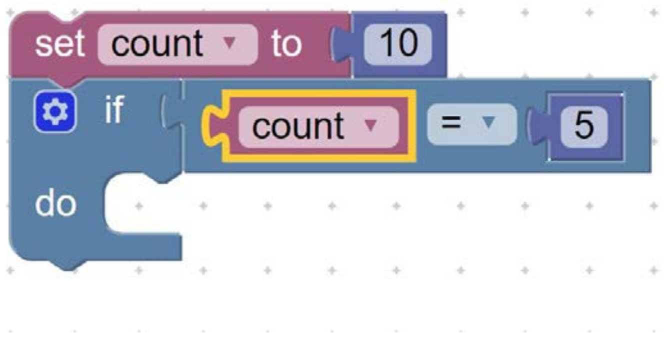


Figure 6.1: A sample block-based program to illustrate navigation

Another keyboard shortcut that was added is the ESC key. In Edit mode, after a user has selected a connection point on the workspace and opened the toolbox, if the user for one reason or the other wants to return to the workspace without adding a new block, pressing ESC would return the virtual cursor to where it was on the workspace and exit Edit mode. The screen reader speech would let the user know they have returned to the workspace.

6.2.2 Screen Reader Module

Many updates were made to the screen reader module to provide more feedback and contextual information. These are described in the following paragraphs.

The previous version of Accessible Blockly did not give feedback after a block was successfully added to the workspace. A user had to use the R-key to read their current location to get a confirmation of their action. The screen reader module was updated to provide feedback to the user after a block is moved from the toolbox to the workspace. After selecting a block from the toolbox and hitting the ENTER key, the screen reader would output the phrase *"block added to workspace"* as a confirmation to the user that the action was completed successfully.

In the previous study, participants expressed the need to distinguish between the container and non-container blocks. Based on our analysis, we decided to provide this information through the screen reader speech alongside the block information. Accordingly, the phrase *"container block"* was prepended to the speech for all container blocks found in Accessible Blockly. For example, in the previous version, focusing on the *"if"* block shown in Figure 6.1 would cause the screen reader to output the phrase *"if count = 5 block."* With the new update, the user would hear *"container block if count = 5."* This lets the user know that they are currently on a container block and that it might be necessary to check the body of the container block for child blocks.

The system's screen reader module was also improved to let the user know when a keyboard shortcut does not apply in a given context, such as using the D-key on a non-container block. If a user presses the D-key on the *"set count to 10"* block shown in Figure 6.1, the screen reader will output *"cannot move inwards, not a container block."* Similarly, if a user presses the D-key on a container block with no child blocks such, as the "if" block in Figure 6.1, the screen reader would output *"Cannot move inwards, no child blocks."* Similar feedbacks were added for other actions where there are either no next steps or where a given keyboard shortcut does not apply.

The screen reader module was also augmented to provide feedback about the direction of navigation and changes in nesting or control flow. The screen reader outputs the phrases "nesting in" and "nesting out" to let the user know they are moving into the body of a container block and out of the body of a container block, respectively. Similar phrases were designed to indicate horizontal inline traversals to the left and to the right. When traversing inline to the right, the screen reader would output the phrase *"traverse in ,"* before the name of the newly focused block is read out. For example, if cursor's current location is on the *"if"* block shown in Figure 6.1, pressing the F-key would cause the screen reader to output the phrase *"traverse in count equals 5."* Pressing the A-key to go back to the "if" block causes the screen reader to read out the phrase *"traverse out if count equals 5"* to indicate to the user that they have moved horizontally left to the "if" block.

6.3 Experiment Design

We designed an experiment to evaluate Accessible Blockly on code creation and code editing. The objective of the experiment was to determine the usability of Accessible Blockly in allowing blind programmers to build and edit block-based code and to obtain feedback to improve the system's overall design. Participants created and edited simple and slightly complex block-based programs during the experiments. The experiment in Chapter 5 demonstrated that Accessible Blockly was accessible and easy to use to navigate and understand block-based code. This was insufficient as it only evaluated Accessible Blockly on one aspect of block-based coding. This experiment completes the evaluations and fully addresses PRQ3 and PRQ4. The IRB approved the investigation before being conducted.

6.3.1 Participants

A total of 11 blind software developers took part in the study. All participants reported being blind. Participants had varying levels of experience with programming ranging from less than one year to more than three years. Participants were all males with an average age of 37.7 years. The participants were recruited via mailing lists and personal contacts. Participants from the previous study who agreed to be contacted for this second study were also invited to participate. Out of the 11 participants, eight were returning participants who participated in the study discussed in Chapter 5. Only four participants indicated that they had tried block-based programming outside our evaluation studies. Participants resided in the United States, India, Canada, and Europe. Each participant was compensated with a $50 Amazon gift card after the study. Table 6.2 shows the details of the participants.

Table 6.2: Participants details including experience with programming and assistive technologies

ID	Age	Profession	Screen Reader Proficiency	Keyboard Proficiency	Programming Experience	BBP Experience	Returning Participant?
P1	42	SWE	Expert	Advanced	> 3 years	0	yes
P2	28	A11y Consultant	Expert	Expert	> 3 years	0	Yes
P3	29	SWE	Expert	Expert	> 3 years	0	yes
P4	51	SWE	Expert	Expert	> 3 years	> 3 years	Yes
P5	55	N/A	Advanced	Advanced	> 3 years	0	No
P6	27	Salesforce Admin	Expert	Expert	> 3 years	< 1 year	Yes
P7	57	SWE	Advanced	Expert	> 3 years	0	Yes
P8	49	Associate SWE	Advanced	Expert	> 3 years	< 1 year	No
P9	23	SWE	Advanced	Expert	2 to 3 years	0	Yes
P10	24	A11y test engineer	Expert	Expert	2 to 3 years	0	no
P11	30	social science	Advanced	Expert	> 3 years	2 to3 years	yes

6.3.2 Set-Up

The study was conducted remotely via zoom. Each session typically lasted about 45 minutes. Participants were encouraged to use their regular screen reader settings while completing the study. Participants were asked to share their screen and audio throughout the sessions. The researchers watched as participants completed the tasks. All sessions were recorded and transcribed for analysis.

6.3.3 Procedure

We designed and scripted an experiment in which participants were required to complete three exercises after an initial training task. The experiment was scripted to facilitate the process during each session and guide the session to maintain the time limit of approximately 1 hour per session. For the training task, participants started with a single block of code on the workspace (Figure 6.2a). The researchers trained the participants with step-by-step instructions until the final program shown in Figure 6.2b was created.

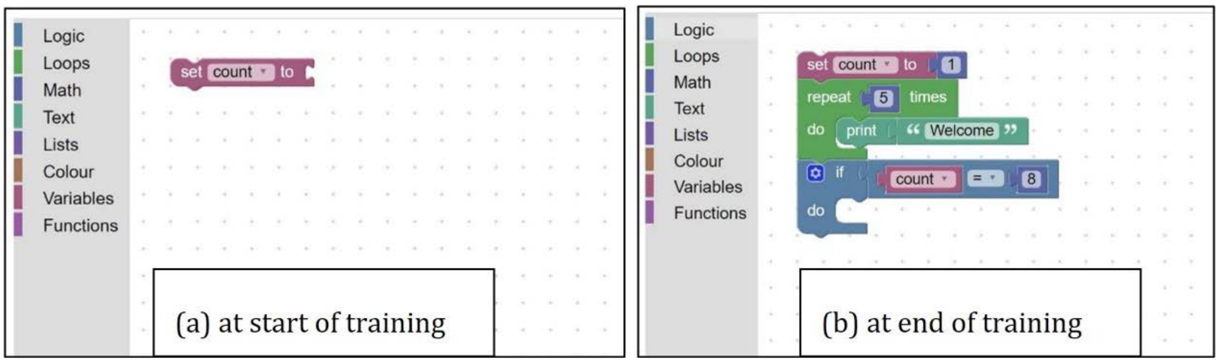


Figure 6.2: Training task at start and end of training

Task 1 on the study script required participants to build a program that would print "hello" 3 times using the *"repeat N times"* block (Figure 6.3). Participants started with an empty workspace and had to build the program using the keyboard and screen reader alone.

Task 2 started with a sample program on the workspace (Figure 6.4a). This program prints "12A3" as output. Participants were asked to edit and correct the program such that the output of the program becomes "123" (Figure 6.4b).

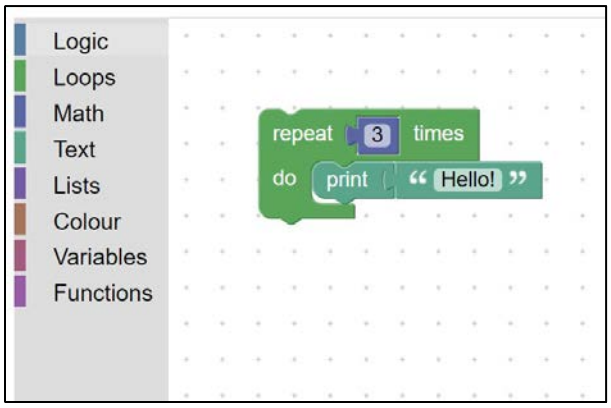


Figure 6.3: Accepted program for Task 1

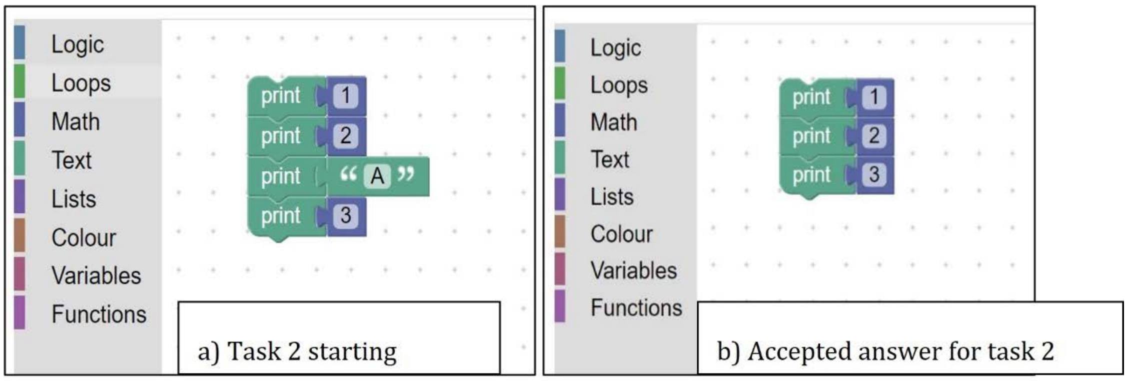


Figure 6.4: Task 2 starting program and the expected answer after completion

Task 3 asked participants to extend the logic of an existing program. The initial program would print “high school” as output when the value of the variable named “age” is less than or equal to 18 (Figure 6.5). Participants were asked to extend the program to print “college” when the value of the variable “age” was greater than 18. This task was very interesting because it had more than one way of being accomplished. Figure 6.6 shows three different acceptable answers for Task 3.

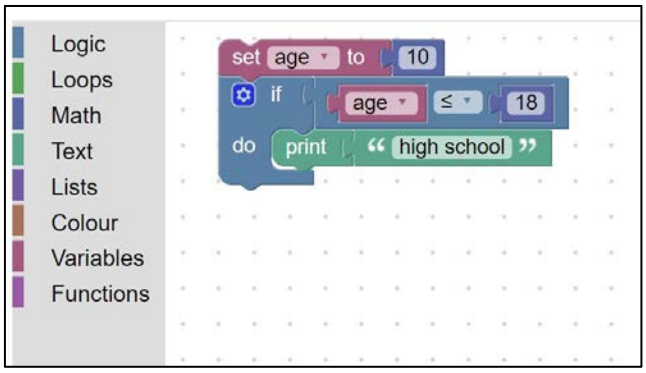
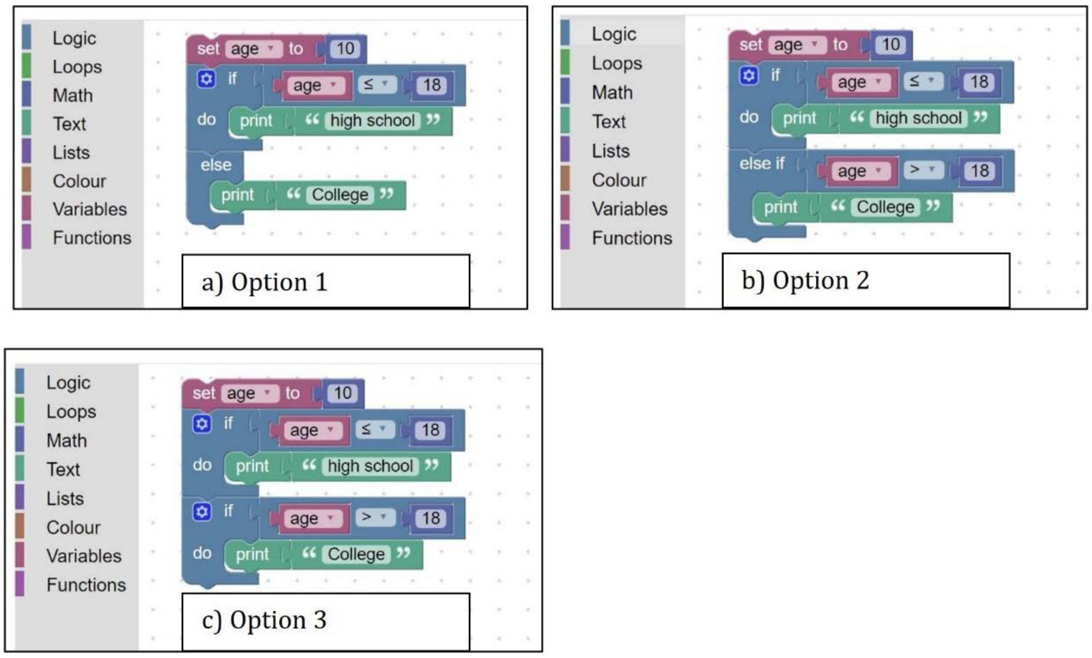


Figure 6.5: Starting program for Task 3



a) Option 1

```
set age to 10
if age ≤ 18
do print " high school "
else
   print " College "
```

b) Option 2

```
set age to 10
if age ≤ 18
do print " high school "
else if age > 18
   print " College "
```

c) Option 3

```
set age to 10
if age ≤ 18
do print " high school "
if age > 18
do print " College "
```

Figure 6.6: Acceptable answers for Task 3

Before conducting the study, we ran a pilot test to pre-evaluate the study with a blind student in the Department of Computer Science and Engineering. The student volunteered to test the system and provide feedback. The objective of the pre-evaluation was to identify any bugs in the system and calibrate the time it would take for a single session. The blind student is a sophomore and has good programming experience. The session lasted for approximately 1 hour 30 minutes. No significant issues were identified during this pilot test except for a few minor bugs that were easy to fix. We anticipated the session to take approximately 1 hour. However, we exceeded that limit. This was mainly due to some initial set-up issues encountered with zoom such as the student having difficulties to share their screen and audio simultaneously. The other factor that consumed time was the training to use Accessible Blockly. Although the student typically took less than 30 minutes to complete the 3 exercise tasks in the study, training took more than 30 minutes. After the pilot test, we noticed our approach to training was designed to consume time. We made the necessary adjustment to the training script to optimize the time it would take. We also fixed the bugs we found during the pilot test and decided to proceed with the evaluation study.

During this training, participants were taught how to open the toolbox, browse categories of blocks on the toolbox, go into a specific category, browse blocks within a category and add a selected block from the toolbox to the workspace while creating a program step by step. The participants were also taught how to delete a block from the workspace and how to undo an action. Participants executed instructions one after the other as instructed by the researchers. Participants were also given time to explore the different categories of blocks and blocks within all categories that were relevant to the exercise tasks later. As participants executed the instructions, they also asked questions to clarify their understanding. Once the program shown in Figure 6.2b was created, participants were given extra time to assimilate what they had just learned, to further familiarize themselves with the system and ask any clarification questions. On average, the training sessions lasted for about 25 minutes. Once participants were comfortable and ready, we moved on to the exercise tasks.

Each of the exercise tasks was designed to evaluate the effectiveness of Accessible Blockly in aiding users to overcome the major challenges that were reported by TVIs and students with visual impairments in Chapter 4. These include difficulties identifying blocks from the toolbox, difficulties editing a program on the workspace, difficulties deleting a block on the workspace, and difficulties adding blocks to an existing program on the workspace.

Task 1, which started with an empty workspace tests a participant's ability to create a block-based program from scratch. It was designed to test the user's ability to find blocks from the toolbox, add blocks to an empty workspace, add blocks within a container block on the workspace as well as test the user's ability to understand the control flow of block-based programs. This task required users to use the "repeat N times" block to restrict users to only one possible solution and focus on testing the usefulness of the system in the tasks mentioned above.

Task 2, on the other hand, focuses on testing the effectiveness of Accessible Blockly in allowing to users delete a block within a program on the workspace. It also tests the user's ability to navigate and understand block-based code with the keyboard and screen reader alone. The user had to figure out how the program output was generated and what needed to be done to edit the program to produce the required output. The user was not given any hint whether they had to use the delete feature or not. Users were simply instructed to read the program and edit the output to the quested format.

Task 3, in addition to evaluating Accessible Blockly for the feature stated for Task 1 also focuses on testing the effectiveness of Accessible Blockly in allowing users to edit and extend block-based code, one of the major challenges reported by the TVIs and students. Unlike Task1 in which users to restricted to use a particular block, Task 3 was open-ended. This also allows us to test the creativity of the participants on the system and their ability to discover different features and blocks on the platform. Task three could be accomplished by using a single "else" block, by using an "if-else" block or by using a second "if" block. Figure 6.6 illustrates all the different possibilities for accomplishing Task 3. It was up to the participant to figure out what to do, and no hints were given to any participant.

Each of the exercise tasks was timed. Each participant had a maximum of 10 minutes to complete each task. A timer was started after the researcher explained the exercise to the participant and the participant indicated they were ready to start. The timer was stopped when the participant indicated they had finished or if the allocated 10 minutes had elapsed.

Each task was scored on a scale of 0.00 to 1.00. Participants received 1.00 if they completed the task correctly. Participants received 0.50 if their answer was partially correct after the timer was stopped. Partial marks were awarded only if a participant was on the right track to complete the program but ran out of time. An example of partially correct programs for Task 1 and Task 3 are shown in Figure 6.7 and Figure 6.8 respectively.

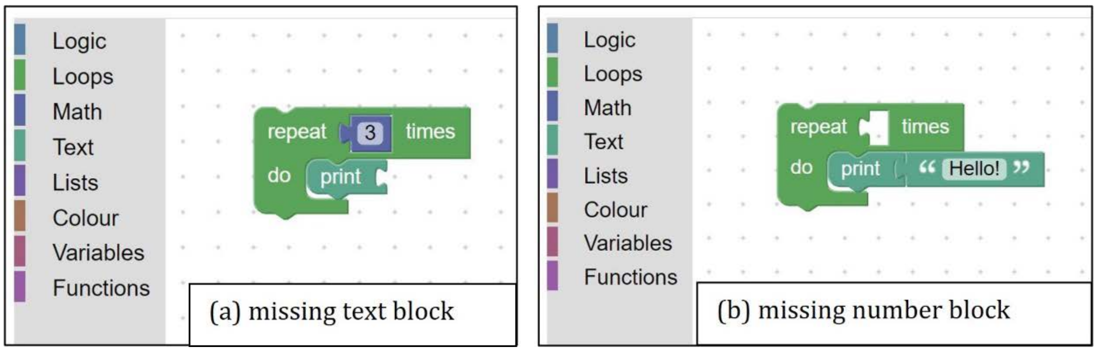


Figure 6.7: Examples of partially correct programs for Task 1

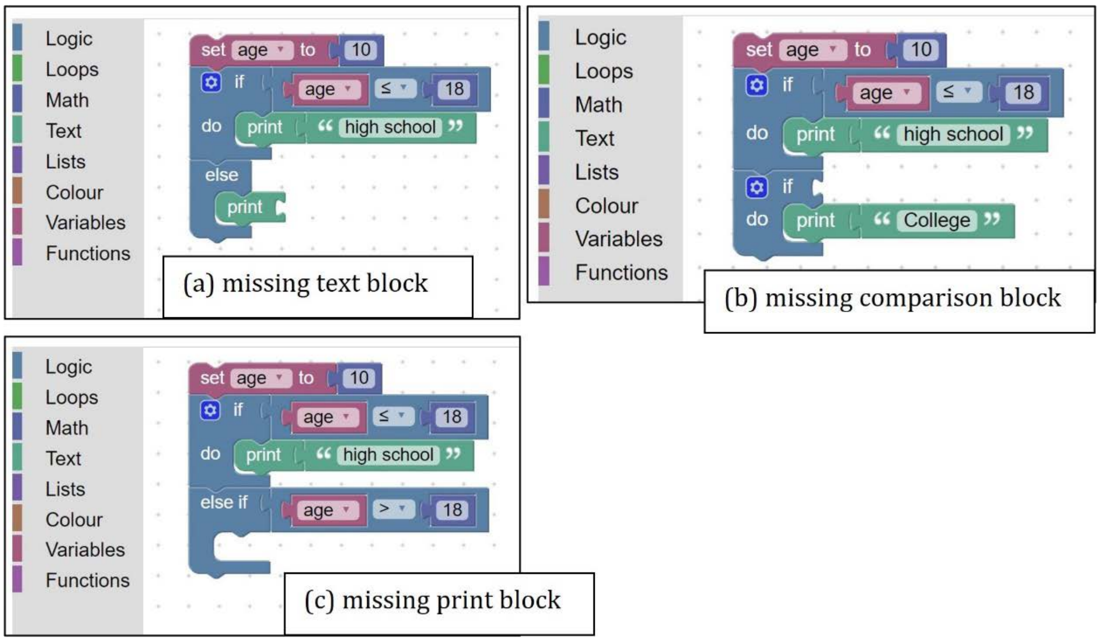


Figure 6.8: Examples of partially correct programs for Task 3

After completing the exercise tasks, participants were asked Likert scale and open-ended questions to capture the participant's experience and thoughts using

Accessible Blockly to create and edit block-base code. Using a 5-point semantically anchored scale, we asked seven questions that allowed participants to rate their experience on different aspects of block-based coding. These questions were curated from the results of our studies with TVIs and students, and from similar studies evaluating accessible navigation tools in text-based programming [25][39]. The questions include:

1. On a scale of 1 to 5, 1 being very hard and 5 being very easy, rate how easy the tasks were to complete.
2. On a scale of 1 to 5, 1 being very frustrating and 5 being not frustrating at all, rate how frustrating the tasks were to complete.
3. On a scale of 1 to 5, 1 being very hard and 5 being very easy, rate how easy it was for you to find an insertion point on the workspace at which to insert a block.
4. On a scale of 1 to 5, 1 being very hard and 5 being very easy, rate how easy it was for you to find a block from the toolbox.
5. On a scale of 1 to 5, 1 being very hard and 5 being very easy, rate how easy it was for you to add a block to the workspace.
6. On a scale of 1 to 5, 1 being I had no idea where I was and 5 being I always knew where I was, rate how well you know where you were in the code.
7. On a scale of 1 to 5, 1 being very hard and 5 being very easy, rate how easy it was for you to delete a block on the workspace.

We also asked open-ended questions for participants to reflect on their thoughts and actions using the keyboard and the screen reader to create and edit block-based code. The complete study script is in Appendix F.

6.3.4 Analysis

The data from all the sessions were grouped and analyzed. Descriptive statistics were used to analyze the quantitative data and the semantically anchored scaled data. Inductive analysis was used to analyze the data from the open-ended questions. Working with Natalio Castaneda, we first coded two transcripts together and identified codes that we agreed upon. Once the initial list of codes was established, we independently coded the remaining transcripts. We frequently met to discuss the codes and resolve any disagreements. Another round of inductive coding allowed us to group the codes into themes. In the following sections, we provide the complete evaluation results and reflections on improving the usability of keyboard and screen reader interactions for creating and editing block-based code as found after our analysis.

6.4 Evaluation Results

6.4.1 Task Completion Time

Overall, participants completed all the tasks in an average of 4 minutes 10 seconds per task. Table 6.3 shows the average task completion time and the task completion time by participant for each task. Looking at the task completion time by tasks, for Task 1, participants took an average of 5 minutes 20 seconds (STD = 3 minutes 30 seconds) to complete the task. Eight out of the 11 participants completed Task 1 in less than 10 minutes, with the lowest completion time of 1 minute 32 seconds by P3 and the longest completion time of 10 minutes by P7, P9, and P9. These participants could not complete the task within the allotted 10 minutes.

Table 6.3: Task completion time by participant

Participant	Timings		
	Task 1	Task 2	Task 3
P1	1min58sec	18sec	1min35sec
P2	3m06sec	18sec	10min
P3	1min32sec	19sec	6min15secs
P4	2min33sec	44sec	2min36sec
P5	4min20s	25sec	9min32secs
P6	2min58sec	23sec	3min24secs
P7	10min	38sec	10min
P8	8min29sec	1min10se	5min53sec
P9	10min	29sec	10min

<table>
<thead>
<tr><th rowspan="2">Participant</th><th colspan="3">Timings</th></tr>
<tr><th>Task 1</th><th>Task 2</th><th>Task 3</th></tr>
</thead>
<tbody>
<tr><td>P10</td><td>10min</td><td>3min35sec</td><td>6min11sec</td></tr>
<tr><td>P11</td><td>3min39sec</td><td>36sec</td><td>4min46sec</td></tr>
<tr><td>Average Time</td><td>5min20sec</td><td>47sec</td><td>6min23sec</td></tr>
</tbody>
</table>

Task 2 was the easiest to complete and all participants completed the task within the allotted 10 minutes with an average task completion time of 49 seconds (STD = 57 seconds). The fastest task completion time in Task 2 was 18 seconds and the longest was 3 minutes 35 seconds by P10.

For Task 3, eight out of the 11 participants completed the task within 10 minutes with an average task completion time of 6 minutes 23 seconds (STD = 3 minutes 8 seconds). The shortest task completion time for Task 3 was 1 minute 35 seconds by P1 and the longest task completion time was 10 minutes by P2, P7, and P9.

As stated earlier, the average task completion time across participants for all tasks is 4 minutes 10 seconds. It should be noted that the majority of the participants completed the tasks in less than 5 minutes. As would be seen in the qualitative results, participants found the system to be fast and easy to use to accomplish the tasks. Shorter task completion time came from participants who had more experience with programming and took part in the first evaluation study reported in Chapter 5. This shows that familiarity with the system and programming experience impacted the performances. Additionally, it took a shorter time to train these participants during the training task as they were already familiar with the system.

6.4.2 Task Score

Generally, participants scored well on the tasks. The maximum score possible for all three tasks was 3.0. The average total score across participants was 2.41 (STD = 0.83).

Six of the 11 participants scored full points (3.00 out of 3.00) on all three tasks. For Task 1, eight participants completed the tasks correctly within 10 minutes and scored full points. One participant scored partial marks (0.50) and the remaining two scored 0.00 as they could not complete the program correctly within the allotted time.

Task 2 was the easiest to complete and all participants scored full points on Task 2. Participants were less successful in completing Task 3. Only six participants scored full points (1.00) on Task 3. Two participants scored partial points and the remaining three scored 0.00. Participants who scored 0.00 could not complete the program correctly within the allotted time. Table 6.4 shows the score by participants for each task.

Table 6.4: Task score by participants

	Task 1	Task 2	Task 3	Total Score
P1	1.00	1.00	1.00	3.00
P2	1.00	1.00	0.5	2.5
P3	1.00	1.00	1.00	3.00
P4	1.00	1.00	1.00	3.00
P5	1.00	1.00	1.00	3.00
P6	1.00	1.00	1.00	3.00
P7	0.00	1.00	0.00	1.00
P8	1.00	1.00	1.00	3.00
P9	0.00	1.00	0.00	1.00
P10	0.50	1.00	0.00	1.50
P11	1.00	1.00	0.50	2.50
Average	0.77	1.00	0.636364	2.41
STD	0.41	0.00	0.45	0.83

6.4.3 Participant Experience

Participants were asked to rate how they felt completing the tasks regarding their level of frustration and the level of difficulty of completing the tasks. Participants were also asked to rate the level of difficulty of accomplishing five major actions inherent to creating and editing block-based code, including finding an insertion point on the workspace, finding a block from the toolbox, adding a block to the workspace, deleting a block on the workspace and knowing their location with respect to the code on the workspace. Participants rated their experience on a scale of 1 to 5. A higher number is desired as it implies participants found the task or action to be easy or not frustrating.

Regarding Question 1 on how easy the tasks were to complete, all participants gave an average score of 3.81 (STD = 0.87). This implies that the participants found the tasks easy to complete using the keyboard and the screen reader. All participants gave a rating of at least 3 to describe how Accessible Blockly keyboard and screen reader interactions helped them build and edit block-based code. The qualitative data from the open-ended questions confirms this as described in the next section.

Regarding their level of frustration while completing the tasks with the keyboard and screen reader (Question 2), participants gave an average rating of 3.77 (STD = 1.03), suggesting the tasks were less frustrating and close to being frustrating-free. Two participants, however, gave a rating of two regarding their level of frustration. The participants (P7 and P10) felt the tasks were somehow frustrating to accomplish. These participants could not complete Task 1 and Task 3 within the allocated 10 minutes and did to get full points.

Participants found the actions of finding an insertion point on the workspace (Question 3) to be easy to accomplish with an average rating of 4.27 (STD = 0.79). This suggests the keyboard and screen reader interactions in Accessible Blocks easily allow users to find a point at which to add a new block to the workspace. Similarly, participants also rated the ability to find a block from the toolbox (Question 4) with an average rating of 4.55 (STD = 0.52). Participants, on average, also found it to be easy to add a block from the toolbox to the workspace using the keyboard and the screen reader (Question 5) with an average rating of 4.65 (STD = 0.51). Adding a block to the workspace was a combination of locating an insertion point, identifying a block on the toolbox, and moving the block to the identified insertion point. This task which was reported by the TVIs and students to be challenging seems to be relatively easy to accomplish on Accessible Blockly as supported by this data.

The lowest average score on the semantically anchored scale data came from Question 6 on how well participants knew their location within the code. Although all the participants gave ratings of at least 3 regarding the awareness of their location, the average rating was 3.45 (STD = 0.82). Participants explained in the open-ended questions that they would have benefitted from more feedback from the system, although they said the feedback was enough for them to accomplish their tasks. One other reason for the low rating was attributed to minor bugs that went unnoticed during the pre-evaluation phase but were not obstacles to an effective use of the system.

For the last question on how hard it is to delete a block on the workspace, all participants thought it was very easy with an average rating of 5 (STD = 0). All participants and almost all with a smile said it was very easy to delete a block on the workspace. Editing to delete blocks was reported by TVIs and students as one of the most challenging tasks to accomplish on Swift Playgrounds, a hybrid block-based programming environment. This difficulty was attributed to the high dexterity required to manipulate the VoiceOver rotor on the iPad. We address this challenge on Accessible Blockly by making it easy for a user to delete a block or group of blocks with a single keystroke. Currently, to delete a block, a user needs to focus on the intended block and hit the delete key on the keyboard. The screen reader provides verbal feedback to the user that the action has been accomplished. The user can also undo this action using the Ctrl + Z keyboard combination. Figure 6.9 shows the average ratings for each semantically anchored scaled question with standard deviation as error bars.

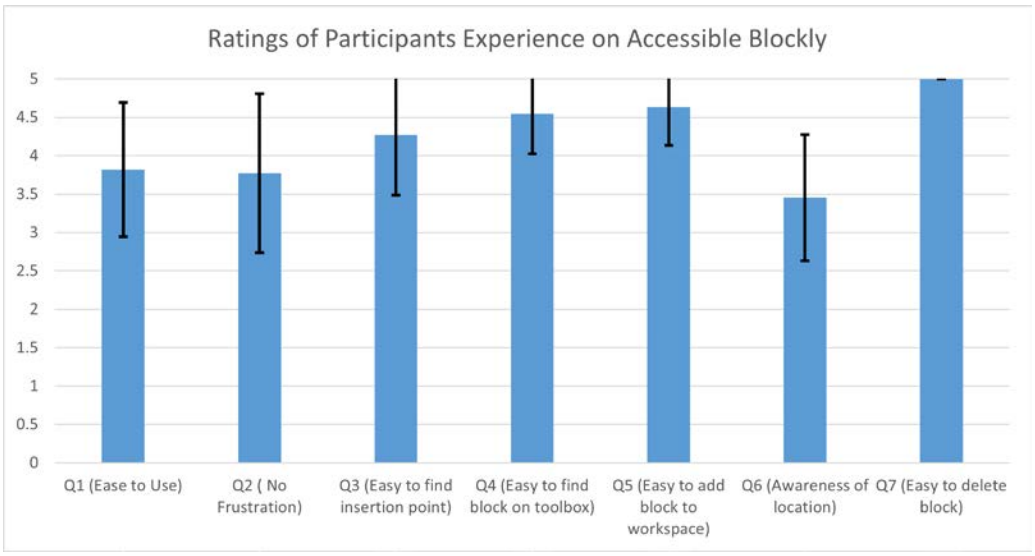


Figure 6.9: Average of each semantically anchored scaled data on participants experience

6.5 Qualitative Results

6.5.1 Qualities of Accessible Blockly

Participants shared their thoughts and experiences regarding using Accessible Blockly to build, edit and navigate block-based code through open-ended questions and discussions. After analyzing the data from the transcripts, most of the results appeared as suggestions and feedback to improve the usability of Accessible Blockly or requests to add new features to the current system that participants deemed would provide a better user experience. Overall, all the participants found Accessible Blockly to be easy to use to build and edit block-based code. In the following sections, we describe in detail the qualitative data and offer suggestions on improving the accessibility of block-based programming environments to people with visual impairments.

6.5.1.1 Accessible Blockly is Accessible and Easy to Use

Participants were asked to describe their experience creating and editing block-based code using the keyboard and screen reader on Accessible Blockly. Almost all participants (N=10) described Accessible Blockly as being easy to use and accessible as suggested by this quote from P4:

> Okay, I think this is very easy to do, so I like this. I think the only thing is discoverability, we need to work on …

P1 thought the system was straightforward to use: "Umm, I will say it was fairly straightforward"

The response from the participants suggests that the system has no accessibility issues. Participants were happy and showed feelings of excitement when using the system. P9 described his experience as follows:

> Certainly, I certainly like it because it was very fast. I mean, JAWS has been slow to us, otherwise it's really fast, and it's really very prompt, I mean, there is no accessibility issue.

The data also suggests that system design offers a shallow learning curve to new users. Five participants explicitly hinted that they see their proficiency in using the system improve with time and practice as the keyboard interaction patterns and screen reader output are intuitive. P9 reported: "…But as of now, the moment you get hands on it, it's really more intuitive for sure."

6.5.1.2 Screen Reader Output is Enough

Participants were also asked to describe their experience with the speech generated by the screen reader as feedback from the system. The objective was to examine if the amount of feedback generated by the system was sufficient for the user to create and edit block-based code. Seven participants thought the feedback from the screen reader was enough to accomplish the tasks. P4 described his experience in the following words

> I think it [screen reader feedback] was perfect. Yeah, and I think I like it, it interrupted fine. That was what I was testing when I sped through the text field. You've got to make sure that you both give enough and you also make it shut up, and it was perfectly doing that, so yeah I think that's great.

P10 who was using Accessible Blockly for the first time expressed his satisfaction with the screen reader feedback as follows: “Feedback was very descriptive and easy to understand. It was great.”

This qualitative that from the participants shows that speech as an alternative output channel on block-based programming environments is effective at allowing users to create, edit, and navigate block-based code.

6.5.1.3 WASD Keys are Good for Interaction

Participants were also asked to describe their experience using the keyboard shortcuts to interact on the block-based programming environment. Most participants (n=7) found the keyboard shortcuts to be okay and easy to learn and remember. P1 explicitly noted:

> Yes, they are easy. They are easy to remember. Yeah, I had to think about D and F a little bit cause they are someway in the way. Yeah, I got it. I got it fairly quickly. I think in the toolbox…

P7 also corroborated the report by P1 as suggested by the following quote:

> Keyboard shortcuts? They’re okay. I got the hang of it from the last time we did it, the W, S, although we were debating wether or not J, K, or something like that, but you went with the W, S, and that worked fine. So yeah, that looks good, and you got the A through F going in there deeper into the…

While observing participants complete the tasks, we noticed most of them had no issues using the WASD keys to navigate around the platform. Most participants were so fast with the keyboard shortcuts and some were even impatient to wait for the screen reader to finish speaking before pressing the next key. It was like they had used the system continuously for a long time.

Some participants attributed this ease of use of the WASD keys to the fact that gamers commonly use them. The pervasiveness of the WASD among gamers was one of the main reasons for making this design choice and also that they allow users to interact with the system one-handed. P4 describing his experience with the keyboard shortcuts noted:

> So I liked them because it's like kind of a gaming system. I don't play a lot of games, but I think the problem I run into is I don't think D as right, and if I got used to D as right, because that's what it is, A S D, you know, then all of it makes sense.

P6 corroborated the opinion of P4 as illustrated by the following quote:

> I mean these keys are fine for the gamers like myself. Because frequently used in games video games.

Although most participants had no problem with the WASD as the primary navigation keys, some of the participants wondered why the arrow keys were not used instead. The main reason for not using the arrow keys is to avoid conflict with existing applications. Some of these participants gave suggestions on improving the keyboard navigation experience. These are discussed in the next section.

6.5.2 Reflections on Improving the Accessibility of BBLs

In the following subsections, we report the remaining results that were obtained after the inductive analysis of the qualitative data. Most, if not all, of the participants found the system to be accessible and easy to use to create and edit block-based code. Participants reported no accessibility issues with the system. The participants were more enthusiastic about the efforts made to improve the accessibility of block-based programming environments to people with visual impairments. The qualitative data from the open-ended discussions mostly were suggestions to add more features to the system and improve the user experience as illustrated by the codes and themes from the inductive analysis steps.

6.5.2.1 Provide Feedback at Different Levels of Verbosity

Although most participants said the screen reader feedback was enough. They also reported that they could benefit from more output from the screen reader depending on the task at hand and their familiarity with the system. Currently, the screen reader would output information about the focused block only. Six participants hinted that in some scenarios such as with more extensive programs, they would like to get more contextual information about the blocks surrounding the focused block. For example, reading a container block and its child blocks with a single keystroke. P5 reported:

> ...Ummm Like I could see in blockly you know, maybe have an option of I want more feedback. You know kind of more to less feedback where say if i'm you know, maybe i'm in ummm You know, a nested conditional or you know within a within a conditional within a loop right maybe I want to know more, the context You know, but maybe I don't.

On the other hand, the user would not always like to listen to a verbose output from the screen reader all the time especially when they become familiar with the system or know what they are doing. P2, talking about the verbosity of the screen reader output noted:

> Maybe when adding a block, the feedback was not exactly enough for me. So, but yeah, at the same time I don't want it to speak out a lot of things so that, like, I have to listen to a lot, there will be too much of audible feedback. So, I was mostly happy.

Our data reveal that participants have subjective opinions about the right amount of screen reader feedback to provide. However, data from five participants suggests that a middle ground can be found by leveraging the screen reader to speak at different levels of verbosity controlled by the user. Some participants took the example of the screen reader that operates at different levels of verbosity based on the user's level of experience.

Therefore, we believe researchers and practitioners should incorporate different levels and amounts of output in their design of accessible block-based coding environments to allow users to select the level that works best for them at any given point.

6.5.2.2 Design a Simplified Interaction Model

There are currently no publications that describe the mental model that people with visual impairments have about block-based coding. Visually, block-based code is spatial with the code growing top-down and left-right. However, our qualitative data suggests that people with visual impairments have a simplified view of block-based code that differs from the spatial visual model. The keyboard interaction model in Accessible Blockly has been designed to closely mimic spatial navigation in which top-down, left-right, in-line and diagonal (in case moving into the body of a container block) navigations are designed to be distinct and convey the sense of spatiality in the nature of the code.

However, to some participants (N = 5), they regard block-based coding as navigating through different sets of lists and selecting options or adding items to the list. For example, they viewed going through the connection points in edit mode as a list, going through the properties of a block as a list of options, going through the blocks vertically as a list of options as well, etc. Therefore, these participants thought it was less confusing to perceive the system in this simplified view. This became more apparent when some participants (N = 3) talked about using the arrow keys to navigate through the blocks of code and connection points as you would with an HTML menu. Prior experience navigating HTML menus and other applications with the screen reader might have influenced the participants' internal perception of block-based code. P3 explicitly noted:

> Overall I would say that ummm well visually there is a lot of distinguishment between umm adding a block within a block or adding a value within one of the connections on a block is very distinct I don't necessarily feel that that needs to be the case for umm for umm for interaction models so what I would say is you could instead of, for example, having different keys for different sibling inputs and different connections, you could literally just have the arrow keys do that and just indicate okay you've just moved the block now you've moved the value now you've move the this.

This simplistic view might also be due to the participants unfamiliarity with block-based code as illustrated by this quote from P8:

> I found them confusing, a little bit. Just because I expect to navigate menus through right, left, up, down, kind of structure, and then found the whole concept of what's a container, and what's a variable with a field, to be, to me, it feels inconsistent, although I understand the logic that you're using.

On the other hand, to some participants, this simplistic view as a list of options might be confusing especially when the same sets of keys are used to navigate in different contexts. As stated earlier, almost all the participants found that the design of Accessible Blockly supports ease of use and learning. However, designers and researchers should take into consideration the mental model that people with visual impairments have about block-based coding while designing accessible solutions.

6.5.2.3 Design to Maximize Transfer of Knowledge

Another common theme that emerged from the inductive analysis was the ability to reuse already acquired knowledge or skills. The report from four participants suggests that wherever applicable the system should adopt common keyboarding techniques or screen reader features already being used in similar scenarios in other applications. A typical example cited by some participants was keeping the toolbox as a tree view and allowing users to navigate through it with the arrow keys as in an HTML menu.

6.5.2.4 Provide Search Functionality

Although some participants described the system as fast, they wanted a quicker way to find blocks within the toolbox. Accessible Blockly currently does not support search. Participants thought the process of creating block-based code would be faster if instead of going through the blocks within the categories to find a block, they could search for the block after selecting a connection point on the workspace. This would speed up the process of adding a new block to the workspace and reduce the time it takes to create a program. P11 explicitly reported:

> I think it is quite interesting. I would prefer if there was a search box so that I can search the entire toolbox and search among those blocks so that I can search, for example, I can input a print, so I can directly select from the search result instead of going through the text [Text category] and going down to find the print.

Another reason that motivated the request for a search functionality is discoverability. Some participants reported that new users of the platform do not know all the types of blocks available on the system and to which category a block belongs. Therefore, a search functionality would minimize the time it takes for a new user to locate a particular block within the toolbox. It is essential as part of providing a seamless user experience for researchers and developers to provide search functionalities and similar shortcuts that would speed up the interaction process of users on the platform.

6.5.2.5 Use Familiar and Consistent Terminology

Half of the participants (n=6) had different opinions about some of the terminologies used to describe blocks or attributes of blocks in Accessible Blockly. Some participants were trying to map the terminologies to their knowledge of text-based programming. For example, some participants misinterpreted the *"do connection"* of the *"if"* block as illustrated by the following quote from P6:

> For example, even earlier when it said, do I wanted to go inside the do to put the function because because of the previous knowledge. I think if someone doesn't have the prior knowledge to programming probably it wouldn't be a concern actually.

The terminologies used are based on the visual information that a sighted user would get looking at a block. The body of most container blocks in the Blockly library are labeled as "do." Keeping this consistency with the visual blocks would facilitate collaboration and communication between sighted students and students with visual impairments working on the same activities.

Although the issues with unfamiliar terminology can easily be overcome with experience and the use of an extensive user manual, what is more important is keeping the terminology consistent throughout. P4 did not understand why the body of the *"else"* block, a container block, was described as *"else input"* rather than "do connection" as it is for other container blocks. P4 talking about the unfamiliar terminologies explained:

> Okay, and of course, like the cleanups that you're doing, like getting them all to say do, or you know body instead of do, or you know because I don't know if do is, I guess do is fine for an if statement. Some languages call it if-then, some call it do, some call it just the body of the if. That was a little confusing, when I got the else and it said inputs. Maybe body will do. But other than that, I think you guys are on the right track.

As highlighted by P4 in the previous quote, different languages use different terminologies. Therefore, we believe what is most important is keeping the terminology consistent across the platform as well as conveying the same or similar visual information that a sighted user would typically get by looking at the blocks.

6.5.2.6 Provide Readily Available Help and User Manual

As reported by the TVIs in Chapter 4, one of the main features that make Swift Playgrounds attractive is the availability of accessible user manuals that help the students easily find their way around the system. Three participants suggested that novices would benefit from readily available help or guidance on the system. One participant suggested providing instructions to novices with the feedback from the screen reader. P8 explained:

> So, if you pick the beginner mode, and you hit the alt key, it'll say "file menu, alt-f," so that it'll tell you what the keyboard shortcut is, so you would know to use that in the future. So, it might be interesting to have a beginner mode where, when you entered the toolbox, it would say, you know, "you are in the toolbox, use these keys to go down, hit this key to go into…" that, you context sensitive keyboard help. One of those two things would probably make this experience much better. And as you became advanced, you could shut those announcements off and just move around because now I know my keys and I can, you know.

6.5.2.7 Avoid Glitches on the System

Occasional bugs in a system can confuse users and lead to frustration. And this is one of the challenges that TVIs reported in Chapter 4. There were some minor bugs that were not caught during the pilot testing before the evaluation study. However, these had less impact on the user experience as they did not affect the functionality of the system. Unlike sighted users who can easily understand and overcome some glitches on a system, it can be hard for users who rely on screen readers because any inconsistency from what they expect can easily throw them off. Therefore, practitioners should thoroughly test the system to identify any bug that can be an obstacle to the effective use of the system by target users.

6.6 Limitations

One of the limitations of this work is that the evaluation was conducted with professional programmers with good coding backgrounds. Block-based programming is mainly designed for novice students. Therefore, it would be interesting to evaluate the system with K-12 students with visual impairments. Another limitation is the number of participants in the study. Although the number is comparable to those in similar studies in this domain [25][39], we believe it is small for more generalizable results. However, the study offers a first look at the actual experiences of people with visual impairments coding block-based programs with a keyboard and screen reader alone.

6.7 Future Work

As part of future work, we plan to continue the user evaluations with more participants and K-12 students. We have already started incorporating some of the suggestions from this study into Accessible Blockly and plan to continue in this direction. Lastly, we plan to conduct a comparative evaluation study in which users will create and edit block-based programs using Accessible Blockly and the Blockly alternate navigation scheme discussed in Chapter 5.

6.8 Conclusion

In this chapter, we have presented the updates made to Accessible Blockly based on the results of the study discussed in Chapter 5. We have also discussed an evaluation of Accessible Blockly for block-based code creation and editing with 11 blind programmers. The results show that people with visual impairments can effectively use the keyboard and the screen reader to create, edit and navigate block-based code within acceptable time limits. The participants found no accessibility issues with the system and offered suggestions to add more features to the system that would improve the overall user experience of the target users.

CHAPTER 7

CONCLUSION, MERITS, IMPACTS AND FUTURE WORKS

In this dissertation, we have discussed the challenges people with visual impairments face in their day-to-day use of block-based programming environments. We have also proposed an Accessible block-based programming library designed to work with the keyboard and the screen reader. We have discussed the evaluation of the accessibility library in multiple studies with programmers with visual impairments and the improvements made to the library based on the feedback from the users.

In Chapter 3, we discussed the results of a systematic literature review conducted to gather insights and data on the experiences of people with visual impairments on block-based programming and text-based programming environments. The results showed that there were no block-based programming environments accessible to blind people, except Blocks4All[4] which was a work in progress.

We then moved from this purely theoretical approach to understanding the experiences of people with visual impairments on block-based programming environments to a more practical approach in Chapter 4. In this chapter, we conducted a series of empirical studies, including surveys and interviews with teachers of students with visual impairments (TVIs) and students with visual impairments. This investigation showed that students with visual impairments did not use block-based programming environments mainly because they are inaccessible. We also discovered that what teachers used for their students is Swift Playgrounds, a hybrid of block-based and text-based programming. We also understood that students faced several challenges on Swift Playgrounds, including difficulties creating, editing, and navigating block-based code. We also explained why these actions were challenging to the students. We also learned how the TVIs mitigate these challenges and what they use with their students in the absence of accessible block-based programming environments.

In Chapter 5 and Chapter 6, we introduced Accessible Blockly, our solutions to the inaccessibility issues and challenges faced by people with visual impairments on block-based programming environments. Accessible Blockly is designed to work with the screen reader and keyboard. It allows users to create, edit, navigate, and understand block-based code using the keyboard and screen reader alone. Accessible Blockly was evaluated with programming with visual impairments as discussed in Chapter 5 and Chapter 6. The participants found no accessibility issues with Accessible Blockly and were happy to create and edit block-based code using the screen reader and keyboard alone. Chapter 6 also offers guidelines for researchers and practitioners to consider when designing accessible block-based programming environments.

7.1 Merits

This dissertation contains several merits. First, the research provides a comprehensive look at the reality of the instructional usage of block-based programming environments by TVIs and students with visual impairments. It discusses in detail the challenges that students with visual impairments face on current block-based programming environments and the root causes of these barriers. Second, the dissertation offers insights into how TVIs compensate for the lack of accessible block-based programming environments in a situation where block-based programming is increasingly present in K-12 curricula and computing outreach activities. Third, the research proposes an accessible block-based programming library designed with people with visual impairments in mind. This library was evaluated with programmers with visual impairments. The results show that people with visual impairments can effectively create, edit and navigate block-based code using the screen reader and keyboard alone. Finally, the empirical data from the evaluation studies provide guidelines for researchers and practitioners designing block-based programming environments or similar platforms.

7.2 Impacts

1. Researchers and practitioners would better understand the accessibility barriers people with visual impairments face on block-based programming and hybrid environments.
2. The research community and TVIs at large can learn from and use the strategies reported by TVIs in our study to compensate for the lack of accessible block-based programming environments.
3. Researchers and practitioners can use our approach to designing keyboard and screen reader interactions to improve the accessibility of block-based programming and hybrid environments. The Blockly alternate navigation discussed in Chapter 5 is a fruit of our collaboration with the Blockly team at Google.
4. Accessible Blockly is open-source and available to everyone in the community.
5. Accessible Blockly can be used to build accessible block-based programming environments that would be usable by sighted and people with visual impairments alike.

7.3 Future Work

As part of future work, we plan to continue improving the accessibility of the system by providing more output and alerts through the screen reader module. This involves exploring how audio cues can be used to supplement the information or feedback a user receives when interacting with the system. We also plan on implementing some of the suggestions from our empirical studies regarding keyboard navigation shortcuts.

Another direction of future work is designing and implementing an accessible block-based compiler and debugger to allow users to run their code and examine the output. This would eventually grow into an AI-enabled platform in which Artificial Intelligence and Machine Learning shall be leveraged to provide an intelligent platform on which students with visual impairments can learn and practice coding independently.

APPENDIX A

ACCESSIBILITY CHALLENGES REPORTED BY PAPER

Publication	Challenges Investigated
Mealin2012 [17]	Code navigation, code comprehension, code editing, code skimming
Albusays2017[16]	Code navigation, code comprehension, code skimming
Albusays2016[28]	Code navigation, Code debugging
Smith2003[30]	Navigation (program structure)
Huff2020[31]	Code Navigation
Potluri2018[33]	Code navigation, code comprehension, code editing, code debugging
Ludi2015[14]	Code navigation
Utreras2020[34]	Code Navigation
Mountapmbeme2020[35]	Code Navigation, Code Editing
Armaly2018[37]	Code Comprehension
Stefik2009 [38]	Code debugging
Stefik2011 [5]	Code debugging

APPENDIX B

SURVEY QUESTIONS FOR TEACHERS OF STUDENTS WITH VISUAL IMPAIRMENTS

1. What is your gender?
2. Type of school where you are employed in
3. What assistive tools do you use when programming or teaching programming? (Select all that apply)
    a. magnification software
    b. screen reader
    c. Braille display
    d. Other (please specify)
4. How long have you been teaching a Block based programming language/environment? (Select one)
    a. Never
    b. Less than 1 year
    c. Between 1 and 2 years
    d. Between 2 and 4 years
    e. More than 4 years
5. In what context do you teach block-based programming? (Select all that apply)
    a. In school
    b. An after-school program or club
    c. A summer workshop
    d. Hour of Code activity
    e. Other (please specify)
6. What grade(s) are the students who you are teaching to program?
    a. Kindergarten - 2nd grade
    b. 3rd - 5th grade
    c. Middle School
    d. High School
    e. Other (please specify)
7. What is the name of the Block-based language you teach?

8. On what platform(s) do you teach block-based programming? (Select all that apply)
    a. Desktop or laptop computer
    b. Tablet
    c. Phone
    d. Other (please specify)
9. What challenges did students in general face in identifying blocks from the toolbox?
10. What challenges did students in general face in moving a block from the toolbox to the workspace?
11. What challenges did students in general face in working with blocks on the workspace?
12. What challenges did students in general face in understanding the relationships between blocks or the logical flow of blocks in an existing program?
13. Have you taught Block-based programming to visually impaired students before?
14. Describe any other challenges that the students faced?
15. What is the approximate number of visually impaired that you have taught to program using block-based programming?
16. Please rate each of the aspects of block-based programming on a scale of 1 to 5, where 1 is Very Difficult and 5 is Very Easy from the perspective of your students with visual impairments.
    a. Identifying a specific block in the toolbox.
    b. Taking a block from the toolbox and placing it in the workspace
    c. Placing a block on the workspace with existing blocks
    d. Understanding how blocks are connected to one another in an existing program. (
    e. Finding the insertion point where a block will be place before/after blocks on the workspace.
17. What challenges do your VI students face in regard to identifying a specific block on the toolbox?
18. What challenges do your VI students face in regard to moving a block from the toolbox and placing it in the workspace?
19. What challenges do your VI students face in regard to placing a block on the workspace with existing blocks?

20. What challenges do your VI students face in regard to understanding the relationship between blocks or the logical flow of blocks in an existing program?
21. What other challenges do your VI students face while using the Block-based language?
22. What features or improvements in Block-based programming environment are needed that if provided will make things easier and better for your VI students?
23. What accessibility features did the Block-based language have? e.g audio feedback
24. Did you use any workarounds to help your VI students in performing the tasks referred to in Question 17 (e.g., locating and placing blocks, understanding programs? If so, what are the workarounds and how well do they work?
25. How do you compare the complexity of programs written by VI students to that written by their sighted peers?
26. What assistive tools did VI students use in your class? (Select all that apply)
27. What are your suggestions or comments on how to make Block-based languages accessible to VI students?
28. What text-based programming languages have you taught? For example Java, C, java script
29. What programming tools do you use to teach the programming languages you just identified in the previous question?
30. On a scale of 1 to 5, 1 being not supported and 5 being fully supported, indicate by how much your VI students' assistive technology is supported by the programming tools listed in previous question.
31. What challenges do your VI students face when they try to read and understand a program which was written someone else?
32. What challenges do your VI students face when they are writing their own programs?
33. What are your suggestions to help make text-based programming and their environments accessible to VI learners?
34. What real world objects or metaphors would you use to explain an "if block" to a VI student?
35. What real world objects or metaphors would you use to explain a "while loop block" to a VI student?
36. What real world objects or metaphors would you use to explain a "for loop block" to a VI student?
37. What real world objects or metaphors would you use to explain a "function or method" to a VI student?

APPENDIX C

INTERVIEW QUESTIONS WITH TEACHERS OF STUDENTS WITH VISUAL IMPAIRMENTS

TVI's background

1. How many years have you been teaching?

2. How many years have you been teaching VI students?

3. What subjects and grades do you teach?

4. What is your level of experience in programming?

5. What is your level of experience teaching programming or related subjects?

6. What languages do you feel comfortable teaching? Are these the same ones you are teaching?

Students Background

7. On average, how many students with VI do you teach?

8. If a CS teacher in a mainstream class, how many VI students have you taught in the past three years?

   For those who teach/have taught VI students

9. What is the age range of your students?

10. What is the level of the students' vision? E.g. no vision, some vision, partially sighted

11. What percentage of your students use a screen reader to interact with programming software? Use magnification software? A Braille display?

Coding Environment

12. What software/languages do you use to teach programming to your students?

13. Do you have a technology preference with your students? If so, what is it?

If it is an iPad

14. Why do you prefer the iPad for your students? Is it because VoiceOver is easy to use? Or because the students have iPad? Or Because SP is just the most accessible platform and runs on iPad?

15. Do the students have prior experience with using an iPad? What type of applications do they interact with?

16. Occasional software glitches have been cited as an obstacle to the students. Can you name some examples of this software failures? Does it have to do with the screen reader or the coding environment? How do the students react to such error?

Programming Challenges

If not using iPad

Describe the challenges you have with the tool you use?

Note: this will lead to follow-up questions about editing, feedback, the learning process

Assuming using an iPad

17. Describe the challenges your students have with SP

18. Do your students have any dexterity/fine motor issues with performing the gestures on iPad? Are the gestures complex to perform or too similar such that they can be confused and difficult to remember? How does this issue vary across age? Is it more common with younger students than with older ones?

19. Editing an existing program on SP has been described as a difficult task for students when using VoiceOver. Have you encountered similar issues? Can you provide some examples?

    a. They say it is because of the advanced commands required to perform this task. How does editing vary with the size and complexity of the program? Are the associated gestures complex to perform or is the sequence (copy, cut and paste using the rotor) too long to be easily remembered? Do you think it can be done in another less complex way?

20. How do you feel about having an accessible BBL vs an accessible TBL for your students?

APPENDIX D

SURVEY QUESTIONS FOR STUDENTS WITH VISUAL IMPAIRMENTS

1. What is your child's age?
2. Please select your child's gender
    a. Female
    b. Male
    c. Other
    d. Prefer not to answer
3. Type of school your child attends
    a. Public School
    b. Private School
    c. Homeschooled
    d. State school for the blind/visually impaired
    e. Other (please specify)
4. What assistive tools does your child use when interacting with a computer? (Select all that apply)
    a. magnification software
    b. screen reader
    c. Braille display
    d. None
    e. Other (please specify)
5. On a scale of 1 to 5, 1 being not skilled and 5 being fully skilled, indicate the degree of your child's skills with the following:
    a. Using their assistive technology.
    b. Using the keyboard or touchscreen interactions to enter information.
    c. Using their primary computing device (e.g., a PC, laptop, tablet, smart phone).
    d. Interacting with websites or smartphone applications.
6. Did your child have prior exposure to computing and computing skills before starting to learn programming? This includes using a PC, tablet, or smartphone (whichever is their primary device for computing).

    a. Yes
    b. No

7. On a scale of 1 to 5, 1 being not skilled and 5 fully skilled, indicate the degree of your child's Orientation and Mobility (O&M) skills.
8. Has your child heard of Block-based programming before? For example, Scratch or Blockly. Block-based programming refers to creating a program by dragging commands rather than typing them.
    a. Yes,
    b. No
9. What is the name of the Block-based language your child uses/has used?
10. How long have your child been using a Block-based programming language?
    a. Never
    b. Less than 1 year
    c. Between 1 and 2 years
    d. More than 2 years
11. On a scale of 1 to 5, 1 being not supported and 5 being fully supported, indicate by how much your child's assistive technology is supported by the Block-based programming your child uses/has used.
12. On a scale of 1 to 5, 1 being not knowledgeable and 5 being expert, rate the level of technical knowledge of your child's instructor with respect to programming knowledge. If you are not sure, select Not Sure
13. Has your child learned block-based programming (e.g., Scratch, AppInventor)?
    a. Yes
    b. No
14. Please rate each of the following aspects of block-based programming on a scale of 1 to 5, where 1 is Very Difficult and 5 is Very Easy.
    a. Identifying a specific block in the toolbox.
    b. Taking a block from the toolbox and placing it in the workspace.
    c. Placing a block on the workspace with existing blocks.
    d. Understanding how blocks are connected to one another in an existing program.

   e. Editing an existing program on a workspace? For example, replacing or deleting a block to correct an error on a program.

15. What challenges does your child face in regard to identifying a specific block on the toolbox?

16. What challenges does your child face in regard to moving a block from the toolbox and placing it in the workspace?

17. What challenges does your child face in regard to placing a block on the workspace with existing blocks?

18. What challenges has your child faced in regard to editing an existing program on the workspace?

19. What challenges has your child faced in regard to understanding how blocks are connected to one another in an existing program?

20. What features or improvements in Block-based programming environment are needed that if provided will make things easier and better for your child?

21. Select the actions your child takes when faced with a challenge while using the Block-based programming environment. (Select all that apply)

    a. I seek help from an instructor

    b. I look up the answer on the internet

    c. I seek help from a sighted friend

    d. Other (please specify)

22. Where did your child learn block-based programming? (Select all that apply)

    a. In school

    b. In a summer camp or workshop

    c. At home

    d. Other (please specify)

23. On what platform does your child do block-based programming? (Select all that apply)

    a. Desktop or laptop computer

    b. Tablet

    c. Phone

    d. Other (please specify)

24. Has your child learned to program with a text-based language? (e.g., Python, HTML, or other languages that are text-based)

    a. Yes

    b. No

25. What text-based programming languages does your child use? For example, Java, C, java script

26. What programming tools do your child use to write programs in the programming languages listed in the previous question?

27. What challenges do you face when writing programs in the programming languages your child listed in Question 20?

28. On a scale of 1 to 5, 1 being not supported and 5 being fully supported, indicate by how much your child's assistive technology is supported by the programming tools listed in previous question.

29. What challenges does your child face when they try to read and understand a program which was written someone else?

30. What is missing in the programming languages or tools your child listed that if provided will make coding easier?

APPENDIX E

EVALUATING ACCESSIBLE BLOCKLY FOR CODE NAVIGATION AND COMPREHENSION

This study evaluates the use of keyboard inputs to navigate existing block-based code on the workspace. The study consists of two experiments. Each experiment contains 3 tasks.

Link to all task: https://mountapmbeme.com/study/home.html

Second Link with Phases alternated: https://mountapmbeme.com/study/home-b.html

Phase 1: Using Accessible Blockly Keyboard Navigation Scheme

Task 0: Training

W- traverse up to previous statement block

S – traverse down to next statement block

D – traverse into container blocks | or blocks at same level

A – traverse out to parent block | or traverse to the left

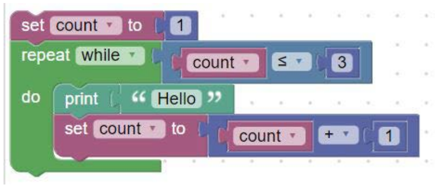
F – traverse to the right into inline statements (value blocks)

J – jump to first block on workspace

R – repeat current location

The following program will be used to teach the basic navigation keys for about 15 mins

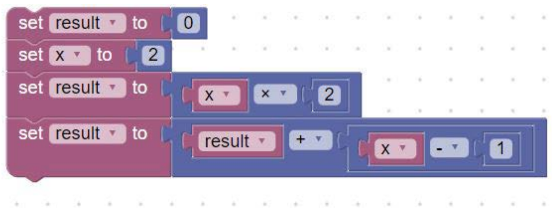
https://mountapmbeme.com/study/accessible/blockly/accessibleblockly/study/training.html



```
set count to 1
repeat while count ≤ 3
do  print " Hello "
    set count to count + 1
```

Task 1: What is the value of the variable named *result* after the following program is executed? Use the Keyboard and Screen Reader to read the program.

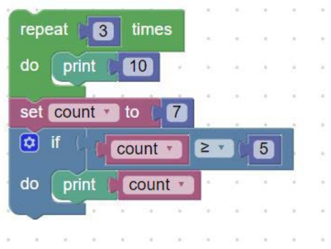
https://mountapmbeme.com/study/accessible/blockly/accessibleblockly/study/task1.html



```
set result to 0
set x to 2
set result to x × 2
set result to result + x - 1
```

Task 2: What is the output of the following program? Use the keyboard and screen reader to navigate and read the program.

https://mountapmbeme.com/study/accessible/blockly/accessibleblockly/study/task2.html



```
repeat 3 times
do  print 10
set count to 7
if  count ≥ 5
do  print count
```

How will the output change if the value of count was 2?

Post study questions

8. On a scale of 1 to 5, 1 being very hard and 5 being very easy, rate how easy the tasks were to complete.
9. On a scale of 1 to 5, 1 being very frustrating and 5 being not frustrating at all, rate how frustrating the tasks were to complete.
10. On a scale of 1 to 5, 1 being I had no idea where I was and 5 being I always knew where I was, rate how well you know where you were in the code

Phase 2: Using Google's Keyboard Navigation scheme

Task 0: Training

W- traverse up to previous statement block or to top connection of first block on stack | traverse right through connection points block.

S – traverse down to next statement block or to bottom connection of last block on stack

D – traverse right into first connection point of block | traverse right or into connecting block from a connection point

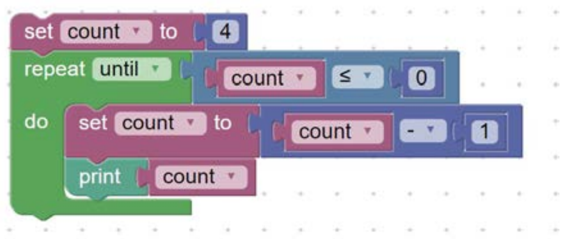
A – traverse left into connection point of parent block | traverse left or out to parent block from connection point. If at the outermost blocks, takes you to top connection of first block

J – jump to first block on workspace

R – repeat current location

The following program will be used to teach the basic navigation keys for about 15 mins

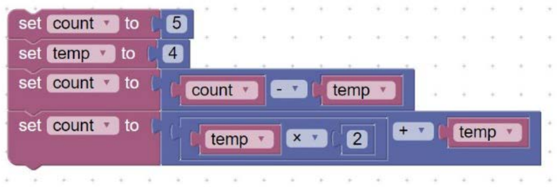
https://mountapmbeme.com/googles/blockly/accessibleblockly/study/training.html



```
set count to 4
repeat until count ≤ 0
do  set count to count - 1
    print count
```

Task 1: What is the value of the variable named *count* after the following program is executed? Use the Keyboard and Screen Reader to read the program.

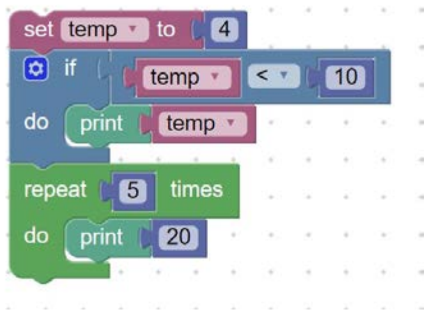
https://mountapmbeme.com/googles/blockly/accessibleblockly/study/taska.html



```
set count to 5
set temp to 4
set count to count - temp
set count to temp × 2 + temp
```

Task 2: What is the output of the following program? Use the keyboard and screen reader to navigate and read the program.

https://mountapmbeme.com/googles/blockly/accessibleblockly/study/taskb.html



```
set temp to 4
if temp < 10
do  print temp
repeat 5 times
do  print 20
```

What happens to the output if the value of temp is 20?

Post study questions

1. On a scale of 1 to 5, 1 being very hard and 5 being very easy, rate how easy the tasks were to complete.
2. On a scale of 1 to 5, 1 being very frustrating and 5 being not frustrating at all, rate how frustrating the tasks were to complete.
3. On a scale of 1 to 5, 1 being I had no idea where I was and 5 being I always knew where I was, rate how well you know where you were in the code

Open-Ended Questions

How do you compare your experience using both navigation schemes?

Do you have a preference for any? If yes, why?

What other suggestions do you have in regards to improving block-based programming accessibility?

APPENDIX F

EVALUATING ACCESSIBLE BLOCKLY FOR BLOCK-BASED CODE CREATION AND EDITING

Link to all tasks: https://mountapmbeme.com/study/home2.html

Task 0: Training

W- traverse up to previous statement block

S – traverse down to next statement block

D – traverse into container blocks | or blocks at same level

A – traverse out to parent block | or traverse to the left

F – traverse to the right into inline statements (value blocks)

J – jump to first block on workspace

C – Open toolbox

R – repeat current location

I – traverse sibling inputs to right

Shift-I – reverse traverse sibling input to left

Delete key – delete a block

Ctrl + Z – Undo previous action

The following program will be used to teach the basic navigation keys for about 20 mins https://mountapmbeme.com/study/accessible/blockly/accessibleblockly/study2/training.html

Describe block morphology to the participant. Use the following steps to guide the participant build a program.

Press J to get to top block on workspace

Add number block to "set count to" block

Edit number to 1

Add repeat N times block

Add number block to repeat n times block

Edit added number block to 5

Add print block inside repeat 5 times block

Add text block to print block

Update text of text block to “welcome”

Go outside repeat 5 times block

Add if block below repeat five times block

Add “A = B” block to Boolean connection of repeat block

Add variable count to left side of “A=B” block

Add number to right side of A=B block

Let the user navigate the blocks

Go to if block

Delete if block

Tasks

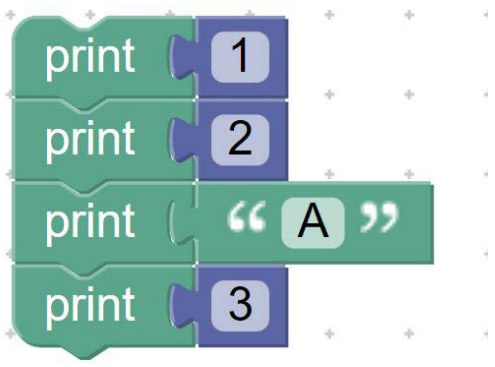
Use a timer for each of the following tasks.

1) Write a program that prints “hello” 3 times using the repeat N times block. You have 10 mins to complete the task.

   https://mountapmbeme.com/study/accessible/blockly/accessibleblockly/study2/task1.html

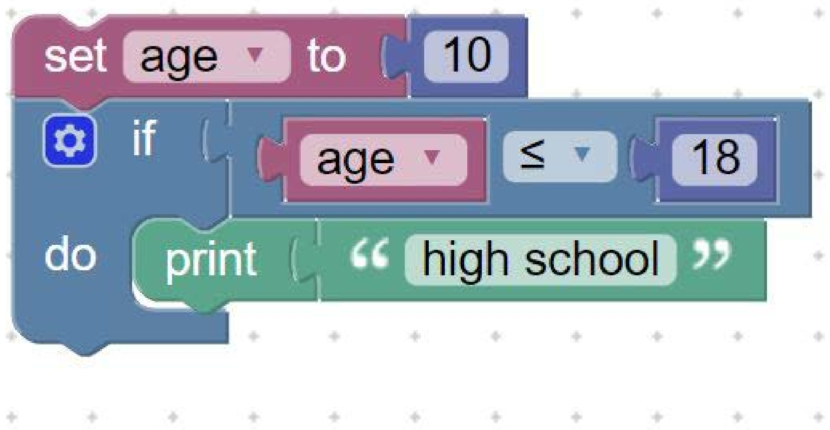
2) The following program prints 1 2 A 3. Edit the program to only print 1 2 3. You have 10 mins to complete the task.

   https://mountapmbeme.com/study/accessible/blockly/accessibleblockly/study2/task2.html


```
print 1
print 2
print “A”
print 3
```

3) The following program contains a variable named age. The program prints "High School" if the value of age is less than or equal to 18. Complete the program to also print "college" if age is greater than 18. You have 10 mins to complete the task.

https://mountapmbeme.com/study/accessible/blockly/accessibleblockly/study2/task3.html



Post study questions:

1. On a scale of 1 to 5, 1 being very hard and 5 being very easy, rate how easy the tasks were to complete.
2. On a scale of 1 to 5, 1 being very frustrating and 5 being not frustrating at all, rate how frustrating the tasks were to complete.
3. On a scale of 1 to 5, 1 being very hard and 5 being very easy, rate how easy it was for you to find an insertion point on the workspace at which to insert a block
4. On a scale of 1 to 5, 1 being very hard and 5 being very easy, rate how easy it was for you to find a block from the toolbox
5. On a scale of 1 to 5, 1 being very hard and 5 being very easy, rate how easy it was for you to add a block to the workspace
6. On a scale of 1 to 5, 1 being I had no idea where I was and 5 being I always knew where I was, rate how well you know where you were in the code.
7. On a scale of 1 to 5, 1 being very hard and 5 being very easy, rate how easy it was for you to delete a block on the workspace.

Open-ended questions:

How would you describe your overall experience writing and editing block-based code?

How would you describe the amount of feedback provided by the screen reader? Was it enough?

How would you describe the keyboard shortcuts?

What do you think could be improved on the current system?

REFERENCES

[1] D. Weintrop, “Block-based programming in computer science education,” *Commun. ACM*, vol. 62, no. 8, pp. 22–25, Jul. 2019, doi: 10.1145/3341221.

[2] D. Weintrop and U. Wilensky, “To block or not to block, that is the question: students’ perceptions of blocks-based programming,” in *Proceedings of the 14th international conference on interaction design and children*, 2015, pp. 199–208.

[3] S. Grover, S. Basu, M. Bienkowski, M. Eagle, N. Diana, and J. Stamper, “A framework for using hypothesis-driven approaches to support data-driven learning analytics in measuring computational thinking in block-based programming environments,” *ACM Trans. Comput. Educ.*, vol. 17, no. 3, pp. 1–25, 2017.

[4] L. R. Milne and R. E. Ladner, “Blocks4All: overcoming accessibility barriers to blocks programming for children with visual impairments,” in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–10.

[5] A. Stefik, C. Hundhausen, and D. Smith, “On the design of an educational infrastructure for the blind and visually impaired in computer science,” in *SIGCSE’11 - Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, 2011, pp. 571–576, doi: 10.1145/1953163.1953323.

[6] “Statistics About Children and Youth with Vision Loss | American Foundation for the Blind.” https://www.afb.org/research-and-initiatives/statistics/statistics-blind-children#population16 (accessed Sep. 14, 2020).

[7] “Goal 4 | Department of Economic and Social Affairs.” https://sdgs.un.org/goals/goal4 (accessed Sep. 14, 2020).

[8] L. R. Milne and R. E. Ladner, “Position: Accessible Block-Based Programming: Why and How,” in *Proceedings - 2019 IEEE Blocks and Beyond Workshop, B and B 2019*, Oct. 2019, pp. 19–22, doi: 10.1109/BB48857.2019.8941230.

[9] A. Stefik, R. E. Ladner, W. Allee, and S. Mealin, “Computer Science Principles for Teachers of Blind and Visually Impaired Students,” in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 2019, pp. 766–772.

[10] “Screen Magnification Systems | American Foundation for the Blind.” https://www.afb.org/node/16207/screen-magnification-systems (accessed Jul. 04, 2022).

[11] “Virtual Magnifying Glass - Wikipedia.” https://en.wikipedia.org/wiki/Virtual_Magnifying_Glass (accessed Jul. 13, 2022).

[12] “Refreshable Braille Displays | American Foundation for the Blind.” https://www.afb.org/node/16207/refreshable-braille-displays (accessed Jul. 04, 2022).

[13] “Orbit Reader 20 – Braille Display, Book Reader and Note-taker. Includes an SD Card, Charger and a USB cable – Orbit Research.” http://www.orbitresearch.com/product/orbit-reader-20/ (accessed Jul. 04, 2022).

[14] S. Ludi, “Position paper: Towards making block-based programming accessible for blind users,” in *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, 2015, pp. 67–69.

[15] Mountapmbeme Aboubakar, Okafor Obianuju, and Ludi Stephanie, “Addressing Accessibility Barriers in Programming for People with Visual Impairments: A Literature Review,” *ACM Trans. Access. Comput.*, vol. 15, no. 1, pp. 1–26, Mar. 2022, doi: 10.1145/3507469.

[16] K. Albusays, S. Ludi, and M. Huenerfauth, “Interviews and Observation of Blind Software Developers at Work to Understand Code Navigation Challenges,” in *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility*, 2017, pp. 91–100.

[17] S. Mealin and E. Murphy-Hill, “An exploratory study of blind software developers,” in *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2012, pp. 71–74.

[18] “Stack Overflow Developer Survey 2020.” https://insights.stackoverflow.com/survey/2020 (accessed Dec. 14, 2020).

[19] “Welcome to the CS10K Community! | CS10K Community.” .

[20] A. Stefik and R. E. Ladner, “Introduction to AccessCS10K and accessible tools for teaching programming,” *SIGCSE 2015 - Proc. 46th ACM Tech. Symp. Comput. Sci. Educ.*, pp. 518–519, Feb. 2015, doi: 10.1145/2676723.2677321.

[21] “Learn | Code.org.” https://code.org/learn (accessed Sep. 19, 2020).

[22] “Scratch - Imagine, Program, Share.” https://scratch.mit.edu/ (accessed Apr. 27, 2020).

[23] “Blockly | Google Developers.” https://developers.google.com/blockly (accessed Apr. 09, 2022).

[24] “Pencil Code.” https://pencilcode.net/ (accessed Dec. 13, 2020).

[25] C. M. Baker, L. R. Milne, and R. E. Ladner, “Structjumper: A tool to help blind programmers navigate and understand the structure of code,” in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, 2015, pp. 3043–3052.

[26] A. Hadwen-Bennett, S. Sentance, and C. Morrison, “Making Programming Accessible to Learners with Visual Impairments: A Literature Review.,” *Int. J. Comput. Sci. Educ. Sch.*, vol. 2, no. 2, p. n2, 2018.

[27] "Guidelines for performing Systematic Literature Reviews in Software Engineering," 2007.

[28] K. Albusays and S. Ludi, "Eliciting programming challenges faced by developers with visual impairments: exploratory study," in *Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering*, 2016, pp. 82–85.

[29] K. Albusays, "The Role of Sonification as a Code Navigation Aid: Improving Programming Structure Readability and Understandability For Non-Visual Users," *Theses*, Nov. 2020, Accessed: Mar. 08, 2021. [Online]. Available: https://scholarworks.rit.edu/theses/10672.

[30] A. C. Smith, J. S. Cook, J. M. Francioni, A. Hossain, M. Anwar, and M. F. Rahman, "Nonvisual tool for navigating hierarchical structures," *ACM SIGACCESS Access. Comput.*, no. 77–78, p. 133, Sep. 2003, doi: 10.1145/1029014.1028654.

[31] E. W. Huff, K. Boateng, M. Moster, P. Rodeghero, and J. Brinkley, "Examining the Work Experience of Programmers with Visual Impairments," in *Proceedings - 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME 2020*, Sep. 2020, pp. 707–711, doi: 10.1109/ICSME46990.2020.00077.

[32] H. Alotaibi, H. S. Al-Khalifa, and D. AlSaeed, "Teaching Programming to Students with Vision Impairment: Impact of Tactile Teaching Strategies on Student's Achievements and Perceptions," *Sustainability*, vol. 12, no. 13, p. 5320, Jul. 2020, doi: 10.3390/su12135320.

[33] V. Potluri, P. Vaithilingam, S. Iyengar, Y. Vidya, M. Swaminathan, and G. Srinivasa, "CodeTalk: Improving Programming Environment Accessibility for Visually Impaired Developers," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–11.

[34] E. Utreras and E. Pontelli, "Accessibility of block-based introductory programming languages and a tangible programming tool prototype," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Sep. 2020, vol. 12376 LNCS, pp. 27–34, doi: 10.1007/978-3-030-58796-3_4.

[35] A. Mountapmbeme and S. Ludi, "Investigating Challenges Faced by Learners with Visual Impairments using Block-Based Programming / Hybrid Environments," in *The 22nd International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '20)*, 2020.

[36] S. Ludi and M. Spencer, "Design Considerations to Increase Block-based Language Accessibility for Blind Programmers Via Blockly," *J. Vis. Lang. Sentient Syst.*, vol. 3, no. 1, pp. 119–124, 2017.

[37] A. Armaly, P. Rodeghero, and C. McMillan, "A Comparison of Program Comprehension Strategies by Blind and Sighted Programmers," *IEEE Trans. Softw. Eng.*, vol. 44, no. 8, pp. 712–724, Aug. 2018, doi: 10.1109/TSE.2017.2729548.

[38] A. Stefik, A. Haywood, S. Mansoor, B. Dunda, and D. Garcia, “Sodbeans,” in *2009 IEEE 17th International Conference on Program Comprehension*, 2009, pp. 293–294.

[39] E. Schanzer, S. Bahram, and S. Krishnamurthi, “Accessible AST-Based Programming for Visually-Impaired Programmers,” in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 2019, pp. 773–779.

[40] A. C. Smith, J. M. Francioni, and S. D. Matzek, “A Java Programming Tool for Students with Visual Disabilities,” 2000.

[41] A. C. Smith, J. M. Francioni, J. S. Cook, and M. Rahman, “Nonvisual Tool for Navigating Hierarchical Structures,” 2004.

[42] A. Armaly, P. Rodeghero, and C. McMillan, “Audiohighlight: Code skimming for blind programmers,” in *Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018*, Nov. 2018, pp. 206–216, doi: 10.1109/ICSME.2018.00030.

[43] O. Falase, A. F. Siu, and S. Follmer, “Tactile Code Skimmer: A Tool to Help Blind Programmers Feel the Structure of Code Figure 1. Indented code (left), its representation on the Tactile Code Skimmer (middle), and a participant interacting with TCS (right),” doi: 10.1145/3308561.3354616.

[44] J. Hutchinson and O. Metatla, “An initial investigation into non-visual code structure overviews through speech, non-speech and spearcons,” *Conf. Hum. Factors Comput. Syst. - Proc.*, vol. 2018-April, Apr. 2018, doi: 10.1145/3170427.3188696.

[45] A. Stefik, R. Alexander, R. Patterson, and J. Brown, “WAD : A Feasibility study using the Wicked Audio Debugger,” 2007.

[46] A. Stefik, A. Haywood, S. Mansoor, B. Dunda, and D. Garcia, “SODBeans,” *IEEE Int. Conf. Progr. Compr.*, pp. 293–294, 2009, doi: 10.1109/ICPC.2009.5090064.

[47] A. Stefik, C. Hundhausen, and R. Patterson, “An empirical investigation into the design of auditory cues to enhance computer program comprehension,” *Int. J. Hum. Comput. Stud.*, vol. 69, no. 12, pp. 820–838, Dec. 2011, doi: 10.1016/j.ijhcs.2011.07.002.

[48] A. Stefik, “ON THE DESIGN OF PROGRAM EXECUTION ENVIRONMENTS FOR NON-SIG HTED COMPUTER PROGRAMMERS,” *undefined*, 2008.

[49] L. B. Caraco, S. Deibel, Y. Ma, and L. R. Milne, “Making the Blockly Library Accessible via Touchscreen,” in *The 21st International ACM SIGACCESS Conference on Computers and Accessibility*, 2019, pp. 648–650.

[50] S. Ludi, G. Adams IV, B. Blankenship, and M. Dapiran, “The architectural challenges of adding accessibility features to ALICE as a case study of maintenance in educational software,” in *Proceedings - International Conference on Software Engineering*, 2011, pp. 33–35, doi: 10.1145/1984674.1984686.

[51] J. S. Y. Ong, N. A. O. Amoah, A. E. Garrett-Engele, M. I. Page, K. R. McCarthy, and L. R. Milne, "Expanding Blocks4All with Variables and Functions," in *The 21st International ACM SIGACCESS Conference on Computers and Accessibility*, 2019, pp. 645–647.

[52] C. Lewis, "Work in Progress Report: Nonvisual Visual Programming," 2014. Accessed: Dec. 05, 2020. [Online]. Available: www.ppig.org.

[53] "Accessible Rich Internet Applications (WAI-ARIA) 1.2." https://www.w3.org/TR/wai-aria-1.2/ (accessed Sep. 09, 2020).

[54] A. Stefik and E. Gellenbeck, "Using spoken text to aid debugging: An empirical study," *IEEE Int. Conf. Progr. Compr.*, pp. 110–119, 2009, doi: 10.1109/ICPC.2009.5090034.

[55] S. Ludi, J. Simpson, and W. Merchant, "Exploration of the use of auditory cues in code comprehension and navigation for individuals with visual impairments in a visual programming environment," in *ASSETS 2016 - Proceedings of the 18th International ACM SIGACCESS Conference on Computers and Accessibility*, Oct. 2016, pp. 279–280, doi: 10.1145/2982142.2982206.

[56] R. M. Siegfried *et al.*, "Teaching the Blind to Program Visually," vol. 3, no. 23, 2005.

[57] R. M. Siegfried, D. Diakoniarakis, K. G. Franqueiro, and A. Jain, "Extending a scripting language for visual basic forms," *ACM SIGPLAN Not.*, vol. 40, no. 11, pp. 37–40, 2005, doi: 10.1145/1107541.1107547.

[58] K. G. Franqueiro and R. M. Siegfried, "Designing a scripting language to help the blind program visually," *Eighth Int. ACM SIGACCESS Conf. Comput. Access. ASSETS 2006*, vol. 2006, pp. 241–242, 2006, doi: 10.1145/1168987.1169035.

[59] R. M. Siegfried, "Visual Programming and the Blind: The Challenge and the Opportunity," 2006. Accessed: May 10, 2021. [Online]. Available: http://www.adelphi.edu/~siegfrir/molly.

[60] R. M. Siegfried, "A scripting language to help the blind to program visually," *ACM SIGPLAN Not.*, vol. 37, no. 2, pp. 53–56, 2002, doi: 10.1145/568600.568611.

[61] M. Konecki, "A new approach towards visual programming for the blinds," *MIPRO 2012 - 35th Int. Conv. Inf. Commun. Technol. Electron. Microelectron. - Proc.*, pp. 935–940, 2012.

[62] "The Quorum Programming Language." .

[63] V. Koushik and C. Lewis, "An accessible blocks language: work in progress," in *Proceedings of the 18th International ACM SIGACCESS Conference on Computers and Accessibility*, 2016, pp. 317–318.

[64] J. Sánchez and F. Aguayo, "APL: Audio programming language for blind learners," in *Lecture Notes in Computer Science (including subseries Lecture Notes in*

Artificial Intelligence and Lecture Notes in Bioinformatics), 2006, vol. 4061 LNCS, pp. 1334–1341, doi: 10.1007/11788713_192.

[65] J. Sánchez and F. Aguayo, "Blind learners programming through audio," in *Conference on Human Factors in Computing Systems - Proceedings*, 2005, pp. 1769–1772, doi: 10.1145/1056808.1057018.

[66] R. M. Siegfried, D. Diakoniarakis, and K. G. Franqueiro, "Making Visual Programming Accessible to the Blind," *undefined*, 2005.

[67] "Blockly | Google Developers." .

[68] "The Quorum Programming Language (Abstract Only) | Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education." .

[69] S. K. Kane, V. Koushik, and A. Muehlbradt, "Bonk : Accessible Programming for Accessible Audio Games Learning Environments for Programming," pp. 132–142, 2018.

[70] V. Koushik and S. K. Kane, "Tangibles + programming + audio stories = fun," in *ASSETS 2017 - Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility*, Oct. 2017, pp. 341–342, doi: 10.1145/3132525.3134769.

[71] V. Koushik, D. Guinness, and S. K. Kane, "StoryBlocks: A Tangible Programming Game To Create Accessible Audio Stories," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019, pp. 1–12.

[72] G. Barbareschi and E. Costanza, "TIP-Toy: a tactile, open-source computational toolkit to support learning across visual abilities," *22nd Int. ACM SIGACCESS Conf. Comput. Access.*, 2020, doi: 10.1145/3373625.

[73] A. Sabuncuoglu, "Tangible music programming blocks for visually impaired children," *TEI 2020 - Proc. 14th Int. Conf. Tangible, Embed. Embodied Interact.*, pp. 423–429, 2020, doi: 10.1145/3374920.3374939.

[74] N. Villar, D. Cletheroe, A. Thieme, C. Morrison, T. Regan, and G. Saul, "Physical programming for blind and low vision children at scale," in *Conference on Human Factors in Computing Systems - Proceedings*, May 2019, doi: 10.1145/3290607.3313241.

[75] A. Thieme, C. Morrison, N. Villar, M. Grayson, and S. Lindley, "Enabling collaboration in learning computer programing inclusive of children with vision impairments," *DIS 2017 - Proc. 2017 ACM Conf. Des. Interact. Syst.*, pp. 739–752, 2017, doi: 10.1145/3064663.3064689.

[76] C. Morrison *et al.*, "Torino: A tangible programming language inclusive of children with visual disabilities," *Human--Computer Interact.*, pp. 1–49, 2018.

[77] C. Morrison *et al.*, “Physical Programming for Blind and Low Vision Children at Scale,” *Human-Computer Interact.*, pp. 1–35, Jul. 2019, doi: 10.1080/07370024.2019.1621175.

[78] Z. Rong, N. F. Chan, T. Chen, and K. Zhu, “CodeRhythm: Designing inclusive tangible programming blocks,” *DIS 2020 Companion - Companion Publ. 2020 ACM Des. Interact. Syst. Conf.*, pp. 105–110, 2020, doi: 10.1145/3393914.3395895.

[79] Z. Rong, N. F. Chan, T. Chen, and K. Zhu, *Toward inclusive learning: Designing and evaluating tangible programming blocks for visually impaired students*, vol. 12183 LNCS. Springer International Publishing, 2020.

[80] A. Shetty, E. Jarjue, and H. Peng, “Tangible Web Layout Design for Blind and Visually Impaired People: An Initial Investigation,” *UIST 2020 - Adjun. Publ. 33rd Annu. ACM Symp. User Interface Softw. Technol.*, pp. 37–39, 2020, doi: 10.1145/3379350.3416178.

[81] Z. Wang and A. Wagner, “Evaluating a tactile approach to programming scratch,” *ACMSE 2019 - Proc. 2019 ACM Southeast Conf.*, pp. 226–229, 2019, doi: 10.1145/3299815.3314464.

[82] M. Paulk and A. Wagner, “CodeBox64 | Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education.” .

[83] S. Kakehashi, T. Motoyoshi, K. Koyanagi, T. Ohshima, and H. Kawakami, “P-CUBE: Block type programming tool for visual impairments,” in *Proceedings - 2013 Conference on Technologies and Applications of Artificial Intelligence, TAAI 2013*, 2013, pp. 294–299, doi: 10.1109/TAAI.2013.65.

[84] S. Ludi, M. Abadi, Y. Fujiki, S. Herzberg, and P. Sankaran, “JBrick: Accessible lego mindstorm programming tool for users who are visually impaired,” *ASSETS’10 - Proc. 12th Int. ACM SIGACCESS Conf. Comput. Access.*, pp. 271–272, 2010, doi: 10.1145/1878803.1878866.

[85] A. M. Howard, C. H. Park, and S. Remy, “Using Haptic and Auditory Interaction Tools to Engage Students with Visual Impairments in Robot Programming Activities,” *IEEE Trans. Learn. Technol.*, vol. 5, no. 1, pp. 87–95, 2012, doi: 10.1109/TLT.2011.28.

[86] S. L. Remy, “Extending access to personalized verbal feedback about robots for programming students with visual impairments,” in *Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility, ASSETS 2013*, 2013, doi: 10.1145/2513383.2513384.

[87] J. Ahn, W. Sung, S. Lee, J. H.-S. for Information, and U. 2017, “COBRIX: A Physical Computing Interface for Blind and Visually Impaired Students to Learn Programming,” *Learntechlib.Org*, 2016.

[88] P. Molins-Ruano, C. Gonzalez-Sacristan, and C. Garcia-Saura, “Phogo: A low cost, free and ‘maker’ revisit to Logo,” *Comput. Human Behav.*, vol. 80, no. September, pp. 428–440, 2018, doi: 10.1016/j.chb.2017.09.029.

[89] G. H. M. Marques *et al.*, “Donnie robot: Towards an accessible and educational robot for visually impaired people,” *Proc. - 2017 LARS 14th Lat. Am. Robot. Symp. 2017 5th SBR Brazilian Symp. Robot. LARS-SBR 2017 - Part Robot. Conf. 2017*, vol. 2017-Decem, pp. 1–6, 2017, doi: 10.1109/SBR-LARS-R.2017.8215273.

[90] L. Abreu, A. C. Pires, and T. Guerreiro, “TACTOPI: A Playful Approach to Promote Computational Thinking for Visually Impaired Children,” *ASSETS 2020 - 22nd Int. ACM SIGACCESS Conf. Comput. Access.*, 2020, doi: 10.1145/3373625.3418003.

[91] A. C. Pires, F. Rocha, A. J. De Barros Neto, H. Simão, H. Nicolau, and T. Guerreiro, “Exploring accessible programming with educators and visually impaired children,” in *Proceedings of the Interaction Design and Children Conference, IDC 2020*, Jun. 2020, pp. 148–160, doi: 10.1145/3392063.3394437.

[92] “Code Jumper.” https://codejumper.com/ (accessed Apr. 12, 2021).

[93] T. Motoyoshi, N. Tetsumura, H. Masuta, K. Koyanagi, T. Oshima, and H. Kawakami, “Tangible programming gimmick using RFID systems considering the use of visually impairments,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016, vol. 9758, pp. 51–58, doi: 10.1007/978-3-319-41264-1_7.

[94] S. Kakehashi, T. Motoyoshi, K. Koyanagi, T. Oshima, H. Masuta, and H. Kawakami, “Improvement of P-CUBE: Algorithm education tool for visually impaired persons,” *IEEE SSCI 2014 2014 IEEE Symp. Ser. Comput. Intell. - RiiSS 2014 2014 IEEE Symp. Robot. Intell. Informationally Struct. Space, Proc.*, 2015, doi: 10.1109/RIISS.2014.7009180.

[95] M. Tsuda *et al.*, “Improvement of a tangible programming tool for the study of the subroutine concept,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Jul. 2018, vol. 10896 LNCS, pp. 611–618, doi: 10.1007/978-3-319-94277-3_95.

[96] R. Dorsey, C. H. Park, and A. M. Howard, “Developing the Capabilities of Blind and Visually Impaired Youth to Build and Program Robots,” 2013, Accessed: Oct. 31, 2021. [Online]. Available: https://smartech.gatech.edu/handle/1853/48209.

[97] S. A. Ludi and T. Reichlmayr, “Developing inclusive outreach activities for students with visual impairments,” *ACM SIGCSE Bull.*, vol. 40, no. 1, pp. 439–443, Feb. 2008, doi: 10.1145/1352322.1352285.

[98] S. Ludi and T. Reichlmayr, “The use of robotics to promote computing to pre-college students with visual impairments,” *ACM Trans. Comput. Educ.*, vol. 11, no. 3, pp. 1–20, 2011.

[99] S. Ludi, L. Ellis, and S. Jordan, “An accessible robotics programming environment for visually impaired users,” *ASSETS14 - Proc. 16th Int. ACM SIGACCESS Conf. Comput. Access.*, pp. 237–238, 2014, doi: 10.1145/2661334.2661385.

[100] “Micro:bit Educational Foundation | micro:bit.” https://microbit.org/ (accessed Oct. 31, 2021).

[101] C. M. Baker, C. L. Bennett, and R. E. Ladner, “Educational Experiences of Blind Programmers,” in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 2019, pp. 759–765.

[102] J. P. Bigham, M. B. Aller, J. T. Brudvik, J. O. Leung, L. A. Yazzolino, and R. E. Ladner, “Inspiring blind high school students to pursue computer science with instant messaging chatbots,” in *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, 2008, pp. 449–453.

[103] “Learn | Code.org.” https://code.org/learn (accessed Apr. 27, 2020).

[104] “Swift Playgrounds - Apple.” https://www.apple.com/swift/playgrounds/ (accessed Jul. 01, 2020).

[105] “The VoiceOver Rotor | American Foundation for the Blind.” https://www.afb.org/blindness-and-low-vision/using-technology/cell-phones-tablets-mobile/apple-ios-iphone-and-ipad-2 (accessed Apr. 27, 2020).

[106] A. Armaly and C. McMillan, “An empirical study of blindness and program comprehension,” in *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016, pp. 683–685.

[107] S. Playgrounds, “Swift Playgrounds - Apple,” 2019, Accessed: Apr. 12, 2021. [Online]. Available: https://www.apple.com/swift/playgrounds/.

[108] E. W. Huff, K. Boateng, M. Moster, P. Rodeghero, and J. Brinkley, “Exploring the Perspectives of Teachers of the Visually Impaired Regarding Accessible K12 Computing Education,” in *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, Mar. 2021, vol. 7, pp. 156–162, doi: 10.1145/3408877.3432418.

[109] “| AccessComputing.” https://www.washington.edu/accesscomputing/accesscsforall (accessed Apr. 12, 2021).

[110] A. Stefk, W. Allee, R. E. Ladner, and S. Mealin, “Computer science principles for teachers of blind and visually impaired students,” in *SIGCSE 2019 - Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, Feb. 2019, vol. 19, pp. 766–772, doi: 10.1145/3287324.3287453.

[111] J. Sabourin, L. Kosturko, and S. McQuiggan, “CodeSnaps,” in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, Feb. 2016, pp. 242–242, doi: 10.1145/2839509.2850508.

[112] “About the VoiceOver rotor on iPhone, iPad, and iPod touch - Apple Support.” https://support.apple.com/en-us/HT204783 (accessed Apr. 11, 2021).

[113] S. Ludi, D. Bernstein, and K. Mutch-Jones, “Enhanced robotics! Improving building and programming learning experiences for students with visual impairments,” in *SIGCSE 2018 - Proceedings of the 49th ACM Technical Symposium*

on Computer Science Education, Feb. 2018, vol. 2018-January, pp. 372–377, doi: 10.1145/3159450.3159501.

[114] S. Mutch-Jones, K., Bernstein, D., Ludi, "Creating access to computer science: Enhancing engagement and learning for students with visual impairments," *Vis. Impair. Deaf. Educ. Q.*, vol. 64, no. 4, pp. 38–51, 2016.

[115] "Minecraft Official Site | Minecraft." https://www.minecraft.net/en-us (accessed Jun. 23, 2022).

[116] N. Fraser, "Ten things we've learned from Blockly," *Proc. - 2015 IEEE Blocks Beyond Work. Blocks Beyond 2015*, pp. 49–50, Dec. 2015, doi: 10.1109/BLOCKS.2015.7369000.

[117] N. C. C. Brown, J. Mönig, A. Bau, and D. Weintrop, "Panel: Future directions of block-based programming," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, 2016, pp. 315–316.

[118] A. Mountapmbeme and S. Ludi, "How Teachers of the Visually Impaired Compensate with the Absence of Accessible Block-Based Languages; How Teachers of the Visually Impaired Compensate with the Absence of Accessible Block-Based Languages," *23rd Int. ACM SIGACCESS Conf. Comput. Access.*, doi: 10.1145/3441852.

[119] M. Das, D. Marghitu, M. Mandala, and A. Howard, "Accessible block-based programming for k-12 students who are blind or low vision," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 12769 LNCS, pp. 52–61, 2021, doi: 10.1007/978-3-030-78095-1_5/FIGURES/7.

[120] "Computer Science For All | whitehouse.gov." https://obamawhitehouse.archives.gov/blog/2016/01/30/computer-science-all (accessed Apr. 13, 2022).

[121] "WAI-ARIA Overview | Web Accessibility Initiative (WAI) | W3C." https://www.w3.org/WAI/standards-guidelines/aria/ (accessed Apr. 09, 2022).

[122] A. Stefik, C. Hundhausen, and R. Patterson, "An empirical investigation into the design of auditory cues to enhance computer program comprehension," *Int. J. Hum. Comput. Stud.*, vol. 69, no. 12, pp. 820–838, 2011.

[123] "RITAccess/blockly: The web-based visual programming editor." https://github.com/RITAccess/blockly (accessed Jul. 06, 2022).

[124] "@blockly/keyboard-navigation Demo." https://google.github.io/blockly-samples/plugins/keyboard-navigation/test/ (accessed Apr. 09, 2022).

[125] "Accessible block-based programs study." https://mountapmbeme.com/study/home.html (accessed Jul. 07, 2022).

[126] S. A. Brewster, "Using nonspeech sounds to provide navigation cues," *ACM Trans. Comput. Interact.*, vol. 5, no. 3, pp. 224–259, Sep. 1998, doi: 10.1145/292834.292839.