© Copyright 2017

Catherine M. Baker

Understanding and Improving Blind Students' Access to Visual Information in Computer Science Education

Catherine M. Baker

A dissertation

submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

University of Washington

2017

Reading Committee:

Richard Ladner, Chair

Alan Borning

Andrew Ko

Program Authorized to Offer Degree:

Computer Science & Engineering

University of Washington

Abstract

Understanding and Improving Blind Students' Access to Visual Information in Computer Science Education

Catherine M. Baker

Chair of the Supervisory Committee:
Professor Emeritus Richard Ladner
Computer Science & Engineering

Teaching people with disabilities tech skills empowers them to create solutions to problems they encounter and prepares them for careers. However, computer science is typically taught in a highly visual manner which can present barriers for people who are blind. The goal of this dissertation is to understand and decrease those barriers.

The first projects I present looked at the barriers that blind students face. I first present the results of my survey and interviews with blind students with degrees in computer science or related fields. This work highlighted the many barriers that these blind students faced. I then followed-up on one of the barriers mentioned, access to technology, by doing a preliminary accessibility evaluation of six popular integrated development environments (IDEs) and code editors. I found that half were unusable and all had some inaccessible portions.

As access to visual information is a barrier in computer science education, I present three projects I have done to decrease this barrier. The first project is Tactile Graphics with a Voice (TGV). This project investigated an alternative to Braille labels for those who do not know Braille and showed that TGV was a potential alternative. The next project was StructJumper, which created a modified abstract syntax tree that blind programmers could use to navigate through code with their screen reader. The evaluation showed that users could navigate more quickly and easily determine the relationships of lines of code when they were using StructJumper compared to when they were not. Finally, I present a tool for dynamic graphs (the type with nodes and edges) which had two different modes for handling focus changes when moving between graphs. I found that the modes support different approaches for exploring the graphs and therefore preferences are mixed based on the user's preferred approach. However, both modes had similar accuracy in completing the tasks.

These projects are a first step towards the goal of making computer science education more accessible to blind students. By identifying the barriers that exist and creating solutions to overcome them, we can support increasing the number of blind students in computer science.

TABLE OF CONTENTS

List of Figures viii

List of Tables xi

Chapter 1. Introduction 1

1.1 Motivation 1

1.2 Thesis Statement 3

1.3 Research Questions and Approaches 4

1.3.1 Barriers in Computer Science 5

1.3.2 Tactile Graphics with a Voice 5

1.3.3 StructJumper 6

1.3.4 Dynamic Graphs 6

1.4 Contributions 8

1.5 Dissertation Overview 9

Chapter 2. Related Work 11

2.1 Practices and Challenges for Blind Programmers 11

2.2 Computer Science Education for Blind Programmers 12

2.3 Programming Tools for Blind Programmers 14

2.4 Making Graphics Accessible to Blind Students 16

2.4.1 Tactile Graphics 16

2.4.2 Computer & Tablet Based Systems 17

Chapter 3. Educational Experiences of Blind Programmers 21

3.1 Introduction 21
3.2 Method 23
3.2.1 Survey and Interviews 23
3.2.2 Participants 23
3.2.3 Analysis 24
3.3 Findings 24
3.3.1 Technology 24
3.3.2 University Learning 27
3.3.3 Informal Learning 37
3.4 Discussion 39
3.5 Summary 41
Chapter 4. Evaluation of IDEs 42
4.1 Methods 42
4.2 Results 43
4.2.1 IntelliJ 44
4.2.2 Visual Studio 44
4.2.3 Eclipse 45
4.2.4 NetBeans 46
4.2.5 Notepad++ 47
4.2.6 Sublime Text 47
4.3 Summary 47
Chapter 5. Tactile Graphics with a Voice 49

5.1 Introduction 50
5.2 Related Work 52
5.2.1 Accessing Textual Information on Tactile Graphics 52
5.2.2 Camera Use By Blind People 53
5.2.3 Finger Pointing 54
5.3 Formative Studies 56
5.4 Tactile Graphics with a Voice (TGV) 57
5.4.1 Tactile Graphics with QR Codes 57
5.4.2 Smartphone Application 58
5.4.3 Feedback Modalities 60
5.5 Integrating with the Tactile Graphics Assistant 61
5.5.1 QR Code Placement Algorithm 61
5.5.2 Feasibility Evaluation 64
5.6 Longitudinal Study 65
5.6.1 Participants 66
5.6.2 Apparatus 66
5.6.3 Procedure 67
5.6.4 Design and Analysis 69
5.7 Results 70
5.7.1 Accuracy 70
5.7.2 Time 71
5.7.3 Feedback Modality 72
5.8 Comparison to Braille 75

5.8.1 Difficulties in Creating Tactile Graphics with Braille Labels 75

5.8.2 Size of Braille vs QR Codes 76

5.8.3 Study 76

5.9 Discussion 78

5.10 Summary 78

Chapter 6. StructJumper 80

6.1 Introduction 81

6.2 Related Work 83

6.3 StructJumper Design and Implementation 85

6.4 Experiment Design 89

6.4.1 Participants 89

6.4.2 Set-Up 90

6.4.3 Procedure 90

6.4.4 Design and Analysis 94

6.5 Results 94

6.5.1 Task Completion Time 95

6.5.2 Task Score 96

6.5.3 Participant Experience 98

6.6 Qualitative Results 99

6.6.1 Quicker and Easier 99

6.6.2 Better Understanding of the Layout 100

6.6.3 Lack of Cues 101

6.6.4 Change in Focus 102

6.6.5 Unfamiliar Code 103

6.6.6 Programming Use 103

6.7 Discussion 104

6.8 Summary 106

Chapter 7. Dynamic Graphs 107

7.1 Introduction 108

7.2 Related Work 110

7.3 System Design 111

7.3.1 Overview of the Prototype 111

7.3.2 Modes 114

7.3.3 Changes in Response to Pilot 116

7.4 Experiment Design 117

7.4.1 Participants 117

7.4.2 Set-up 117

7.4.3 Procedure 118

7.4.4 Design and Analysis 121

7.5 Results 121

7.5.1 Data Cleaning 121

7.5.2 Time 123

7.5.3 Accuracy 124

7.5.4 Actions 126

7.5.5 Participant Experience 127

7.6 Qualitative Results 128

7.6.1 Preferences 129
7.6.2 Lost Context 129
7.6.3 Saved State 130
7.6.4 Memory 130
7.7 Discussion 132
7.8 Summary 134

Chapter 8. Contributions and Future Work 135

8.1 Summary of Chapters 135
8.2 Summary of Contributions 137
8.2.1 Artifact Contributions 137
8.2.2 Empirical Contributions 140
8.3 Support of Thesis Statement 142
8.4 Limitations 144
8.5 Future Work 146
8.5.1 Creation of Accessible Materials 146
8.5.2 Creating Visuals 148
8.6 Concluding Remarks 150

Bibliography 151

Appendix A 158

Appendix B 160

Appendix C 161

Appendix D 163

Appendix E 165

Appendix F.. 167

Appendix G 168

Appendix H 169

LIST OF FIGURES

Figure 3.1. Survey respondents were asked what proportion of their classes had inaccessible portions. This chart shows how many survey respondents selected each proportion27

Figure 3.2. This chart shows the resources that the survey respondents have used to learn about topics in computer science. The solid bar is the resources they have used and the striped bar is the resources that they have used (or tried to use) and found accessibility problems with. 37

Figure 5.1. The Tactile Graphics with a Voice system in use. There is an embossed bar chart with QR codes for labels and the subject is using the finger pointing mode to select which QR code to scan. 50

Figure 5.2. A sample low fidelity tactile graphic showing a bar chart made with pipe cleaners and sand paper. 51

Figure 5.3. Above is an example of each category that the images can be placed into. The categories are: identified correct QR code (left), correct QR code was not found (middle), identified incorrect QR code when the correct was possible (right)......................... 59

Figure 5.4. This is an example of each of the tasks that the participants completed. In each session, participants used similar graphics, but with different labels (i.e. the parabola might be the opposite direction and have a different vertex). 67

Figure 5.5. A comparison of the average accuracy for the Bar Chart task across the six sessions for the three modes (n=10) and Braille (n=6) on the last session. Participants were asked to find the range of the tallest bar and their answer could be 0, 50 or 100% correct.... 70

Figure 5.6. A comparison of the average time for each participant to give the answer for a task for the three modes (n=10) across the six sessions as well as for Braille (n=6) on the final session. 71

Figure 5.7. A comparison of the same image which is similar to one from a pre-calculus textbook [35] in its original form, tactile graphic form with the labels in Braille and tactile graphic form with labels as QR codes. The bottom text is a good example where the QR codes can be smaller than the equivalent Braille. 75

Figure 6.1. Screenshot of StructJumper with source code file on top and tree of nesting structure on bottom. 81

Figure 6.2. Code for a simple calculator class, which is turned into the tree in Figure 6.3Figure 6.3.. 85

Figure 6.3. The tree created from the code in Figure 6.2. Code sections have no further nesting. Note in this image, the first code section corresponds to the code containing the member variable declared at the beginning of the Calculator class....................................... 86

Figure 6.4. This is an example of a portion of what the Package Explorer in Eclipse would show. 88

Figure 6.5. This chart shows the average completion time that it took participants to complete the three tasks broken down by type. The bars represent the standard error................. 95

Figure 6.6 This chart shows the average score for all participants on the three tasks broken down by task. The bars represent the standard error. 96

Figure 6.7. This chart show the average score for the participants for the semantically anchored questions. A higher value is better for all three questions. The bars represent the standard error.. 98

Figure 7.1. A sample image that could occur in a slide or textbook when teaching the insert operation of min-heap. This image shows the before and after of the percolate up operation after node 2 has been inserted into the min-heap. This image illustrates the use of dynamic graphs to teach CS concepts. 108

Figure 7.2. This shows the differences in how focus (highlighted in red and circled) moves in the two modes. The graph in a) is the present graph and the graphs in b) and c) show where the focus would be in the future for the two modes. In b), we are assuming that this is the first visit to the graph, so it goes to the initial placement in the graph. In c), it goes to the same relative location.. 114

Figure 7.3. These graphs show the time taken in seconds to complete the tasks with each of the modes by participant. Each bar represents a single task. The three graphs are for each type of task with the solid bar representing the time with the previous location mode and the striped bar with the relative location mode. The missing bars are due to the data points that needed to be removed as described in 7.5.1... 123

Figure 7.4. This shows the average accuracy for each completed task. Participants could get a max of 1. The error bars represent the standard error. 124

Figure 7.5. These graphs show the number of actions taken by each participant to complete the tasks. The graph switches are represented by a darker blue. Each bar represents a single task. The three graphs are for each type of task with the solid bar representing the previous location mode and the striped bar the relative location mode. The missing bars/participants are due to the data points that needed to be removed or were missing as described in 7.5.1. 126

Figure 7.6. This figure shows the percentage of participants with each Likert response. Agreements is shown with green vertical lines, disagreement with orange horizontal lines and solid gray is the middle value. The less white visible, the more extreme the agreement/disagreement. The values of the Likert question relating to memory and frustration are swapped so that the higher numbers are better for all four questions.128

LIST OF TABLES

Table 1.1 This table summarizes the questions I seek to answer in my dissertation and the approaches I took to do so. 7

Table 5.1. Overall average accuracy of participants on the tasks with each mode........... 71

Table 6.1 A table of the keyboard shortcuts that can be used to navigate in the tree and the code editor 87

Table 7.1. The list of keyboard shortcuts for the prototype and their association actions and what is spoken when the action is taken. I assumed that the starting point is the graph in Figure 7.2a, and that the user is in the previous location mode and the user reached Brad via the out edges of Tom.. 112

Table 7.2. Summary of the Likert scores... 127

ACKNOWLEDGEMENTS

I would like to thank my committee and collaborators for their support and feedback on the work presented in this dissertation.

The Educations Experiences of Blind Programmers work was done in collaboration with Cynthia Bennett and Richard Ladner. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. (DGE-1256082).

The Accessibility Evaluation of IDEs and Code Editors work is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1256082.

The Tactile Graphics with a Voice work was done in collaboration with Lauren Milne, Jeffrey Scofield, Cynthia Bennett, and Richard Ladner. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-1256082 and National Science Foundation Grant No. IIS-1116051. This work was supported by the U.S. Department of Education, Office of Special Education Programs (Cooperative Agreement #H327B100001).

The StructJumper work was done in collaboration with Lauren Milne and Richard Ladner. This material is based upon work supported by the NSF Graduate Research Fellowship under Grant No. DGE-1256082 and NSF grant IIS-1116051.

The Dynamic Graph Tool work was done in collaboration with Richard Ladner.

Any opinions, findings, and conclusions or recommendations expressed in this dissertation are those of the author and do not necessarily reflect the views of the National Science Foundation or the Department of Education.

DEDICATION

To my family for all the support they have given me over the years.

Chapter 1. INTRODUCTION

Diversity can bring many benefits, however computer science is not a very diverse field [60]. In reaction, several diversity efforts have sprung up to raise participation. While these efforts are necessary, most have focused on increasing the participation of women and minorities. Few have considered disability as a part of diversity. Notably, AccessComputing[1] attempts to increase the participation of people with disabilities in computer science by connecting students to successful professionals and funding conference travel and internship experiences. While these actions are beneficial, they do not address the accessibility barriers that exist in the field of computer science. These barriers are important to address to provide equal access to students with disabilities. Providing equal access is important both from a legal standpoint, as universities in the United States are required to provide accommodations to students with disabilities, and from an equity standpoint, as we want everyone to be able to study computer science. In this dissertation, I will present research I have done to 1) discover the barriers that prevent people who are blind from entering the field of computer science and 2) design, develop, and evaluate tools and interactions that provide novel ways to access visual information that is necessary in computer science.

1.1 MOTIVATION

In 2006, the United Nations contracted a preliminary accessibility audit to understand the state of web accessibility. The results were not good; only 3 of the 100 websites investigated met the single A accessibility standard [62]. This standard was determined using the WCAG 1.0 accessibility guidelines[2], which has 14 guidelines with check points of different levels of priority. To achieve

[1] https://www.washington.edu/accesscomputing/
[2] https://www.w3.org/TR/WAI-WEBCONTENT/#Guidelines

single A status, a website must meet all top priority check points for each guideline, and this only ensures the most basic accessibility. And the accessibility barriers are not just limited to the web. When Branham and Kane looked at the accessibility challenges the blind individuals faced in the workplace, computer software was the most common issue raised, with specialized software having the most issues [17].

Though accessibility problems are prevalent, we have the opportunity to decrease the barriers in the future by educating the next generation of designers and developers of software. In a recent survey by Stack Overflow, 3.4% of the respondents identified as having a disability (1% identified as themselves as blind) [76]. My goal is to increase the number of developers in the field who are blind, as we will see benefits in two ways from this increase. The first is that developers with disabilities will be able to create their own solutions to challenges that are unique to their disability. The second is that they will have an impact on the teams they work on in industry by increasing awareness of the needs of people with disabilities, which can increase the creation of inclusive technology.

When it comes to designing and creating technology to overcome barriers that are specific to a person's disability, we should empower them to create their own solutions [50]. When people with disabilities are not involved with the creation of solutions, it can come off as paternalistic [50]. Instead of being dependent on people who are less familiar with their disability to create solutions for them, when we empower people with disabilities to create their own solutions, there is an expert creating the tools, improving the likelihood that they are actually addressing the problems faced and will be adopted.

When disabled developers are a part of the workforce, the impact that they have is not just limited to the solutions they are able to create, but also what they teach their other team members.

Having companies include accessibility in their products often requires two things: 1) someone to advocate for and prioritize accessibility and 2) an understanding of the experience and needs of someone with a disability. In Porter and Kientz's interviews with game developers, they found that the second condition was particularly important, as many young, able-bodied people did not understand how someone's disability may affect that person's experience and therefore could not create accessible products even if they wanted to [65]. When someone with a disability is on a team, the rest of the team is more likely to become aware of the implications of how a specific design or implementation affects users with disabilities. They can use this knowledge to create more accessible software.

But for people who are disabled to have the technical skills necessary to be a part of the field, they need to be able to learn the skills. This can present challenges for blind students. As it is currently taught, computer science tends to be a very visual subject and uses a lot of different technologies that may not be accessible. This can create barriers that cause a blind student to decide not to learn computer science in a college or university [27]. For this reason, I have focused on making computer science education more accessible for blind students.

1.2 Thesis Statement

The work in my dissertation is to support the following thesis claim:

Blind students face many barriers in their education, including access to technology and visual information, which can decrease their motivation to study computer science and other technical fields. To improve the access to visual information, we can use mainstream technologies to augment its accessibility, which can support: 1) Increased understanding of the structure and relationships of information and 2) Decreased cognitive load.

1.3 Research Questions and Approaches

To support my thesis, I present the results of my investigations into the following research questions:

- RQ1: What are the barriers that can prevent someone who is blind from studying computer science?
- RQ2: How can we provide access to graphics for people who are blind and do not know Braille?
- RQ3: How can we make it easier for blind programmers to contextualize their location and navigate through code?
- RQ4: How can we make it easier for someone who is blind to understand changes in graphs?

To answer RQ1, I conducted a survey and follow-up interviews with blind graduates of computer science or related fields and did a preliminary accessibility evaluation. The survey and interviews identified barriers that the students had to face, and the accessibility evaluation was a deeper dive into one of the barriers raised. To answer RQ2, I developed and evaluated Tactile Graphics with a Voice to determine if QR codes could be as viable alternative to Braille labels for those who do not know Braille. For RQ3, I designed and evaluated StructJumper, an Eclipse plug-in that harnessed the structure of the code to determine if using specific statements in the code similar to website headers improved a blind programmer's ability to navigate and contextualize statements. Finally, for RQ4, I designed and created a dynamic graph prototype to determine the effects of different methods of handling the focus change when moving between the different versions of the graph.

1.3.1 *Barriers in Computer Science*

The goal of this qualitative research was to understand the barriers that exist for blind students in computer science and related fields. To do this, I conducted this research in a couple of stages. First, I did a survey with 15 blind people who had graduated with degrees in computer science or related fields. The goal was to get a high-level view of the barriers that exist in computer science education. I then did follow-up interviews with 10 of the survey respondents. The interviews went into more details on the challenges they encountered, the solutions (or lack of solutions) that addressed those issues, and the effect that the issues had on their studies. I found that barriers permeated all spectrums of education, from access to materials and technology to interacting with faculty members. As a follow-up to these results, I looked closer at the accessibility of integrated development environments (IDEs) and code editors. I found that many of the popular IDEs and code editors were completely unusable.

1.3.2 *Tactile Graphics with a Voice*

One common way for blind students to access images from textbooks is through tactile graphics, raised versions of the images which can be felt tactilely. Braille is the typical way that labels are placed on tactile graphics and current alternatives require specialized devices. Therefore, I investigated the potential of using QR codes in place of Braille labels. This research was done in several phases. The first phase investigated the use of applications that require the camera for blind users and what challenges they face. As a result of that study, I designed and evaluated Tactile Graphics with a Voice, a system that included the graphics with QR codes placed as the labels and a smartphone app that had multiple modes of feedback for the users. I wanted to evaluate whether users could scan the QR codes, particularly when labels were close to together. To do so, I had ten

participants complete six sessions using TGV to answer questions about the diagrams with each of the modes. I found that while users had different preferences for the modes, they were able to successfully answer questions about graphics with the system.

1.3.3 *StructJumper*

As screen readers only have access to a single line of code, it can be difficult for a blind use to navigate through the code quickly. I designed, developed, and evaluated StructJumper to determine whether the structure of the code could be harnessed to allow a blind user to quickly navigate or look up information. StructJumper is an Eclipse plug-in that created a modified abstract syntax tree which only included some of the lines of code. The tree included the lines of code that precipitate an indentation change and all other lines were condensed into entries labeled Code Section. I evaluated StructJumper by having seven blind programmers complete tasks with and without StructJumper to understand the differences in accessibility between the two interfaces. I found that there was a trend that users were faster with StructJumper while having similar accuracy and users liked having this navigation structure as they could quickly navigate and understand the structure of the code.

1.3.4 *Dynamic Graphs*

There are many instances in computer science where dynamic graphs[3] are used, including textbooks and slides. Students also encounter dynamic graphs in code visualizers as many data structures are just a specific type of graph. As of now, code visualizers are not accessible to blind students as there is no way to automatically translate the visualization into a modality that a blind

[3] I am using the term dynamic graphs to mean graphs that are made up of nodes and edges and are changing over time, whether that is through an animation or multiple images shown side by side.

student can access. Yet they provide useful information for students. For that reason, I investigated the best ways to make these visualizations accessible by allowing students to navigate the graphs that these visualizations create. To do this, I designed, developed, and evaluated a dynamic graph prototype. This prototype had two modes for handling the focus change when switching between the versions of a graph. The previous location mode treated the navigation of the graphs as independent. The relative location mode used the location of the focus in the first graph to determine where to place focus in the second graph. To evaluate the modalities, I had seven people who are blind or have low vision and have experience with graphs answer questions about changes in graphs with both modes. I found that participants were able to answer the questions about the changes in the graphs and wanted to be able to access both modes.

Table 1.1 This table summarizes the questions I seek to answer in my dissertation and the approaches I took to do so.

	Question	Approach
RQ1	What are the barriers that can prevent someone who is blind from studying computer science?	Survey and interview with blind programmers (chapter 3); Preliminary accessibility evaluation of IDEs and code editors (chapter 4)
RQ2	How can we provide access to graphics for people who are blind and do not know Braille?	Design, development, and evaluation of Tactile Graphics with a Voice (chapter 5)
RQ3	How can we make it easier for blind programmers to contextualize their location and navigate through code?	Design, development, and evaluation of StructJumper (chapter 6)
RQ4	How can we make it easier for someone who is blind to understand changes in graphs?	Design, development, and evaluation of a dynamic graph prototype (chapter 7)

1.4 CONTRIBUTIONS

My dissertation resulted in both empirical and artifact contributions that support my thesis statement above. The contribution outputs and how they support my thesis are summarized below:

1. Knowledge of barriers that blind students may face when studying computer science. Through a survey with 15 blind graduates and follow-up interviews with 10 of the respondents, I found that there are many barriers for blind students learning computer science, from access to materials to opinions of the faculty. This work provides support for the first part of my thesis statement by showing examples of the barriers and how they have affected students' motivation to study computer science.
2. Design, development, and evaluation of Tactile Graphics with a Voice (TGV). I showed that TGV provides an alternative to Braille labels. Users could successfully scan QR code labels using the application and became faster over time. In this project, I used a smartphone to allow non-Braille readers the ability to access labels. As many graphics are unusable without the labels, TGV provided access to this information. The labels were available tactilely on the graphic, so a user could determine both what the label said and how it related to the graphic, providing an increased understanding of the relationship of information.
3. Design, development, and evaluation of StructJumper. I showed that by using the structure of the code, a navigation system could be created that allowed users to easily determine their context or navigate to a new location in the code. This work decreases the cognitive load faced by blind programmers by decreasing the amount of information that is held in memory and provides a way to blind programmers to easily determine the relationship of two lines of code (e.g. are they nested within the same conditionals).

4. Design, development, and evaluation of two interaction modalities for a dynamic graph tool. I showed that participants could answer questions about the changes in graphs and wanted to have access to multiple modes so that they could select a mode based on the task they were trying to complete. One mode of this tool explicitly provided a link between the past and present version of a graph at a specific location making it easy to determine changes that occurred at a location and therefore an increased understanding of the relationship of the two graphs. Additionally, as the modes differed in what information needed to be held in memory in order to understand the context of the focus when switching between graphs, I saw the participants preferred the mode they thought decreased their cognitive load.

1.5 Dissertation Overview

This dissertation is divided into 8 chapters, which are summarized below.

- In Chapter 2, Related Work, I summarize the work done previously that looks at blind programmers. This includes the work that investigates the experiences of blind programmers as well as the tools that have been created to help them. Additionally, as diagrams are heavily used in teaching computer science, I investigated making graphs accessible.
- In Chapter 3, Education Experiences of Blind Programmers, I describe the survey and interviews I did with blind programmers to understand their experiences while learning to program. I aimed to understand the barriers they faced and the work arounds they needed to use. I report on the findings of 15 survey responses and 10 follow-up interviews.

- In Chapter 4, Evaluation of IDEs, I describe a preliminary investigation of the accessibility of programming environments. I looked at six popular code editors or integrated development environments and report on the accessibility of their basic features.
- In Chapter 5, Tactile Graphics with a Voice, I describe the design and evaluation of Tactile Graphics with a Voice (TGV) system. This system provided an alternate way to access labels on tactile graphics and could be integrated with the Tactile Graphics Assistant [44]. I report on the results of the evaluation of TGV, which showed mixed preferences on the feedback modalities, but that users could successfully use TGV to scan QR codes on tactile graphics.
- In Chapter 6, StructJumper, I describe the design and evaluation of StructJumper, an Eclipse plug-in that created a hierarchal tree based on a modified abstract syntax tree that could be used to navigate through the code and look up contextual information. I report on the results of the evaluation which showed that there was a trend that users were faster completing the tasks with StructJumper and there were multiple situations where they thought it would be useful.
- In Chapter 7, Dynamic Graphs, I describe the design and evaluation of an interaction modality to improve the accessibility of dynamic graphs by placing the focus at the same location in the past or future version of the graph instead of the last visited. I report on the results of my evaluation which show that users could successfully answer questions about the changes in graphs and wanted to have access to multiple modes in the tool.
- In Chapter 8, Contributions and Future Work, I summarize the contributions of my dissertation, discuss the limitations of my research and highlight areas for future work.

Chapter 2. RELATED WORK

In this chapter, I will highlight the related work in a number of areas relating to making computer science more accessible to blind programmers. I will highlight the work done 1) investigating the practices of blind programmers, 2) improving the education of blind programmers, 3) creating tools for blind programmers, and 4) making diagrams more accessible.

2.1 PRACTICES AND CHALLENGES FOR BLIND PROGRAMMERS

The space of blind developer tools and the investigation of the blind developer programming practices is still a relatively unexplored field. Mealin and Murphy-Hill [57] interviewed eight blind developers and highlighted several practices employed by and challenges faced by blind developers. They found that despite issues with integrating screen readers with the complexities of integrated development environments (IDEs), five of the eight blind developers had used an IDE such as Eclipse or Visual Studio. However, the researchers found that blind developers rarely used and were not aware of the tools available to them within these complex IDEs. They also highlighted many of the practices that are employed by blind software developers, such as having a temporary text buffer to store notes in and also to work in [57]. More recently, Albusays and Ludi [2] did a survey of 69 blind programmers to discover some of the challenges faced by blind programmers and the workarounds that they use. Some of the challenges included the inaccessibility of IDEs, debugging, interface layout, code navigation, and diagrams.

Armaly and McMillan [4] did a study tracking the areas of the code focused on by blind programmers, sighted programmers, and sighted programmers who were blinded via turning off the display to see if there were differences in the parts of codes focused on by different groups and if it affected code comprehension. They found that there was little difference between sighted

programmers and blind programs in the areas of code that they focused on. There also was no difference in their comprehension of the program. However, there was a difference between those groups and the sighted programmers who were blinded that showed that there is an initial learning phase of using a screen reader that affects the programmer's comprehension of code and changes what parts of the code they focus on.

The prior work in this area has focused primarily on advanced programmers who have already learned to code. However, the challenges faced by students learning to program may be different as there may be challenges that are specific to education. In Chapter 3, I present my investigation the barriers in education for blind students to determine if there are any barriers that have not been presented in prior work.

2.2 Computer Science Education for Blind Programmers

There has been some prior work describing the curriculum and accommodations that educators have used when working with blind students. This work has either focused on tools created for education or the perspective of the educator on what they needed to change in their curriculums. It has not focused on the students' perspective.

Much of the work has focused on the introduction to programming. Stefik et al. developed a curriculum for a programming camp for blind high schoolers [79]. In this camp, they used physical objects just as dice and switches to introduce concepts like integers and Booleans and hand-on projects to practice the concepts. Another group created Audio Programming Language (APL), a new programming language specifically designed to help teach people who are blind how to program [70]. This language was designed for and by blind programmers and used variables that stored sounds to allow programs manipulating sounds. The environment was designed to be simple and users could access all the actions from a list.

Lego robots are popular choice to introduce K-12 students to programming as there are many programs like the FIRST Lego league. Therefore, some researchers have focused on making these environments and robots more accessible. Howard et al. [40] created new ways to provide feedback so that blind students could use the Lego Mindstorms NXT and receive feedback whether their code was working correctly. They evaluated three types of feedback: 1) haptic feedback that was implemented with a Wii mote and different vibration patterns correlated to different events. 2) Auditory feedback that correlated short music files with specific actions and 3) verbal feedback that summarized the actions taken and whether an error occurred and found that students preferred haptic and auditory feedback. Ludi [53] focused on the accessibility of the programming environment to work with the robots. She first identified the accessibility issues of current environments and then developed her own programming environment to ensure that it was accessible.

Other groups have focused on developing projects that are exciting and already accessible using existing technology. These projects serve as a blind student's introduction to computer science. Kane and Bigham [47] introduced the students to computer science using code to analyze Twitter data and make 3-D printings. Bigham et al. [12] put on a workshop at the National Federation of the Blind Youth Slam where the students made their own chatbots with unique personalities. McMillan and Rodda-Tyler [56] created a mentorship course where university students would partner with blind or low vision high school students and work with them to complete a computer science project through both one-on-one meetings and email communications.

There have also been multiple reports from instructors on their experiences teaching blind students computer science. Connelly [25] discussed the specific changes that he made to each of

the courses that he had the blind student in class. These were changes such as creating an audio header file to make the standard input and output streams audio based using text to speech, relying on a tutor/partner to explain the changes happening in assembly code, and providing some of the answers for worksheets. Francioni and Smith [31] instead took a holistic view of how they made the necessary adaptions for the blind students they had, discussing topics like how to make diagrams more accessible and the JavaSpeak system which they had developed with Matzek as an initial version of in the past [75] and improved upon to help provide information about a program's structure.

Much of the work in this area has focused heavily on the introductory materials for computer science, including both curriculums and tools. However, there has been little work looking at the changes necessary for upper level courses. While there have been a few educators that have discussed the changes they made for their courses for their blind students, we only see the educators perspectives of what they changed and it is a small sample size as they typically only work with one blind student. In Chapter 3, I present the results of my survey and interviews with blind students who studied computer science. By getting feedback from many students that have completed their degrees, I am able to get a broader picture of the changes that are made at all points in the university education. Additionally, as I focused on students' perspectives instead of educators', I could also discover information about situations where changes were not made, but would have been useful.

2.3 Programming Tools for Blind Programmers

There have been many groups that have worked to create tools for blind developers. Many have focused on adding additional auditory cues for tasks such as debugging. Stefik et al. created Sodbeans, a new programming IDE, which relies on audio cues to convey information such as

complier errors or changing the values of variables while debugging. Sodbeans' auditory cues are built on three principles that I have applied to the screen reader cues given in the work in my dissertation: 1) they are short, 2) they are "browsable" (i.e. a user can browse through the cues by only listening to the beginning of each cue), and 3) the important information comes first [79]. Stefik at al. [77] also created a debugging tool for Microsoft's Visual Studio IDE, which used sonification (non-speech audio) to aid developers. In a feasibility study, they found that developers were able to have 86% accuracy on the tasks using the sonification cues. Although not designing specifically for blind developers, Vickers and Alty [83] also added auditory cues to debugging tools by mapping the entry, exit and evaluation of program constructs (if, while, for, etc.) in Pascal to different musical cues. They found that sighted people learning programming found this useful for finding bugs. Although the authors found that the audio cues were useful, they only used a small number of cues to map onto a small number of constructs. Stefik et al. explored the use of audio cues to indicate the lexical scoping relationship between program statements [78]. These relationships were determined dynamically and the cues played when a change in scope was detected as the program executed. Ludi et al. [54] investigated using speech, earcons (audio tones, etc.), and spearcons (sped-up speech) to navigate and understand blocks based programming languages like Blockly. They found that while users did not like the earcons, spearcons performed similarly to speech, while being able to be conveyed faster than speech.

There has also been work on navigating through large projects in a non-visual manner. Smith et al. [74] wrote an Eclipse plug-in to navigate the hierarchal structure of files in the Eclipse IDE and found that both blind programmers and sighted programmers who could not see the screen found it to be useful. This work helped guide the work I present in Chapter 6.

The work in this section has primarily focused on creating new ways to present the information that is available to sighted programmers to blind programmers. As blind programmers already get a lot of information through speech, the work in this section looked at how to optimize providing speech feedback and looked at new modalities of feedback such as sonification (non-speech audio) and adding navigable structure to the information. My work in Chapters 6 & 7, has used the second approach, adding navigable structure to the information, to present additional information to blind programmers.

2.4 Making Graphics Accessible to Blind Students

Graphics are used frequently in computer science and are used to explain many concepts like data structures and graphs to students. In this section, I will focus on the two most common approaches to make graphics accessible, tactile graphics and computer based systems.

2.4.1 *Tactile Graphics*

Creating tactile representations is a common way to provide access to graphics. These graphics can be made using a variety of materials from gluing materials such as yarn and spaghetti on paper to using an embosser. They allow a blind student to explore the graphic tactilely and provide the student with the spatial information about the graphic. The labels in these graphics are typically placed in Braille.

One of the challenges that exists for tactile graphics is the placement of labels in Braille. The placement can pose problems for both students and teachers. Teachers, in particular, had issues placing the labels without text overlapping the figure [71]. Students struggled with associating the label with a specific part of the figure when it stretched across the entire graphic [3].

Despite the issues with using only Braille for accessing text, there is little work in the HCI literature using alternative methods. There has been some progress made in the access technology community. Touch Graphics developed the Talking Tactile Tablet (TTT) [51], a touch-sensitive tablet on which a user could place a tactile graphic and hear audio information upon touch. However, this method required a large touch sensitive surface (~12×15 inches, 6.5 pounds) and had to be connected to a computer via USB, which contained the information for the tactile graphic to be explored. Touch Graphics also created the Talking Tactile Pen (TTP) [52], which allowed blind users to access information on custom tactile graphics tagged with a proprietary code. The pen contained a small camera used to photograph the proprietary codes. When the pen contacted a tagged area, it read aloud the corresponding file stored on the pen. Despite the pen's portability in comparison to the TTT, it is a specialized device, and is only useful on properly tagged tactile graphics that have their information stored on the pen. TGV, discussed in Chapter 5, is a solution that attempts to solve the same problem by using non-proprietary codes and a non-specialized, portable, mainstream device like a smartphone. This provides a lower cost method of access as it relies primarily on devices that users are likely to already own.

2.4.2 *Computer & Tablet Based Systems*

When seeking to make graphics more accessible to users, there has been a lot of work that has used computer and tablet based systems to present visual information. Research in this area has worked to provide new ways to allow users to determine spatial information, such as Giudice et al. who found that blind users could determine spatial relationships using a vibro-audio tablet interface [33]. As there are many types of graphs which require different interventions to make them accessible on the computer or tablets, in this section I focus on the graphs which are showing the relationships of data using nodes and edges, such as data structures, UML diagrams, or generic

graphs. Much of the work presented focused on understanding a single graph. I seek to augment prior work by exploring a new dimension of information that needs to be provided to blind users of graphs, temporal changes. The work done to improve accessibility of these types of graphs is summarized below.

2.4.2.1 *Single Graph Exploration*

Much of the prior work in creating accessible diagrams has focused on how we can provide users an understanding of the layout of the graph or how we can support using the graphs to look up specific information. Layout information can be important as the location of an element on the page can also have meaning that is either not conveyed at all or not conveyed easily to blind users. Additionally, it is important that users are able to use the graphs to easily draw conclusions and answer questions about the data.

Many people have investigated the use of touch and pen input to provide ways for blind users to explore and receive audio feedback on graphs. Kennel developed Audiograf [49], a solution that had a touch panel that would provide information about the diagram where the person was touching on the panel. Users could get different types of information by using different pressure levels on the panel. Another group developed PLUMB [21,22,24]. This system uses pen-based tablet PCs and provided audio feedback as users explored with the pen. Information was read as users entered and exited nodes and sonification was used to help users follow edges between nodes.

The work discussed above focuses on conveying the spatial information to blind students to allow them to understand the additional context conveyed by the location of nodes. But it is also important for the creation of visually decipherable graphs. As it is common for a blind student to

need to share their graphs with sighted peers or instructors, Balik et al. investigated techniques to allow blind students to create visual graphs to share using a grid system layouts [10,11].

Some researchers have focused on providing a quick way to get an overview of the graph. Some groups have looked at using sonification to provide access to graphs to blind users (e.g. [84]). Sonification allows users to quickly perceive information such as trend direction or number of direction changes. Others have focused on trying to provide an overview by generating automatic text summaries of the graphs (e.g. [30]). However, thus far this work has focused more on the bar charts and line graphs types of graphs and not on the type with vertices and edges that we are working with.

In order to access data flow information, Blenkhorn and Evans [15] created a system called Kevin that allowed users to access information by reformatting it so that users could use a generic tactile overlay. The data flow diagram was turned into an N^2 chart that was the format of the tactile overlay. The users could then explore the tactile overlay and have information relative to the graph loaded read aloud based on what section of the overlay they were exploring.

In their development of GSK, Balik, et al. tried to improve the experience of using the graphs to answer specific questions. The traditional use of Excel to present graphs did not allow users to easily complete tasks such as following a path in the graph or finding common neighbors. GSK was designed to improve the experience of these types of tasks [10].

The work in this section provides insights of different ways to design the within graph navigation. In the creation of a dynamic graph tool in Chapter 7, I based the navigation within a graph heavily on GSK [10,11]. I choose this tool to be the basis because it did not require any specialized hardware (touch screen, pen, etc.) and they had promising results in answering questions using the graphs.

2.4.2.2 *Temporal Changes*

As the group responsible for PLUMB iterated on their design to create PLUMB EXTRA[3], they identified the need for animation of the graphs for courses like data structures [21] to help explain concepts such as inserting a node in a linked list. For these animations, users could explore a graph and then apply an operation (e.g. insert) and explore the updated graph. The authors did not provide any information about how they handled the focus change when moving between graphs.

While there has not been a lot of work looking at how to make the changes in a graph clear to blind students, there has been a lot of research in this area for sighted students. There have been many systems developed that provide algorithm animations (e.g. [26,36,37,38]). These systems allow students to see algorithms applied to a dataset (e.g. watch a series of bricks be sorted using various sorting algorithms) and learn from them. And research has shown that when students are actively interacting with the animations, they do help learning [41].

As they are valuable and used in classes, it is important to study how we make these temporal changes accessible to blind students. Therefore, I believe that we need to understand not only how we make navigation within a graph more accessible, but also navigation between graphs, which has not been studied. In Chapter 7, I discuss my investigation of the effect of where the user's focus is placed when they move to a new graph.

Chapter 3. EDUCATIONAL EXPERIENCES OF BLIND PROGRAMMERS

Much of the academic literature focusing on blind programmers thus far has focused on understanding the barriers in either introductory computer science courses [79] or the experiences of these engineers in industry [57]. Therefore, in this chapter I focus my research on the period between the two, as blind students are getting their degree. We know that students with disabilities are less likely to study Science, Technology, Engineering, & Math (STEM) fields and more likely to drop out of STEM programs out than their non-disabled peers [20]. Based on this, I seek to answer *RQ1: What are the barriers that can prevent someone who is blind from studying computer science?* This chapter is in response to the many informal conversations I have had with blind programmers and the mentions of challenges that they have faced in the classroom. I conducted a survey and follow-up interviews with blind programmers who had completed their degree in computer science and related fields to get their perspectives on the accommodations (or lack of accommodations) provided. My findings highlighted a variety of barriers blind programmers faced in college and the impact that they had on that student's ability to succeed in the field. [4]

3.1 INTRODUCTION

Increasing diversity has many benefits, however computer science is not a very diverse field [46]. In reaction, several diversity efforts have sprung up to raise participation. While these efforts are necessary, most have focused on increasing the participation of women and minorities, but few have considered disability as a part of diversity. Notably, AccessComputing [1] has attempted to increase the participation of people with disabilities in computer science by connecting students to

[4] The work in this chapter was done in collaboration with Cynthia Bennett and Richard Ladner

successful professionals and funding conference travel and internship experiences. Though AccessComputing has helped numerous disabled students obtain careers, the program does not penetrate the day-to-day classroom environments that blind students must encounter to complete computer science coursework. To better help organizations like these, it is important to understand the challenges that people with disabilities must overcome as part of the computer science field. My work focuses on the challenges faced by blind programmers.

Prior work has identified a few potential barriers that exist. Mealin and Murphy-Hill [57] found that many blind software engineers were not using many of the tools in the integrated development environments (IDEs) and hypothesized that it was due to the students not learning them in school. Additionally, in a blog post about the tools used by blind programmers, Doustdar mentions that he chose to continue his programming education outside of a university as it presented many challenges as courses were often taught visually [27]. I wish to discover if there were barriers in education that are unique to their studies during university and therefore had not been found by prior work.

While this research focuses on blind programmers who successfully completed their undergraduate degrees, the barriers they faced and overcame may give us insights into the barriers that prevented other students from completing their degrees. Using the insights gained from this work, I may be able to identify improvements that will help future blind students who wish to study computer science.

3.2 METHOD

3.2.1 *Survey and Interviews*

I conducted a survey and follow-up semi-structured interviews with a subset of the survey participants. The initial survey was used to gain a broad overview the problems they encountered during their degree, in general CS resources, and with IDEs. The interview questions were gleaned from survey results as I hoped to gain a deeper understanding of participants' responses and experiences. The interviews took 30 minutes to an hour and were transcribed for open coding and analysis afterwards.

3.2.2 *Participants*

I recruited survey participants using an international mailing list for blind programmers, research contacts with connections to blind programmers and snowball sampling. On the survey, participants were given the option to indicate if they would be willing to participate in the follow-up interview. Inclusion criteria for participating in the study were that they completed an undergraduate degree in computer science or a related field, they used a screen reader while completing the degree, and they were 18 or older. After closing the survey, I removed any responses that were likely spam. This resulted in 15 complete responses.

All fifteen survey respondents were male. The average age was 31.7 (SD = 6.9). Six participants had a graduate degree. Ten participants did their degree in North America, the other five in Europe. The median graduation year was 2009 (range 1995-2014). Participants reported an average of 12 years of computer science experience (SD = 7.5).

For the interviews, I selected from the survey participants who indicated they would be willing to participate in a follow-up interview. I conducted the interviews with 10 of the survey

respondents (referred to as i1-i10). I selected interview participants to get a broad spectrum in terms of when they graduated, whether they got a graduate degree, and their responses to the survey.

The average age of the interview participants was 29.2 (SD = 5.1). Four of the participants had received a graduate degree. Six of the participants did their degree in North America, the other four in Europe. The median graduation year was 2009.5 (range 1995-2014). Participants reported an average of 12.3 years of computer science experience (SD = 7.8).

3.2.3 *Analysis*

I recorded and transcribed the interviews. After reviewing all the transcripts, myself and a collaborator used open coding with the two of us initially coding together to establish the code book (Appendix B). Once the code book was established, we coded independently the remaining interviews. Once all interviews were coded, we reviewed the interviews coded by the other, added any codes that were missed, and arbitrated any disagreement regarding the definitions of codes.

3.3 Findings

I present my findings according to three main areas: technology, formal education, and informal education. Formal education refers to the learning done in university that was related to their degree. Informal education refers to learning done by the participants outside of their degree program. In each of these areas, I address the barriers that blind programmers encountered.

3.3.1 *Technology*

This section details the current state of the IDEs and other technologies that are commonly used by participants and how they relate to their education in computer science.

On the survey, participants were asked to detail their preferred set-up for coding (or set-ups if they varied by programming language). The four most common IDEs were Visual Studio (seven participants), Eclipse (six participants), Notepad (three participants), and Notepad++ (three participants). Though IDEs and editors designed with blind programmers in mind have been created (EdSharp [28], Emacspeak [68], and Sodbeans [79]) only four participants indicated that one of these IDEs was one of their primary set-ups. JAWS was the most common screen reader listed (eight participants), followed by NVDA (three participants), Window Eyes (two participants), and Voiceover (2 participants). Four of the participants did not list which screen reader they used. Finally, five participants used a Braille display, while three did not. The other seven participants did not indicate explicitly whether or not they used a Braille display.

Similarly to Mealin and Murphy-Hill [57], I found that many participants encountered issues with IDEs as many were inaccessible. The accessibility challenges ranged from completely unusable to just some of the advanced features not being usable. All but three respondents to the survey indicated that they had accessibility challenges with the IDEs that they used most often. Inaccessible features indicated by users on the survey were interface builders (6 participants), debuggers (3 participants), syntax highlighting (2 participants), and diagrams (2 participants).

Another major challenge that participants noted was that while an IDE and its features may be accessible, learning how to use it was much more complex. Most guides for IDEs that participants attempted to use for assistance were geared toward sighted users, often littered with directions like “click here” which made them difficult to use, if not unusable, to a blind programmer. Determining keyboard equivalents was complex, often leading participants away from assignments in search of accessible tutorials with relevant information, asking other blind programmers, or spending extra time exploring on their own. This challenge was compounded

when participants recounted having to learn how to use an IDE in introductory courses when they did not yet understand basic programming concepts.

Attempting, and failing, to use inaccessible IDEs deterred some participants from trying new ones. One participant said:

> *And like my confidence was pretty low that these programs would even work if I had the time to spend. So I was kind of going under this assumption that they wouldn't anyway. And I think I would have been right in most cases, but I don't know. ... So I didn't have time to spend and you know, put like 40 hours into Eclipse and then learn, oh ok cool, you know, it's not accessible. - i1*

The burden of spending so much extra time to no avail stuck with him to the point that potentially wrongfully assuming an IDE would be inaccessible made more sense to him and his time. Similarly, i8 would request to use a different language with an environment he was already familiar with. This allowed him to remove the burden of learning a new language and IDE, which could be very time consuming for him, and focus on learning the concepts.

For some participants like i8, choosing to avoid learning new IDEs at the university was a purposeful choice that, in the short term, mitigated some of the overhead skill acquisition while keeping up with their classes. However, such choices came with tradeoffs, in this case, tradeoffs that went unrealized for years. At the time of the interview i8 was learning tools like the debugger that he had never used in college.

3.3.2 *University Learning*

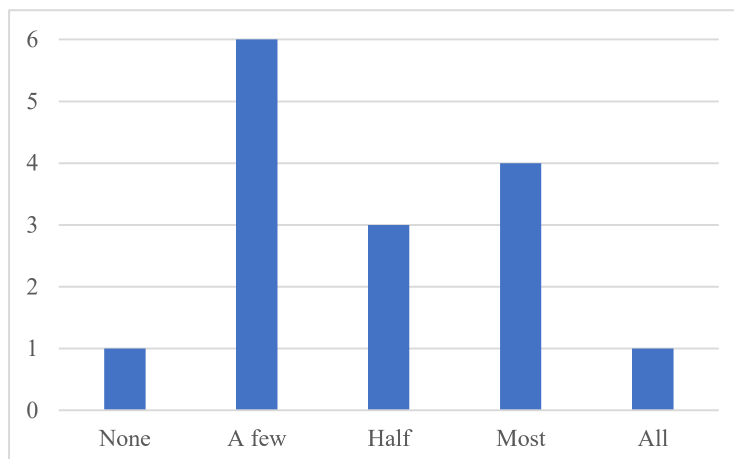


Figure 3.1. Survey respondents were asked what proportion of their classes had inaccessible portions. This chart shows how many survey respondents selected each proportion

The results of the survey and interviews showed that there were a variety of issues that students faced while completing their degree. These barriers could come in the form of materials, assignments, or from the faculty themselves, but they were very common. In the survey, I found that 8 of the 15 respondents said that at least half their classes had inaccessible portions (Figure 3.1). In the following sections, I will describe the accessibility challenges the students faced during their degree and the impact that they had.

3.3.2.1 *Lecture*

One of the major problems blind students faced in lecture was that there was often missing context in what was spoken and it was only available visually:

> *So the professor would say something like sizeof int, right. So he would leave out that it was actually sizeof left parenthesis int right parenthesis semi-colon. … So you miss all that. So when I first started learning C in college, I would actually write sizeof space int semi-colon on my exams. –i2*

Issues such as this are compounded when the lecturer is writing on the board and then pointing to something and not reading it out. I2 acknowledged that it was a new experience for the lecturer and would try to sit as close to the lecturer as possible to serve as a reminder that they needed to verbalize what was on the board. When the professor did not verbalize what they are writing and did not have slides or other materials the student can access, some students were left relying on friends in the class whose notes might use equation editors that were hard to use with screen readers or were handwritten and required another person to dictate them to the student. Some participants found that this issue can be mitigated somewhat if they had access to the slides in an accessible format during lecture to follow along with.

3.3.2.2 *Materials*

One thing I found was common was the need for alternate formats for many of the materials that were provided for the class. In particular, there are three types of materials that I will focus on in this work: books, diagrams, and math content. I focused on these three materials in the interviews based on the results from the survey where they were the most common examples of materials that were not fully accessible in classes.

For books, they tended to be in one of a few formats: Braille, audio, or electronic format. Each of these had their benefits and downsides as explained below.

Braille books tended to take a long time to produce and were expensive to create. For those who received Braille textbooks, there could be large delays in getting the book as they took months to create. I2 was getting the materials made 2 months in advance in order to have them ready for when classes started. For i1, books were extremely delayed and there were times when he would not get the book until past half way through the term, if at all. To have access to the book for the entire term, he would sometimes choose to drop a course and then retake it at a later date by which

the textbook should be ready. On the other hand, i4 got books from a Braille library. They often were not the same book as the rest of the class, but the faculty would look them over and determine if they adequately covered the same topics.

Multiple (i2, i4, i7, i8) participants used audiobooks from sites like Learning Ally[5] (previously RFB&D). Books from Learning Ally were appreciated because they had readers who were versed in the subject. This had two benefits. The first was that they could read the subject material in a way that made sense. For things like code, they would know what needed to be read, such as braces. Additionally, they would often describe the diagrams and would know what the pertinent information was. But they also had they downside. One participant found the spoken code very hard to listen too:

So, books and audio when it comes to programming is very inefficient. Some of the readers, what they would do, is they would actually spell the entire code. So they would say f-u-n-c-t-i-o-n. Space... - i2

There was also some context that could still be missing such as spelling and formatting Additionally, it was not possible to search audiobooks, which made them less useful for certain tasks such as referencing a specific equation or definition.

Electronic formats were preferred by some participants as they provided the student with options on how they could access the book. The student could use text-to-speech and listen to the books or they could also use refreshable Braille displays to access it as well. Three of the interview participants (i1, i5, i6) tried to contact the publishers directly to get electronic formats of the books, but did not have success. For those who were unable to get the electronic format from the publisher,

[5] https://www.learningally.org/

another option was to scan the book and use OCR on it. For participants that had support for the errors to be corrected (i7), this was a good option for them. However, if the student was unable to get the OCR corrected (i3, i5, i6), then they received the text with errors in it. These errors meant the books were not as usable and hurt the students' ability to learn from them. One participant described the effect as:

Then scan was always the last scenario because that would have a drastic effect on my mark. ... to the point in [university] where I was getting First for the things where I had electronic copies of books readily available to me, but I was barely passing modules where I had to get the book scanned. - i6

For diagrams, I will look at both how the diagrams are provided to the student and how they are produced by the student. Diagrams were provided in a variety of different formats to the students. In general, they were either provided in a tactile form or a textual description. For those that were provided in text, it varied who provided the descriptions. For one student, the professor included descriptions on his slides. For others, they relied on friends or family members to describe the images. One of the issues that arose with having to find people to describe the diagrams was that it was important for the people who are assisting a blind student to be knowledgeable of the field:

And it's kind of like well, ok, I need to know where the switches are and stuff like that. ... But, you know, if you don't know about computers, you can't just look at some of these diagrams and give an accurate description of them. So it was more an issue with getting people to describe things to me I think. - i6

It was important to have the person describing the graph be aware of the field for a variety of reasons. They needed to understand which information needs to be presented to the student and which can be ignored. As the student was still learning the content, they were not able to provide the reader any information about what they need to be told either. This is one of the areas where Learning Ally did well as they had readers with expertise.

For producing the diagrams, students would generally describe them using text. A few of the students developed their own notation that they could use for the types of diagrams they were learning.

As many CS programs have significant math requirements, many of the students took multiple math courses. For accessing the math, there were a couple of ways. Students would sometimes get access to it in Braille. One student's professor even starting learning Braille to help produce the Braille for them. Another common format was TeX or LaTeX. A third way some students accessed math was from someone dictating the math to them. This was not a pleasant solution for some students:

> *As far as the assignments, thankfully all of the problems that were assigned were optional. And given how challenging and time consuming it was to just deal with the proctors, I opted not to do them. – i5*

Much like with the diagrams, I saw the need for people who were aware of the field to be important for dictating math. Having someone who knew the content was important as there is jargon that is used to communicate many of the math topics. If the person dictating/transcribing does not know the jargon, the blind student has to communicate in ways that are different from how they would in class, which can put an extra burden on them. Additionally, if the person is

reading the content to the student and is not familiar with it, they may not communicate it as well. I9 made sure that he only asked questions of people that already knew the content.

When the students produced their own math, I saw that they used similar techniques. In some cases, they would use different techniques for themselves than what they would provide the faculty member. For instance, they might do the work for a problem in Braille, but then dictate it back to the professor.

3.3.2.3 *Assignments*

I saw that many of the participants completed alternate assignments during the course of their time in university. While much rarer, some instructors excused the students from assignments. Some examples of how instructors handled inaccessible assignments will be detailed below.

Many instructors altered the assignments for the students in some form to remove the accessibility barrier. One common alteration was changing the form factor of the assignment. This often meant that they were doing nearly the same assignment, but with a slight change. The most common examples of this was allowing students to write text descriptions or use a text based notation for diagrams instead of creating the visual diagram (i5 and i9) or doing a console based application instead of GUI application (i3, i6, i10).

Another tactic was giving a totally different assignment. One such example was when the rest of the class was doing a visual game, i9 did a chat application. This application "covered some common concepts and others which the game did not. So it was equal but not the same." – i9.

Some schools chose not to remove the accessibility barrier from the assignment. There were two examples of how this could play out in the assignment. For i2, the professor decided just to take it into account when grading. A situation like this came up at least twice for this student.

The student was okay with this outcome as for at least one of the assignments as it was toward the end of the quarter when he had already learned most of the content.

The other case was having a sighted student or aid help the student with the visual parts of the assignments. One student encountered this when creating a drawing application. He had a sighted peer help him by watching to see if it succeeded, but he was not very motivated by the assignment:

And to be completely honest, I didn't put a huge amount of effort into that assignment. Because I just hated the idea of kind of spending hours and hours of making something I'd just never be able to use. –i6

The student's motivation was decreased by having an assignment that he would be unable to use (and debug). The lack of effort meant that he didn't learn the underlying concepts as well which can be a problem as computer science often builds off topics taught in earlier courses.

Another problem was that many schools have done some of the work for student by configuring lab machines. Many time these machines were locked down and could not be used by the blind students:

But the problem was that you could only access that server from the lab machines and the lab machines didn't have any speech on them. So I couldn't use it all. So I actually – I ended up having to set up my own machine and you know set-up Oracle, set up PHP, Apache, and all things like that.. – i6

Challenges like this put an extra burden on the blind student. They had to do extra work to succeed at the same level as a sighted peer. In the case of i6, he found the experience of setting up his own

machine to host the server valuable as it taught him skills he later used in his job, but it did put extra stress on him.

Accessibility barriers also cropped up in group projects. One students had a group project that required creating diagrams and it created barriers to participating in the project:

But even so, it wasn't ideal. I mean they were understanding, but, you know, there were lots of sessions where they'd all be talking and I wouldn't be able to say anything at all. Not because I didn't want to, but just because I couldn't really follow what they were talking about. - i6

The students could not follow the real-time creation of these diagrams and therefore could not participate in the discussions. This can have further consequences than just their participation in the work the group was doing. Being unable to participate in parts of the group projects was an isolating experience for the student which carried over into his social life as he was not invited when the rest of the group hung out.

3.3.2.4 *People*

I found in my interviews that faculty can have a huge impact on the success of a student in both a positive and negative manner. As one participant stated,

You need both individual buy-in and you also need organizational buy-in from the university. Otherwise you can actually have the best and brightest student in the class who happens to be blind and if they can't get the requisite support, then they've got a huge, huge, huge problem. - i4

My other interviews also showed this to be true. I found many instances where the student did not have a good experience due to the lack of buy in from the faculty. It manifested itself in instances where students were unable or talked out of taking certain courses. For instance, i1 was interested in taking a course that the professor was uncertain how he could complete it due to the large number of diagrams. Instead the student took an independent study version of the course. The experience was further degraded by the fact that the professor only provided hand written notes that OCR worked poorly on and there was no textbook for the student to reference in its place. At the conclusion of the independent study course, he did not feel he had learned much and definitely not the same material as what the other students were learning. Another participant, i6 was talked out of taking a course he was interested in because the professor was unsure how they would be able to complete the course.

Faculty were also often responsible for providing students with the materials. So, in the worst case faculty would refuse to provide the materials, such as lecture slides, to the student. This required intervention from by the university. But in the best case, faculty would make sure the student got the materials ahead of time and made sure there were text descriptions of the diagrams.

Beyond materials, the general attitude of the faculty was a hindrance for some students. As the subjects were very visual, some students received some push back indicating that maybe they should major in subjects that were less visual and had less technology. Some faculty attributed any lack of success to the student’s ability to succeed and not to other problems such as lack of access to course materials. On the flipside, many faculty members were supportive and provided the students the support and accommodations that were necessary for them to do well.

3.3.2.5 *Prior Experience*

It is becoming more and more common that students have experience in computer science before they start university. Due to this, I asked the interviewees about how much experience they had and how it affected their ability to complete the degree.

Many of the interview participants had prior experience coming into university. For 4 of the participants this prior experience was important, if not essential, to their success in completing their degree. One participant put it as:

> *I would say for a blind person, not even beneficial, I would say it's absolutely necessary for you to have some sort of programming experience before you start something like a college level computer science degree. –i2*

The effect of the prior experience was often that it was mitigating some barrier that blind students faced. One of the barriers that prior experience was able to help with was learning the technology. As many IDEs are not accessible and those that are may not have a straightforward way to perform actions with a screen reader and keyboard, that means that it can add a lot of time for the student to learn the IDE. If they already have that experience coming into the university, it allows them to focus on the classes themselves and not have to do the extra work of finding an accessible IDE and learning to use it.

Additionally, students would sometimes not have access to all the materials ahead of lecture, such as the slides or the textbooks. In those cases, having experience really helped:

> *Yeah, definitely, because though, like I said there were a lot of times with the programming lectures where I perhaps didn't have access to the stuff or if I did it was*

going to be afterwards. ... You know, my existing knowledge kind of filled in some of the gaps. –i6

3.3.3 *Informal Learning*

Informal learning is an interesting area to consider, as students do not have the same resources as in a formal setting, such as a disability office which will work to make inaccessible materials accessible, and therefore accessibility challenges may differ. For this section, the survey and interview participants were asked to talk about their experiences up until the present.

3.3.3.1 *Resources*

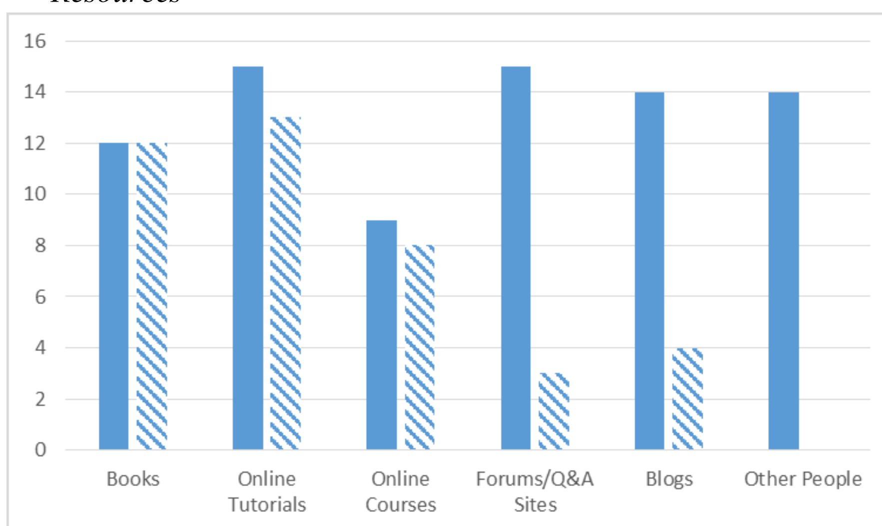


Figure 3.2. This chart shows the resources that the survey respondents have used to learn about topics in computer science. The solid bar is the resources they have used and the striped bar is the resources that they have used (or tried to use) and found accessibility problems with.

The blind programmers often used a variety of resources to learn new concepts in computer science, but many of them had accessibility problems (Figure 3.2). Many of them are challenges that have solutions that the author/creator chose not to implement. Common examples are putting in code or math as a screen shot, not having alt text for diagrams, or videos with poor descriptions or no audio. Another type of problems some people encountered was inaccessible web editors. For

instance, one participant described an online editor that did not disable the screen reader hot keys, which rendered the user unable to type in editor as it would navigate the web page anytime a hot key was pressed.

While accessibility challenges were common, it rarely prevented people from accessing the materials that they need.

I've never been in a situation where I wanted to learn something and I've simply not been able to do it. But there have been situations that I've been in where I wanted to learn stuff and I've kind of looked at tutorials and the first few have been inaccessible. –i6

As there are a lot of resources available on the internet, many of the people would just move on to a different resource if the first they found was not accessible. If it was only a small portion of the content, the person may just choose to skip that part.

3.3.3.2 *People*

All but two of the people I interviewed indicated that they had used accessibility specific mailing lists as a resource, though some were no longer using them in the present. While the rate of participation may be higher than typical as mailing lists were one of the forms of recruitment, there are some interesting takeaways that I will address in this section.

As many of the IDEs are difficult to use via the keyboard, many of my participants found that learning tricks from other blind programmers was helpful. Many times, setting up and getting to know a new environment is difficult and the mailing lists may provide a more efficient way to complete a task. In general, these were used for accessibility specific questions.

One interesting strategy that I encountered from one of my interviews was to use general programming mailing lists to solve accessibility challenges that he faced:

But sometimes I will ask, you know, I'm a VoiceOver user, this is not working at all, how do you sighted people do this so that I can try to figure out how to replicate what you're doing with Voiceover. And then we go back and forth. –i8

This was a unique strategy that one of the participants used to solved the accessibility challenges that came from not having a guide on how to perform actions via the keyboard. He used both mailing lists as well as Skyping with a sighted developer who would watch his screen to see what happened as he took certain keyboard actions and gave feedback on which step to try differently. Knowing from a sighted user that there was a hovering window for example gave him ideas on what he should try next.

3.4 Discussion

One of the major themes that came across in the results is that accessibility barriers can decrease motivation in many ways that can harm a student's chances to succeed in computer science education. It affects the students' technology use as they are not motivated to find new IDEs or features within the IDE when they are not sure that they will be accessible. There are potential side effects of always choosing a language and IDE that are familiar to the student. The computer science field is rapidly changing and unless general accessibility of IDEs improves, blind students may be less willing to explore the new technologies that come into play. There has been some work to create more accessible IDEs [28,68,79] and accessible tools within the IDE such as navigation [7,74] and debugging [78,83]. But more work needs to be done. There are a limited number of IDEs that are accessible which limits a blind programmer's options. As long as IDEs tend to not to be accessible and provide limited extra benefits, blind programmers are less likely to explore new IDEs.

Additionally, accessible assignments are important for motivating students to put in the work to learn the concepts they are trying to teach. If many of the assignments in the classes are inaccessible, it may turn them away from the field of computer science. And if they do stay, they may not learn the concepts as well. Stefik et al. [79] have been working on creating the curriculum with accessible assignments, which is beneficial for the introduction to computer science. However, these accessibility issues arise throughout the entire degree and faculty who have blind students in their classes need to consider how to make assignments accessible their classes. Burgstahler [19] has good advice on making classes accessible to blind students and others with disabilities.

Another major theme that reappeared throughout the interviews was the extra work that blind students had to do to have the same experience as the sighted students. From having to find accessible versions of the textbook to learning how to do things that are taught in class in a different manner, there were many occasions where blind students were doing more work than their sighted peers.

For this to improve, a few things need to be done. One thing that would be very helpful would be tutorials that describe how to do specific actions with the keyboard instead of the mouse. This would remove the burden of the blind student having to discover how they work for themselves.

Additionally, instructors need to be aware of how their assignments and lab set-ups can affect the blind students. Making sure that resources can either be accessed from any computer or ensuring that the student can use their assistive technology on the lab computers is one way to reduce the burden some students felt. Assignments that are inaccessible can mean that they are

having to find someone to act as a sighted assistant or they are having to find new solutions to access visual materials.

3.5 SUMMARY

For computer science to be a more diverse field, we need to have inclusive programs at the university level. My work shows that there are steps that need to be taken to make computer science more inclusive of blind students. They face many barriers from getting access to materials and assignments to lack of support of the faculty. These barriers can decrease their motivation to learn computer science and may turn them away from the field.

Chapter 4. EVALUATION OF IDES

In this Chapter I describe my preliminary accessibility evaluation of popular code editors and integrated development environments (IDEs). This is a follow-up investigation to *RQ1: What are the barriers that can prevent someone who is blind from studying computer science?* which I investigated in Chapter 2. In that work, I learned that many blind students had problems with code editors and IDEs being inaccessible. I investigate this issue in more depth by looking at the basic features of code editors and IDEs to determine if they are accessible. I found that many IDEs and code editors were completely unusable by screen reader users and all had at least some accessibility problems, even in just the basic features.

4.1 Methods

I chose to evaluate a variety of the most popular IDEs and code editors that are available for Windows for their accessibility to a novice programmer. I selected 6 of the most popular Windows IDEs and editors based on the results of a survey of the other IDEs and editors used by users of Codeanywhere [18] and based on Google analytics of the most searched for IDEs [14,81]. I chose to evaluate Windows based IDEs and editors because 85% of screen reader users use Windows [86]. IntelliJ, Visual Studio, Eclipse, NetBeans, Notepad++, and Sublime Text were the IDEs and editors evaluated. These evaluations were done in May of 2016.

For the evaluation, I tested the programs on Windows 10 with NVDA[6], a free, open source screen reader. To facilitate the evaluation, I turned on the speech viewer, which displays the text spoken aloud in a text box, to clarify the output to the actions. The evaluation was broken down into two sets of tasks, one set which was evaluated on all IDEs and editors. This included checking

[6] http://www.nvaccess.org/

that the screen reader was able to read the code typed in the editor, the menus were accessible, and code completion was accessible. If either of the first two were not accessible, the IDE/editor was deemed unusable and no further evaluation took place. The second set of tasks was running a trivial program created for testing purposes that included both input from and output to the console, checking for syntax errors, and basic debugging including setting breakpoints and running the program with the breakpoints. The second set of evaluations was done only on the IDEs as code editors would be run and debugged from the command line.

As I was testing for the accessibility for a novice programmer, I tested the output of the expected interactions of for these tasks. I discovered in my interviews that there are some features that may not be accessible using the expected interaction, but may be accessible using complex work arounds or there may be add-ons or advanced settings of the screen reader which will augment the accessibility of some IDEs. I did not search for these work arounds as they may not be found by many novice programmers and are a hindrance to the perceived accessibility of the IDE. Additionally, some of the accessibility challenges may be able to be solved with a different screen reader or configurations of downloaded software (e.g. Java Access Bridge). Similarly, I choose not to search for solutions as they are still barriers if a user had to do complex configurations or learn a new screen reader just to use the basic features of an IDE.

4.2 RESULTS

The results of the evaluation are broken down by IDE/editor. Overall, none of the IDEs/editors that I evaluated were accessible in all the features I evaluated, even when only looking at the basic features. Additionally, half the IDEs that I evaluated were completely unusable.

4.2.1 *IntelliJ*

The version I evaluated of IntelliJ was IntelliJ IDEA Community Edition 2016.1.2. IntelliJ was deemed to be completely unusable as the menus were completely inaccessible. Both on the landing page and in the editor itself, there was nothing spoken when accessing the menus.

4.2.2 *Visual Studio*

The version of Visual Studio I evaluated was Visual Studio Enterprise 2015, version 14.0.23107.D14REL.

Visual Studio's basic features (text and menus) were accessible. There was no problem getting the code to run via keyboard commands. One issue that did arise was that the test program both required input from a user and printed to the console. When running the program (not in debug mode) the console was not fully accessible. It would automatically read all the content as the program ran. However, for most of the runs it would not read the initial text that was on the dialog. It did read the initial text in the dialog for a few of the runs, but I was not able to replicate what caused this.

Code completion was partially accessible. The user could use the down arrow key to scroll through the options and it would read the autocomplete option and tell the user where in the list it was (i.e. 2 of 147). However, the first option could not be accessed initially. The first time the user selected the down arrow, it placed focus on the item, but did not read it. I had to arrow down to the second option and then arrow back up to the first option. If there was only one option in the list, then it was not accessible. Additionally, the extra information such as the method signature with the parameters was not accessible.

The screen reader did not give an indication when the user moved the cursor over areas of the code that were underlined due to a syntax error. However, users could access this information in other ways. After building the program, they were available in the output window and had information about the line of code and the error type. Additionally, they were available in the dedicated Error Window. This window had the advantage that a screen reader user could directly go from the error they are reading about to the editor window with the cursor updating to the location indicated for the error.

Debugging was not as accessible as the other features of the IDE. Users were able to set a breakpoint using the keyboard, but there was no indication as the user moves the cursor to a line with a breakpoint. The user could find out where the breakpoints were by opening the breakpoints window, which listed line number and files only. Running the code could also be done via the command line, but the dialog was not as accessible in debug mode and did not speak out what is printed. Additionally, when the code hit a breakpoint or as the code was stepped, there was nothing spoken unless it switched to a new file. The cursor location did update, so a user could figure out what line of the code it was on, but it required extra work. Finally, the window that showed the local variables was not accessible, so the user could not see the value of the variables at the different points in the code.

4.2.3 *Eclipse*

The version of Eclipse I evaluated was Eclipse IDE for Java Developers version Mars.2 Release (4.5.2). Eclipse's basic features were accessible. A user could both read what was typed and access the menus. There were some problems when navigating between the different windows. The focus would not always change as expected and the user would have to access the editor using a work around of going to the outline and then hitting enter to update the cursor to the selected function.

Running a program was not an issue. If the program running required the console for input or has output there was nothing spoken, but as long as the user was aware that something was on the console, they could navigate to it and it was accessible.

Code completion was accessible. In order to access the options, the user needed to tab to enter the dialog with the code completion options. A tab while one of the options was selected took the user to the dialog with extra information about that selection which they could then read. Syntax errors were not indicated to the screen reader as a user move their cursor through the code. However, there was a problems window that was accessible that listed the problem and where in the code it was. The user could also press enter and have their cursor be taken directly to the location in the code.

Debugging still had some accessibility problems. A user was able to set a breakpoint with the keyboard, but as they were on the line with a breakpoint, the screen reader did not give any indication of the breakpoint. The locations of the breakpoints could be accessed with another window that listed the line numbers of the breakpoints and enter would take the user to the specific line of code. Much like Visual Studio, as the program executed, the screen reader was silent. Understanding the state of the program required a lot of navigating between windows. The cursor location was updated in the editor, so the user could discover where they were in the code. Additionally, the windows with the variable and the stack execution were accessible, so they could access that information as well.

4.2.4 *NetBeans*

For my evaluation of NetBeans, I chose to evaluate NetBeans IDE 8.1 (Build 201510222201). While there has been work done by Stefik, et al. to improve the accessibility of the NetBeans via

adding a layer on top (Sodbeans) [79], this is not the version of NetBeans that most people and classes would be using. Therefore, I choose to evaluate the base version of NetBeans.

My evaluation found NetBeans to be completely unusable. Both the menus and the editor itself were unusable as nothing was spoken in the menus or in the editor window when trying to read a line of code.

4.2.5 *Notepad++*

The version I evaluated of Notepad++ was v6.9.1. The essential features were accessible for Notepad++ and the users were both able to type and read the code and the menus were accessible. However, the code completion was not accessible and did not speak the autocomplete options.

4.2.6 *Sublime Text*

The version of Sublime Text I evaluated was Sublime Text 3. My evaluation showed that Sublime Text was completely unusable. While the menus were accessible, the text was not. There was feedback as the user typed, but when trying to read the code that was written, there was no speech output.

4.3 SUMMARY

The results of my evaluations of integrated development environments (IDEs) and code editors showed that there were a lot of accessibility barriers. Half the environments were completely unusable with a screen reader and all environments had at least some problems. One common issue I saw for environments that were mostly accessible was that accessing much of the information that was available visually required much more work for the screen reader user. For items like syntax errors and break points, the information is available visually in the editor window,

but the screen reader could not access this information at that location. A user would have to open an extra window and navigate between the window and the editor to get the same information. Additionally, screen reader users needed to seek out updates, such as which break point was hit or if text had been added to the console, as they were not provided these updates automatically.

Chapter 5. TACTILE GRAPHICS WITH A VOICE

In this chapter, I seek to answer *RQ2: How can we provide access to graphics for people who are blind and do not know Braille?* This is an important problem as images are used heavily in computer science education and need to be made accessible to blind students. A common method to do this is by making the images tactile, however for those who do not know Braille, there is not a low-cost way to access the labels using mainstream devices. To provide this, I proposed Tactile Graphics with a Voice (TGV), which used QR codes to replace the text, as an alternative to Braille. The codes were read with a smartphone application. I evaluated the system with a longitudinal study where 10 blind and low-vision participants completed tasks using three different modes on the smartphone application: 1) no guidance, 2) verbal guidance, and 3) finger pointing guidance. My results showed that TGV was an effective way to access text in tactile graphics, especially for those blind users who are not fluent in Braille. I also found that preferences varied greatly across the modes, indicating that future work should support multiple modes. [7]

---

[7] The work in this chapter was done in collaboration with Lauren Milne, Jeffrey Scofield, Cynthia Bennett and Richard Ladner and is based on work previously published at ASSETS 2014 [8,9] and in TACCESS [6].

5.1 INTRODUCTION

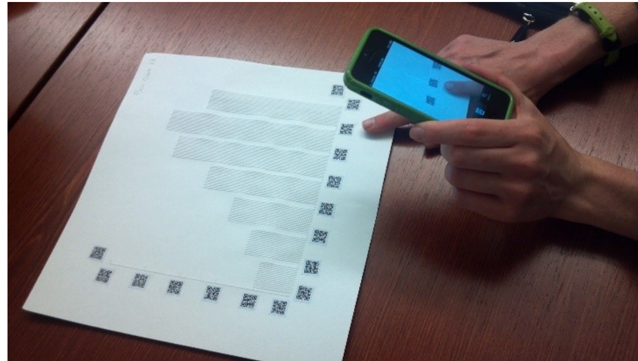


Figure 5.1. The Tactile Graphics with a Voice system in use. There is an embossed bar chart with QR codes for labels and the subject is using the finger pointing mode to select which QR code to scan.

From visualizations of data structures to UML diagrams, images are an integral part of most computer science textbooks, and frequently convey information that cannot be understood from text alone. Therefore, these images and the text contained within them should be accessible to all students, and there is a need to create alternative access methods for people with disabilities. The common solution when making a textbook accessible for blind students is to create tactile representations of the images, or tactile graphics. Tactile graphics can be low fidelity made by using craft supplies like spaghetti, pipe cleaners, and sand paper (see Figure 5.2) or a high-fidelity graphic printed on an embossing printer (see Figure 5.1). Studies have shown that tactile graphics are valuable for conveying graphical information [39]. In a survey of 24 teachers that worked with visually impaired children, all indicated that there were situations where tactile graphics were important for and effective at teaching a lesson [71]. Teachers also indicated that the ability to explore graphics, discover the information, and answer questions about the information independently was a fundamental part of the learning process [69,71].

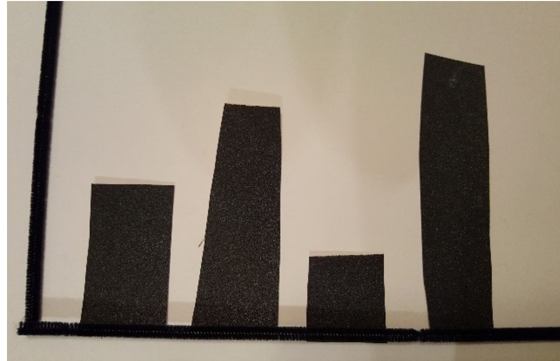


Figure 5.2. A sample low fidelity tactile graphic showing a bar chart made with pipe cleaners and sand paper.

The text in tactile graphics is typically represented using embossed Braille. However, a 2009 report by the National Federation of the Blind states that less than forty percent of the functionally blind population in the United States is fluent in Braille [59]. Therefore, tactile graphics with Braille labels are not accessible to a significant number of blind people.

There have been a few solutions to this problem presented by the access technology community. Examples include a system where an overlaid tactile graphic on a tablet gave audio feedback when touched [51] and a talking pen to explore a tactile graphic [52]. However, these solutions require using specialized devices, which can be expensive.

I present and discuss the development of a system for embedding and accessing text in tactile graphics using QR codes, which are small codes that directly encode textual information (Figure 5.1). QR codes can be read by a smartphone and can easily be created by anyone with access to a computer. I created a smartphone application for blind users called Tactile Graphics with a Voice (TGV) that scans QR codes and provided feedback to help users aim the smartphone camera. I conducted interviews and surveys with people who are blind or low vision to design the

application and determine what types of non-visual feedback were most helpful to aim the smartphone. In addition, I developed a finger pointing method to help determine which QR code should be read when there are multiple QR codes in the camera view.

I evaluated the application in a longitudinal study and found that people who are blind or low vision could successfully answer questions about tactile graphics by scanning QR codes. Key findings from the study are listed below.

(1) Four out of ten participants could correctly answer questions about the images using the QR codes, but were not able to use the Braille equivalents as they were not fluent in Braille.

(2) Participants fluent in Braille spent an equivalent amount of time on tasks and had similar accuracy for both the QR codes and Braille equivalents.

(3) Preferences varied greatly among participants as to what kind of feedback from the smartphone application is most helpful. Four participants preferred the Silent mode, four preferred the Finger Pointing mode and two preferred the Verbal mode.

5.2 RELATED WORK

Beyond the general work to make diagrams more accessible discussed in Chapter 2, I build upon related work in three areas related to the system: (i) methods to embed textual information on tactile graphics, (ii) methods to access the information and (iii) use of the finger pointing technique as a means to select which information to be read aloud.

5.2.1 *Accessing Textual Information on Tactile Graphics*

To embed the text on tactile graphics, I was interested in using QR codes. In the search for prior use of QR codes to encode information for the blind, I found that Voiceye codes were also being

used to encode text on graphics [39]. Voiceye codes were similar to QR codes, but could contain more information for a given area. Users scanned these codes with a smartphone application and the corresponding text appeared for reading aloud or visual magnification. While not used on tactile graphics, they were used in South Korea to make government forms accessible. However, users were not given feedback to assist in scanning the code and the codes must be created with expensive proprietary software. TGV provides a major benefit over current approaches because QR codes can be freely created.

5.2.2 *Camera Use By Blind People*

TGV required the use of a smartphone camera, because it enabled the use of QR codes. While aiming the smartphone was a challenging task for blind people, there are research efforts in the accessibility community to tackle this problem. Bigham et al. created an application called VizWiz::LocateIt [13], which allowed blind users to locate objects using the camera on their smartphone. VizWiz::LocateIt used crowdsourcing to identify the object in the photo and computer vision techniques to provide audio feedback about the proximity to the object. TGV uses similar audio feedback to guide users to the QR code, but does not rely on crowdsourcing, thus providing quicker feedback.

Using computer vision techniques exclusively with a smartphone camera may enhance camera feedback. Jayant et al. created EasySnap [43], a camera application that assisted users in taking pictures by providing audio feedback. EasySnap used computer vision to locate people or objects in the viewfinder and relayed information on their location and size in proportion to the viewfinder. They followed with another application, PortraitFramer, which incorporated features of EasySnap and used haptic feedback communicate where in the viewfinder the people or objects were located. They found that it took little training for users to take better pictures. A similar

feature has been built into the camera application on recent versions of iOS [34]. When text-to-speech is enabled, the camera application provided feedback about faces, such as "face at top of screen," to guide users in taking portraits. TGV also incorporated audio feedback, but because users were using their sense of touch to explore a tactile graphic, I decided not to use haptic feedback to avoid cognitive overload.

TGV utilized audio feedback, but there are diverse options, such as tone and speech. Vasquez and Steinfeld [82] were interested in learning what type of audio feedback was preferred among blind people using a camera. They considered speech, tone, and no feedback. People strongly preferred speech feedback and found it easier to use than either silent or tone feedback. As a result, I use speech feedback in TGV as opposed to tone.

Most camera applications mentioned above were focused on taking a quality picture of a person or a physical object. Another related space is in technology that allow blind users to scan barcodes, which are similar to QR codes. The majority of commercial applications, such as the i.d. mate[8] or Digit-Eyes,[9] do not provide feedback. However, Tekin and Coughlin [80] experimented with different feedback modalities to help blind users scan barcodes on products. They used both verbal feedback and sonification, but their application was evaluated by a single user. TGV distinguishes itself in two ways: 1) QR codes are labels that can be located by touch and 2) multiple QR codes can be close together.

5.2.3 *Finger Pointing*

Because textbook images may have multiple text labels in close proximity of one another, the use of a finger may help select the preferred QR code when multiple codes are in the viewfinder. Thus,

---

[8] http://www.envisionamerica.com/products/idmate/

[9] http://www.digit-eyes.com/

I present related work on the practicality of finger pointing as a method to select a preferred QR code.

There are numerous projects that use finger pointing to identify an object or information of interest. One example is the EyeRing [58], a camera worn on the finger that read information aloud based on where the finger was pointing. Similarly, OrCam[10], also used finger pointing for people who are low vision. The OrCam was a wearable camera that used computer vision to identify objects towards which a user was pointing and read aloud information about that object. The manufacturers envisioned that OrCam could recognize faces, places, objects, and text.

Kane et al. developed Access Lens [48], a way for people who are blind or low vision to access documents. This system used a camera connected to a computer to read aloud the text on documents. Users could point to any element on the document to hear the associated information. This system brought promise to the accessibility of printed documents and demonstrated that finger pointing was an easy way for blind users to control what information they heard. However, Access Lens was not portable. In TGV, I capitalize on finger pointing as a simple means of selecting the information the user wants to hear.

Based on the results of the previous work, I believed that finger pointing was a good approach for selecting specific QR codes. However, the prior work used different camera devices where the orientation of the camera in relation to the finger was known. TGV used a mobile phone, which has not been used before. One difference in the mobile phone from the other devices used in prior work is that the code will not know the orientation of the phone in relation to the finger. To adjust for this, the software does not rely on the tip of the finger and can use any part of the finger to select a QR code.

---

[10] http://www.orcam.com

5.3 FORMATIVE STUDIES

To determine the feasibility of substituting QR codes for text labels on tactile graphics, I conducted a survey and follow-up interviews with people who are blind or low vision. I was motivated to learn about the current use of tactile graphics and cameras, and whether people would take interest in using QR codes as labels on tactile graphics.

The online survey was distributed to blind and low vision mailing lists and inquired about their use of Braille, tactile graphics, and camera applications on the smartphone. Twenty-two people completed the survey, where fifteen of the respondents were blind and seven were low vision. There were 12 female and 10 males with an average age of 38.18 (SD=13.46). All respondents had taken some college courses, and nine respondents had a graduate degree. Sixteen respondents knew Grade 2 Braille (contracted Braille), while only 3 respondents had little to no knowledge of Braille. All but one of the respondents owned a smartphone.

I conducted follow-up interviews with ten of the survey respondents, 6 of them female. I selected a diverse subset of those who indicated they would be willing to be interviewed on the survey. The interviews provided more detail about their survey responses and provided feedback about the proposed system, TGV. The participants' ages ranged from 21 to 67 with an average of 37.6 (SD=13.95). Five participants identified as blind and five as low vision. Five used Braille at work, three knew Braille but did not use it often, and two participants had little familiarity with Braille. Eight participants used tactile graphics in their education and work.

I found that many of the respondents frequently used cameras, especially on smartphones, and were interested in using tactile graphics with QR code labels. Seven of the ten participants reacted positively to replacing Braille with QR codes. One participant noted: *"You can fit a lot*

more information on a QR code than on a Braille label," a sentiment shared by five of the participants.

In addition, many of the survey respondents were familiar with using the camera on their smartphone, and thus have completed similar tasks to scanning QR codes. Fifteen respondents used an application that required the camera on a daily to weekly basis.

I learned that people found non-visual feedback for aiming the camera to be helpful. Just over half the respondents used an application that gave them feedback to help aim the camera. The majority of those respondents indicated that the feedback was helpful, with only one respondent mentioning that he had received feedback that was not helpful as it was unclear what it meant.

In the follow-up interviews, I investigated preferences for feedback modalities on a smartphone camera application: verbal, tonal, haptic, and no feedback. While the participants had a variety of preferences, I found that most participants preferred having the option of a quiet mode. Participants wanted a quiet mode because they felt that expert users needed less feedback, and they would not want to disturb others such as during a meeting.

5.4 Tactile Graphics with a Voice (TGV)

The first iteration of TGV was composed of tactile graphics with QR code labels and a smartphone application. The application provided multiple non-visual feedback modalities, and allowed the user to select which QR code they wanted to scan.

5.4.1 *Tactile Graphics with QR Codes*

The creation of tactile graphics for TGV required a similar amount of work as traditional tactile graphics. Traditionally, converting a textbook graphic into a tactile graphics is a labor-intensive process. First, the text must be removed from the graphic. In addition, some extra processing may

be needed to make the image understandable in a tactile form. Once the text was removed, it needed to be translated into Braille and be placed back on the image in similar location to the original text. The only change needed for the creation of the graphics for TGV would be instead of generating Braille, TGV generated a QR code from text using a free online generator. Because the embosser that was used to create the tactile graphics cannot print ink, I printed QR codes on a separate sheet of paper and glued them onto the graphic (See Figure 5.1 & Figure 5.4 for examples). It was not necessary to mark the QR codes with an embossed symbol because the height difference of the QR codes was sufficient to be felt. If the creator has an embosser capable of both embossing Braille and printing ink, the only difference from the traditional process is that the creator would place the QR code labels (with accompanying tactile markers) on the graphic in place of the Braille labels.

5.4.2 *Smartphone Application*

I created an accessible application for iOS that allowed a blind or low vision user to scan a QR code easily, even if there were multiple QR codes close together. The smartphone application was built on top of the ZXing software for scanning QR codes. This software identified QR codes by looking for an area of black and white variation. I made two modifications to the code: 1) providing a way for users to indicate which QR to scan when multiple are visible as otherwise the app will by pick one and 2) I added verbal feedback to help users scan QR codes.

Based on the survey and interviews, I integrated feedback for aiming the camera. In addition, I determined that it was important to have a feedback mode and a silent mode. Because the participants' preferences on feedback modalities varied, I used verbal feedback, based on prior work [82] and that most of the interview respondents indicated that verbal feedback was the easiest to learn.

I presented short clear verbal feedback to assist a user in moving the phone. I based the feedback on the screen location of the QR code, based on work by Vázquez and Steinfeld [82]. In their work, the verbal feedback implied a specific action ("up" or "down") of the camera that was not always the action that should be taken by participants. When they held the camera in a non-traditional orientation (e.g. facing down instead of forward), the participants were confused by the feedback as "up" would mean that they should move the camera forward. I expected that the participants may also hold the smartphone in non-traditional orientations (e.g. sideways), therefore I based the feedback on the QR codes location on the screen (using "top" and "bottom" instead). Then the user could translate this feedback into an action based on the orientation they were holding the phone.

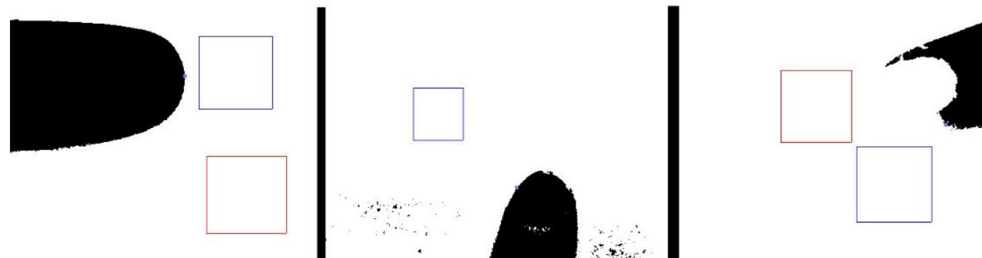


Figure 5.3. Above is an example of each category that the images can be placed into. The categories are: identified correct QR code (left), correct QR code was not found (middle), identified incorrect QR code when the correct was possible (right)

When multiple QR codes are visible, it was necessary to determine which QR code should be scanned. Therefore, I implemented finger pointing as a method to distinguish which label should be scanned. The selected QR code was the one with the shortest distance to the users' finger. To prevent the application from scanning the incorrect QR code, I set a maximum distance in which a finger could choose a QR code to scan. Unlike Kane, et al. [48], which selects the information at the tip of the finger, the application selected the QR code that was closest to any part of the finger. This was because unlike in the system in Kane et al.'s work, the camera does

not know which direction the tip will be, so it would require additional computation to determine what is the tip of the finger, decreasing the number of scans per second the code can do. As a result, users needed to be aware of their hand placement to ensure a false positive does not occur. I identified the finger with color based skin detection [29,63,64]. Because of the constrained black and white environment of tactile graphics, the app could identify a fingernail even if it is painted by looking for colored pixels and grouping them as part of the finger. Although the algorithm was generally successful, in testing it I have identified two main failure cases (Figure 5.3). First, if the correct QR code was not identified, the system would report back the nearest QR code as long as it was within a certain distance of the finger (Figure 5.3, middle). The other failure case occurred when the fingernail was not detected and the person pointed at angle to a QR code in a row of QR codes (see Figure 5.3 right).

5.4.3 *Feedback Modalities*

Because feedback and finger pointing are not appropriate in every situation, I created three modes for the application: Silent, Verbal, and Finger Pointing.

(1) *Silent mode* gave no feedback to help aim the camera. If multiple QR codes were visible in the viewfinder, the application did not scan. When it had successfully scanned a QR code, it chimed and then read the scan aloud.

(2) *Verbal mode* provided spoken feedback to help aim the camera. If multiple QR codes were visible in the viewfinder, the application spoke this information and did not scan. When the application had successfully scanned a QR code, it chimed and then read the scan aloud.

(3) *Finger Pointing mode* provided spoken feedback to help aim the camera. The application needed to detect the finger to scan. If the finger was not detected, the application spoke this information and did not scan. If multiple QR codes were visible, the application would

scan if the finger is detected. When the application had successfully scanned a QR code, it chimed and then read the scan aloud.

The different modes were likely beneficial in different situations. When a user is in a situation where they need to listen to other information, potentially a meeting or lecture, they may not want to have feedback spoken by the application as it may distract them. Additionally, when there are few QR codes on the graphics, the finger pointing mode may be less useful to indicate which QR code the user intends to scan.

5.5 INTEGRATING WITH THE TACTILE GRAPHICS ASSISTANT

The process described above for creating the tactile graphics is laborious. However, if the creators have access to the printers that can do both embossing and ink, I have done some work on integrating with the Tactile Graphics Assistant (TGA) [44], which sought to automate as much of the process of creating tactile graphics as possible. The current process to create tactile graphics using TGA worked to automate all five steps of the process: 1) cleaning up the image, 2) identifying and removing the text, 3) resizing the image 4) translating the text into Braille, and 5) replacing the text with Braille in the new tactile image. The new process replaces steps 4 and 5 with automated placement of QR codes.

5.5.1 *QR Code Placement Algorithm*

Jayant et al. found that most time consuming of the non-automated tasks was editing the placement of the Braille in the newly resized tactile images [44], therefore I am interested in using a heuristic algorithm to automate the process and reduce the amount of human editing needed. I use a greedy algorithm to place the QR codes as near as possible to the original text location

without overlapping each other or any pixels that will be embossed. I also compared output figures using various evaluation functions.

As input the placement algorithms took 1) a resized image from which the text had been removed, 2) a text file containing all the text that had been removed and 3) an XML file that had information about the text (including alignment and original placement) and both the x and y scale factors for the resizing of the image. For the study, the algorithms output images with black boxes in place of the actual QR codes, but it would not take much to change the boxes into the actual QR codes. I ensured that the black boxes were the correct size for the amount of text in the QR code and had a buffer of forty pixels to give a white border around the QR code, so it could be read and would not blend in with the embossing. Because I am using these boxes instead of QR codes, I will refer to them as QR labels for the rest of this section.

The initial placement of the QR labels in the resized images was calculated by multiplying the x and y coordinates of the original image by the x and y scale factors. As the dimensions of the QR labels were significantly different than those of the original text, which x and y coordinates were used depended on the alignment of the original text: if the text was left-aligned, the initial placement used the top left corner, if it was right-aligned, the top right corner was used and if it was centered, the top center was used. If any part of the QR label was initially placed off the image, it was moved until its edge was along the outside of the image. This initial placement was used as the starting point for all the algorithms.

For the greedy algorithm, the QR labels were placed in a priority queue ranked by their evaluation score, highest score first. The algorithm removed the worst-placed QR label (the one which had the highest evaluation score) from the queue and evaluated moving it in eight directions (up, down, left, right and along the diagonals) by ten pixels. Along the diagonals, the label was

moved 10 pixels along both the x and y axes. The algorithm picked the direction that improved the evaluation score of that QR label by the most. If movement only worsened the evaluation score, the QR label was put aside and rejoined the priority queue only after another QR label was moved. To prevent thrashing, I also limited the number of times a QR label could be moved to 50. While a greedy algorithm had the possibility to get stuck in local optimum, it provided an initial demonstration of how much even a basic algorithm can reduce the amount of time spent placing the labels.

I explored several functions to evaluate the placement of a QR label. The functions were all of the form: $score = \sum w_i \ f_i$, where the w_i's are weights and the f_i 's are numeric features that I identified as possibly being important in the evaluation function. The features included in the evaluation functions were: (1) distance from original placement (along both the x and y axes), (2) overlap over other QR labels, and (3) overlap over the image. In determining the image overlap, there were two related features: one computed the number of non-background image pixels that were overlapped, and the other computed the number of non-background image pixels weighted by the section of the image that QR label overlapped. Pixels near the center of the QR label had a higher weight than those on the outside. An example case where this would help is where the center of a QR label is overlapping the image. Moving the QR label 10 pixels may provide no improvement, but moving the QR label 20 pixels may improve the overall placement.

I tested the placement algorithms using a number of different weights for the features on images acquired from a Precalculus textbook [35] previously used for the TGA. The images had already been digitized, and classified into different categories. I wanted to make sure I tested the algorithms on a variety of types of images to make sure that I was not over-fitting the evaluation functions to one type of image. Therefore, I selected the images from two data sets, complex

images which had a wide variety of images that were considered harder to place the QR labels on and clean lines which contained mostly graphs, a very common image type in the textbook. To get a representative sample, I selected every third image from these two sets. After using these two data sets, I had 27 figures to use in the evaluation.

5.5.2 *Feasibility Evaluation*

To determine whether the above method decreases the amount of manual labor for a potential user, I had a tactile graphics expert who did not participate in the creation of the images with QR labels evaluate the label placement. I used their feedback to assess the effectiveness of three different evaluation functions (sets of weightings). The different weightings considered different choices of what features to consider more important than others.

For each of the 27 figures, I created a task sheet for the expert to reference. I placed the original image from the textbook at the top of the page. This image included the English text so the expert could tell what text each QR label contained. Below the original image were the four images containing different QR label placements, one for each evaluation function I tried as well as one for the initial placement of the QR labels. The four images were placed in a randomized order and given labels A, B, C, and D so the expert was unaware which image went with which placement type to prevent bias.

To determine the amount of saved human labor, I used two different metrics. The first metric I looked at was what percentage of images required fewer QR labels to be moved than the initial placement. Ideally, I would like to have every image require fewer QR labels to be moved or at least none to require more labels to be moved. I found that based on the expert evaluation, all the images had the same or fewer number of QR labels that needed to be moved and 77.7% of the time the images required fewer QR labels to be moved.

Once I knew that most the images did not require more QR label moves, I wanted to see what the effect was on the total amount of work that needed to be done. Therefore, I used the metric of average number of QR labels that need to be moved per image. I found that the total amount of work also decreased when compared to the initial placement. When considering all the images, the number of QR labels that needed to be moved decreased slightly, but the decrease was much more dramatic when looking just at the images deemed solvable. Some images were deemed unsolvable as with the current constraints of the image size, it was not possible to fit all the QR labels. This was most common with images that had axes with large number of labels along the side. Either the image would need to be larger (or have more buffer around it so that the labels can be staggered) or some QR codes removed. If I look at only the solvable images using the best of the three evaluation functions, the number of QR labels that need to be moved went from 1.9 in the initial placement to .3 per image in the placement provided by this algorithm.

This study of QR code label placement algorithms demonstrates that some human time in placing QR codes can be reduced by relatively simple heuristic algorithms. I explored three possible evaluation functions with a greedy algorithm, but better algorithms are likely possible.

5.6 Longitudinal Study

To evaluate the efficacy of TGV, I conducted a six-session longitudinal study with ten blind and low vision participants. Participants answered questions using TGV with the three modes of feedback (Silent, Verbal and Finger Pointing). In the last session, I had participants who knew Braille complete the same tasks using tactile graphics with Braille labels.

5.6.1 *Participants*

I conducted the study with ten participants (four male, six female), with ages ranging from 30 to 54 years, and an average age of 41.9 (*SD* = 8.1). Five had college degrees, three had some college education, and two had a high school education. Four participants identified as low vision and the remaining six identified as blind. Six participants completed the Braille portion of the study, while four were not Braille literate or were not confident in their Braille skills. Overall, participants did not have much experience with tactile graphics, with five never using them, three rarely using them, one using them once per month, and one using them once per week. Nine participants had smartphones; seven had iPhones and two had Androids. The smartphone users had used camera applications for varying frequencies: two used them daily, two weekly, two monthly, and three rarely used their smartphone cameras. Finally, eight participants had no experience scanning QR codes with their smartphones and two had some experience.

5.6.2 *Apparatus*

The TGV application ran on an iPod Touch 4th generation and an iPhone 5, each running iOS 6. Each participant used the same device for all six sessions. The tactile graphics were printed on standard 11x11.5 inch Braille paper and embossed with a Tiger embosser[11]. QR codes were printed on standard printer paper and cut and pasted onto the tactile graphics in the appropriate places. Braille labels were embossed directly on the graphic using Nemeth code, the type of Braille usually found in math textbooks. Numbers in Nemeth code and Grade 1 and 2 Braille are similar; in Nemeth code the dots are shifted down a row [61].

[11] http://www.viewplus.com/

5.6.3 *Procedure*

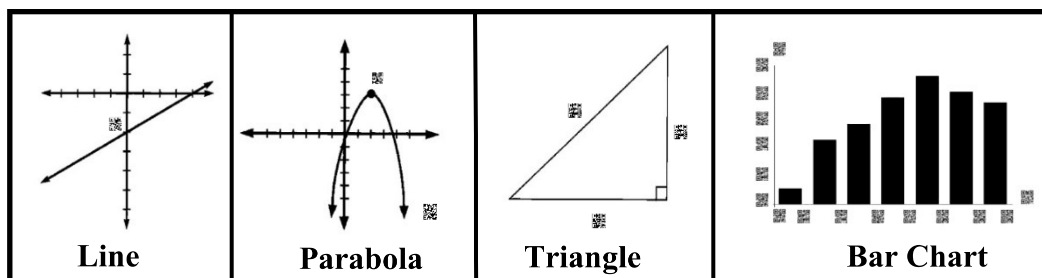


Figure 5.4. This is an example of each of the tasks that the participants completed. In each session, participants used similar graphics, but with different labels (i.e. the parabola might be the opposite direction and have a different vertex).

I had each participant complete six sessions over a two week period. I wanted participants to interact with the application over time to emulate a real-world situation, such as using the application to complete schoolwork.

During the first session, I collected demographic information from the participants, and taught participants how to use the three modes of the TGV application (Silent, Verbal and Finger Pointing). I explained how each mode worked and provided basic information about using the application, such as the suggested scanning height and where the camera was physically located on the device. Participants had a chance to practice scanning a QR code with each mode.

During each session, participants completed a total of twelve tasks by using each mode of TGV (Silent, Verbal and Finger Pointing) on the following four tasks (Figure 5.4):

(1) Line. The first task was to find the y-intercept on a line graph. The graphics always had one QR code representing the value of the intercept.

(2) Parabola. The next task was to find the (x.y)-coordinates of a parabola vertex. The graphics used in this task always had two QR codes, one for the coordinates of the vertex and one for the equation of the parabola.

(3) *Triangle.* The third task was to find the length of the hypotenuse of a right triangle. The graphics in this task always had three QR codes, as the lengths of all sides were labeled.

(4) *Bar Chart.* The final task was to find the left and right values on the x-axis of the tallest bar in a bar chart. For this task, the bar chart had seven bars, and there was a QR code marking the bounds of each bar on the x-axis and each tick mark on the y-axis as well as axes labels.

The images for each task were based on images taken from a precalculus textbook [35]. At the beginning of each task, a tactile graphic was placed in front of the participant, and they were instructed to begin. The task ended when the participant responded with their answer. I recorded the task completion time and their answer. I video recorded participants to validate this data. For the first three tasks (Line, Parabola, and Triangle), participants can only receive 0 or 100% accuracy. On the final task (Bar Chart) task, participants can also receive 50% accuracy, as that task required finding both the left and right values of a range. Participants were not told whether their answers were correct to mimic a testing situation. I randomized the order of modes used in each session but kept the order of tasks consistent: Line, Parabola, Triangle, and Bar Chart. At the end of each session, I conducted a survey to gauge the participants' preferences for the feedback modes. Participants were asked to rank the modes based on their preferences and then rate the following semantically anchored scales.

1. Please rate how much you liked using the different types of feedback: (1-Strongly like to 7-Strongly dislike)
2. Please rate how helpful you thought the different types of feedback were. (1 – Very helpful to 7 – Very unhelpful)

3. Please rate how easy to use you thought the different types of feedback were (1 – Very easy to use to 7 – Very hard to use)
4. Please rate how easy to understand you thought the verbal feedback was: (1 – Very easy to understand to 7 – Very hard to understand)

In the last session, participants who were proficient in Braille attempted to complete the same tasks using Braille labels in lieu of the QR codes. I choose to have the comparison to Braille only in the last session for two reasons. The first is that participants were already familiar with Braille so I felt that they did not need the time to learn it. By completing the sessions with TGV, they would be familiar with the tasks by the sixth session, making the comparison from TGV to Braille more equal. The second is I wanted to limit the length of the sessions to prevent fatigue. As some of the participants were Braille-literate, but not familiar with Nemeth code, I explained the difference between Nemeth and Braille to those participants.

5.6.4 *Design and Analysis*

The study was a 6×3 within-subjects design with factors for *Session* and *Mode.* The levels of *Session* were (1-6); the levels for *Mode* were (Silent, Verbal, Finger Pointing). Each participant completed a total of 72 trials, for a total of 720 trials with the smartphone, and six participants additionally performed 4 trials with Braille at the end of the session. The other four participants were not comfortable enough with Braille to attempt those tasks. I measured completion time and accuracy for each task. If participants took longer than 180 seconds to complete a task, I stopped them and recorded that they had timed-out on the task. Participants were still allowed to submit an answer if they timed-out on the task.

While analyzing completion time for a task, I used a mixed-effects model analysis of variance with fixed effects of *Session* and *Mode*, with *Participant* modeled as a random effect. For accuracy and preference data, I looked at the descriptive statistics.

5.7 RESULTS

5.7.1 *Accuracy*

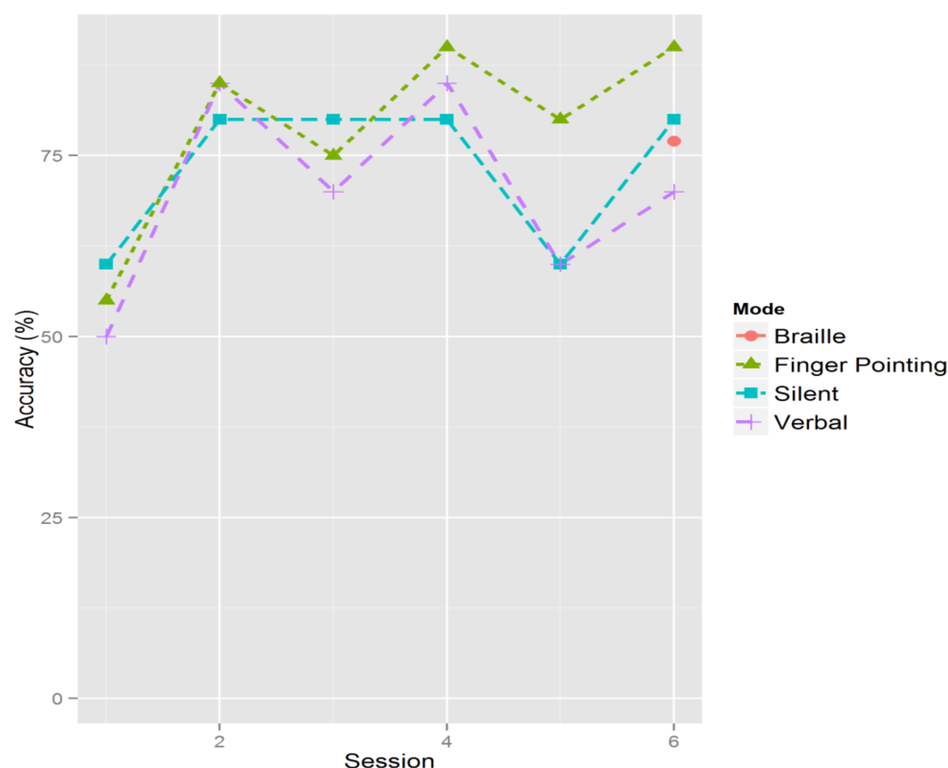


Figure 5.5. A comparison of the average accuracy for the Bar Chart task across the six sessions for the three modes (n=10) and Braille (n=6) on the last session. Participants were asked to find the range of the tallest bar and their answer could be 0, 50 or 100% correct.

The accuracy for each task did not vary across the different modes (Silent mode: 88%, Verbal mode: 88%, Finger Pointing mode: 89%). While there was no significant difference in the accuracy between the first and last session, I found that accuracy tended improve in the last session (Figure

5.5). In addition, I saw that the accuracy tended to be lower on the bar chart task (Table 5.1). I hypothesize this was the case because this task had the most QR codes closest together.

Table 5.1. Overall average accuracy of participants on the tasks with each mode

	Silent	Verbal	Finger Pointing
Line	97%	97%	93%
Parabola	93%	95%	95%
Triangle	88%	88%	90%
Bar Chart	73%	70%	79%
All Tasks	88%	88%	89%

5.7.2 *Time*

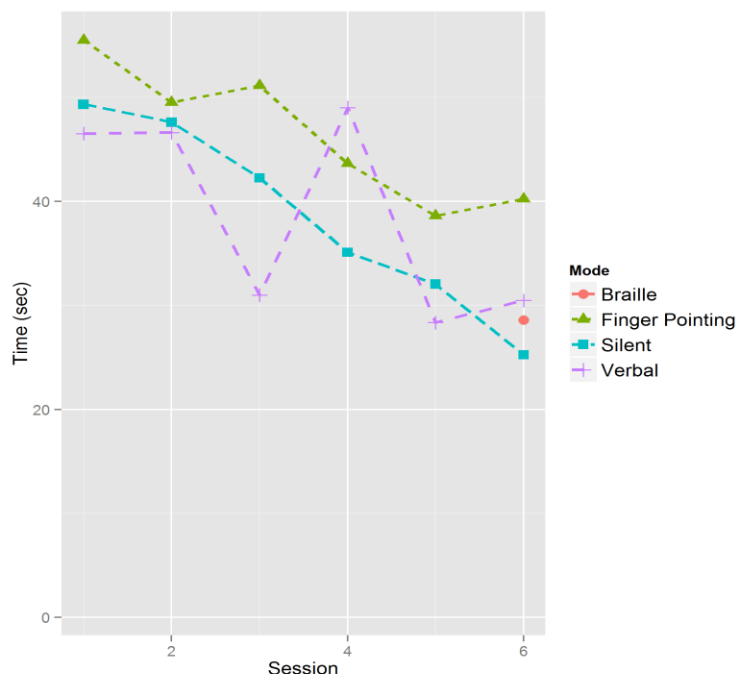


Figure 5.6. A comparison of the average time for each participant to give the answer for a task for the three modes (n=10) across the six sessions as well as for Braille (n=6) on the final session.

If participants reached 180 seconds without answering the question and completing the task, this was counted as a time-out, and the time is not included in the average or the statistical analysis. Out of 720 tasks, the total number of tasks that timed out was 41, or 5.7%, and almost

half of those time-outs (19) occurred in the first session. Additionally, over half of the time-outs (21) occurred during the difficult Bar Chart task. The time-outs occurred in all the modes, with 16 time-outs occurring in the Finger Pointing mode, 16 occurring in the Silent mode, and 9 occurring in the Verbal mode.

With time-outs removed, the average QR code task completion time for all sessions and all modes was 40.9 seconds (SD =36.3). However, the participants were faster in the sixth session than the first (see Figure 5.6) and the effect of *Session* on time was statistically significant ($F_{5, 640}$=2.268, p<.05). In session six, the average Silent mode completion time was 25.3 seconds (SD=18.7), Verbal mode completion time was 30.5 seconds (SD=29.8) and Finger Pointing mode completion time was 40.3 seconds (SD=29.4). The effect of *Mode* on time was not statistically significant ($F_{2, 640}$=0.619, p=.5391), though it is observed that Finger Pointing mode took more time than the Verbal and Silent modes.

5.7.3 *Feedback Modality*

At the end of each session, I asked each participant to indicate their feedback modality preference by ranking the different modes. In addition, each participant rated to rated how much they liked using each mode, how helpful they found each mode, and how easy to use they found each mode on a 7-point semantically anchored scale. The exact questions can be found in Section 5.6.3. For each scale, a 1 was the best and 7 was the worst. Like the survey and interview, I found a wide range of preferences.

When looking at the participants' ratings of how well they liked the modes, I found Verbal mode received an average rating of 2.87 (SD=1.57), Finger Pointing mode received an average rating of 3.63 (SD=1.72) and Silent mode received an average rating of 3.98 (SD=2.11). For the rating regarding helpfulness, Verbal mode received an average rating of 2.85 (SD=1.38), Finger

Pointing mode received an average rating of 3.33 (SD=1.71) and Silent mode received an average rating of 4.13 (SD=2.05). For the rating regarding how easy to use the modes were, Verbal mode received an average rating of 2.68 (SD=1.26), Finger Pointing mode received an average rating of 3.33 (SD=1.89) and Silent mode received an average rating of 3.53 (SD=2.16).

However, the ranking of each method varied strongly between participants and over time. In the final session of the study, four out of ten participants ranked Silent mode as their favorite mode, four ranked the Finger Pointing mode as their favorite and two ranked the Verbal mode as their favorite. Interestingly, many of participants had their least favorite mode in the first session be their favorite mode by the last session or vice versa. Half of the participants had this change in preference between the first and last session, with three participants selecting Silent mode as their least favorite mode in the first session and favorite mode in the last session.

Participants that preferred the Finger Pointing mode generally thought it was more accurate. Participant 1 stated:

I like the concept of the finger pointing. I feel more confident that since it looks for a finger it's getting the right QR code if you have multiple on the same page.

People who did not like the Finger Pointing mode thought it was difficult to use. If users put their finger too close to the QR code, it would not recognize the QR code. There needs to be a small gap between the finger and the QR code for both the finger and QR code to be recognized. This caused difficulties because participants did not realize that their fingers were obscuring the QR code. P2 expressed frustration:

With the pointing with the finger it kept not registering cause either my finger wasn't in the right spot or it kept picking up the wrong one somehow.

Participant 3 stated that "I haven't been able to see my finger point in years, so knowing where my finger is isn't useful," but thought that it might be useful for others:

> *I did like that you're - that it's trying to branch out and give people options for identifying things like with a finger. It's a pretty neat touch. I like that. I could see that turning into something useful. I think my preference was still for just taking it with a simple picture with the camera*

Participants that preferred the Silent mode were fatigued of audio feedback, as participant 3 said, "To be honest, I use screen readers every day and I am so sick of electronic noise." Participants disliked Silent mode because they felt that they needed feedback to know what was happening in the application. In the words of Participant 1: "the lack of feedback makes it harder to use because you don't know whether it sees a QR code," and Participant 9: "I still prefer having more versus less feedback."

Participants that preferred the Verbal mode liked it because it provided feedback, but was less of a cognitive load than Finger Pointing. In the words of Participant 9: "the other thing I like about Verbal mode is that every time I hear a zero I think so I need to move it a little bit," as opposed to the Finger Pointing mode, which:

> *Presents more issues to deal with you already have to deal with how many labels are here and then I got this finger issue this finger needs to be there, but it can't be too close [and] it can't be too far away.*

5.8 COMPARISON TO BRAILLE

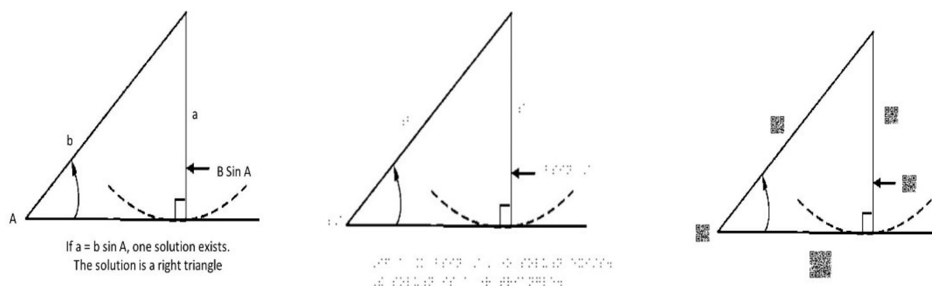


Figure 5.7. A comparison of the same image which is similar to one from a pre-calculus textbook [35] in its original form, tactile graphic form with the labels in Braille and tactile graphic form with labels as QR codes. The bottom text is a good example where the QR codes can be smaller than the equivalent Braille.

While the system was designed primarily for blind users who are unable to read Braille, there are benefits for people who are Braille-literate as well.

5.8.1 *Difficulties in Creating Tactile Graphics with Braille Labels*

To assess the difficulties in producing tactile graphics, I spoke with three tactile graphics experts. All three had extensive experience in creating tactile graphics and had encountered a variety of problems with the creation of tactile graphics.

From the expert interviews, one common problem was how to place Braille labels on tactile graphics. Because of the limits of human tactile perception, Braille cannot be resized to fit into a small area [73]. This means that labels with a large amount of text must be moved. One technique for mitigating this problem is to create a key and legend. A short code is placed on the graphic where the label should be and the corresponding label is placed on a separate page. One of the experts estimated that the key and legend system is necessary for a quarter to a third of all the images he produces. Another tactile graphics expert mentioned that three quarters of tactile

graphics require an explanation for them to be understood, and the explanation would not fit on the original graphic, requiring a second page.

5.8.2 *Size of Braille vs QR Codes*

I did a size comparison between Braille and QR code labels and found that the QR codes can encode 45% more text in the same amount (Figure 5.7). This calculation was completed by looking at 82 images from a pre-calculus textbook [35]. I calculated the estimated size of the Braille label using: the product of the number of characters in the text and the size of a Braille cell, which is the standard size of all Braille characters [73]. While many math symbols require multiple Braille characters, my conversion from text to Braille provides a good approximation. Unlike Braille, which has a standard size, QR codes vary in size based on the amount of text they encode and the distance from which they are meant to be scanned. By assuming a scan distance of six inches, I calculated the size of a QR code label based solely on the number of characters it encoded [67]. I found that the average QR code label size is 225 mm^2 and the average Braille label size is 327 mm^2.

5.8.3 *Study*

The main goal of the study was to determine if the TGV system was a feasible solution to making labels accessible to those who did not know Braille, and I feel the study demonstrates this fact. Below, I will explain the results from the comparison to Braille in the last session of the longitudinal study that was done with the 6 participants that did know Braille.

5.8.3.1 *Accuracy:*

Across all participants, the average accuracy for TGV using any mode was higher than the average accuracy using Braille (Silent mode: 88%, Verbal mode: 88%, Finger Pointing mode: 89%,

Braille: 77%). For the bar chart task, the TGV accuracies are similar to the average accuracy for Braille (Figure 5.5). While this finding goes against my hypothesis, this finding is likely due to two reasons. First, the Braille tasks the labels were written in Nemeth code. Even though I explained the how to read Nemeth-coded numbers, some participants made mistakes. Second, some of the Braille-literate participants indicated that they were out of practice reading Braille.

5.8.3.2 *Time:*

The average completion time with the Braille graphics was 28.6 seconds (SD=19.0). This was faster than the average time of TGV with all the different modes. However, after the participants learned to use the application, the times were similar. This can be seen in Figure 5.6, where the dot representing the Braille mode was faster than Verbal and Finger Pointing modes but slower than Silent mode in session 6.

5.8.3.3 *Preference:*

Four of the six participants who used the Braille labels on the graphics stated that was their favorite. One reason was because of ease of use, as P6 stated that (with Braille): "it's already there and you can just read it." Additionally, people were more comfortable with Braille and thought it was more accurate. In the words of P4: "I'm very comfortable with Braille. It feels more reliable."

The other two participants who preferred TGV to Braille, did not feel comfortable with their Braille literacy skills. In the words of P1:

> *I guess if you're reading a textbook in Braille you're probably up on your Braille so you wouldn't need a smartphone or anything to access that,*

and P5 said that:

I wish I had learned Braille when I was in school because that might that may have made a world of difference and I would be a lot more successful than I am right now so I do I really enjoy the Braille a lot.

5.9 DISCUSSION

Errors on the first three tasks (Line, Parabola, and Triangle) were a result of misidentifying which label to scan or timing out. In contrast, with the Bar Chart, errors occurred when a participant attempted to scan the correct QR code, but really scanned a different QR code. This issue occurred because of the small distances between the labels on the axes. Participants developed strategies to avoid this problem in later sessions, such as covering the neighboring QR codes. This technique helped increase the accuracy for all three modes from 55% (SD=35) for the first session to 80% (SD=30) for the last session. Figure 5.5 displays the changes in accuracy across the sessions by mode.

Although Finger Pointing mode was the most accurate, many participants had difficulty using the mode. There were two levels to the feedback as the application would first check for the finger and then look for a QR code if it had found. Therefore, if a user had both parts missing, it would take two steps to provide the feedback. Additionally, users had to determine if a QR code was not visible because they were covering part of the code causing it not to be recognized or if it truly was not visible to the camera. I believe this is the main reason that Finger Pointing took significantly longer than the other modes and lead to more time-outs.

5.10 SUMMARY

I have presented Tactile Graphics with a Voice, a method to access the text labels in tactile graphics, using a smartphone application that provided feedback to help blind users scan QR codes.

To the best of my knowledge, this system is the only solution for people who are blind or low vision and cannot read Braille to access the text on an image that does not require the use of a specialized device or access to proprietary software. I discuss the development of the system and algorithmic details of how I created graphics and enabled finger pointing. I conducted a longitudinal study and found that even for people who can read Braille, TGV was comparable in terms of time and accuracy to Braille labels. Ensuring that blind people can quickly and accurately access the text labels on tactile graphics is an important part of making educational materials accessible to all.

Chapter 6. STRUCTJUMPER

In this chapter, I seek to answer *RQ3: How can we make it easier for blind programmers to contextualize their location and navigate through code?* As I know from prior work that navigation and information look-up is a challenge for screen reader users, I wanted to design a system that would provide access to some of the same cues that sighted programmers are able to use when navigating through the code. To do this, I created an Eclipse plug-in which creates a hierarchal tree based on the nesting structure of the code. The tree can be used to update the cursor location or just look up contextual information. To evaluate StructJumper, I had 7 blind programmers complete tasks both with and without StructJumper. I found that there was a trend that users were faster with StructJumper. Additionally, they found it quicker and easier to navigate and when participants needed to look up contextual information, they found it quicker and they did not have to hold as much information in memory. [12]

[12] The work in this chapter was done in collaboration with Lauren Milne and Richard Ladner and is based on work previously published at CHI 2015 [7].

6.1 INTRODUCTION

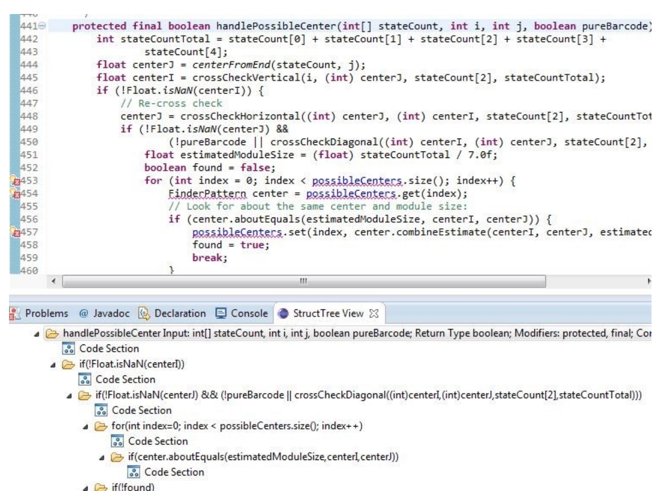


Figure 6.1. Screenshot of StructJumper with source code file on top and tree of nesting structure on bottom.

Computer programmers rely on the use of visual aids when programming [78], especially in an integrated development environment (IDE) such as Eclipse. These visual aids range from using different colors for syntax highlighting to using indentation within the code to indicate scope. The use of visual aids present difficulties for blind programmers, as they are unable to quickly access the same information available to sighted developers. In fact, blind developers have more difficulties navigating and understanding the structure of code than their sighted counterparts [57,74,78]. Screen readers only allow blind programmers to have access to a single line of code at a time. Therefore, to move around in the code, the programmers are limited to a few options: using the arrow keys to go through each line of code, using the outline or package explorer to navigate to a specific method and then navigating within the method line by line, or using a search mechanism. Despite these difficulties, the space of accessible developer tools and studying the practices of blind programmers is still a relatively unexplored field.

Smith et al. created a tool to allow blind programmers to navigate the hierarchical structure of a program, specifically the tree structure of files in the Eclipse IDE [74]. My tool, StructJumper,

expanded on this work by creating a hierarchical tree of the nesting structure of a program (see Figure 6.1, Figure 6.3, and Figure 6.2) to allow users to both navigate within the program and gain an understanding of the structure of code within the program. I created one tree per Java file, and the root of each tree was an invisible node corresponding to the file. A node was a child to another node if the code of the child node is nested within the code of the parent node. Inner nodes represented classes, methods or statements, and leaf nodes were code sections without any changes in nesting. I included these code sections in the tree, because I wanted to allow users to easily switch between coding and finding where they were in the tree structure, so every line of code must be contained within a node on the tree.

StructJumper allowed the user to quickly discover in which nested structure a particular line of code she was working on. She could do so by pressing a key in the tree view to jump to the node corresponding to the current location. Moreover, it allowed the user to switch between being able to make edits within the code and gaining contextual information without losing her place. I present a prototype of the plug-in for Eclipse for a single programming language (Java) and evaluate the tool in a user study with seven blind developers. I had participants perform a variety of code exploration tasks both with and without the tool.

With my evaluation, I aim to answer the following research questions:

(1) Does StructJumper make it easier for a blind programmer to navigate the code?

(2) Does StructJumper make it easier for a blind programmer to understand where they are within the code?

To evaluate StructJumper, I had seven blind programmers complete three tasks related to navigation and answer questions about the context of a line of code. I found that the users in the

study thought the tool was useful for navigation and for understanding the structure of code and there was a trend that they were faster at completing the tasks with StructJumper.

My contributions are:

(1) The StructJumper tool itself, available as a plug-in to Eclipse.

(2) The results from the evaluation of the tool, which show that StructJumper was useful in helping blind developers navigate code and gain an understanding of which statements a line of code is nested within.

(3) Insights into how designers should create similar navigational tools for blind programmers.

6.2 RELATED WORK

Beyond the related work described in Chapter 2, this work also pulls heavily from work looking at the navigation of web pages and text documents. In early work by Asakawa and Itoh [5] developing an add-on to a screen reader that could read web pages, the researchers found that navigation was important to the design of the screen reader. Unlike navigation in IDEs, many controls allowed users to skip between links or lines on a page or skip directly to the first or last link.

With current standard screen readers, users can use controls to switch between header types (e.g. h2 and h3) and then skip from header to header on web pages. In a 2012 survey by WebAIM [85], 61% of 1782 respondents reported using headers as the main form of navigation when trying to find information on a lengthy web page, as opposed to using the find feature, navigating using links, landmarks or simply reading the page. Additionally, 82% of respondents found having different heading levels either useful or very useful when navigating a web page. However, to allow users to navigate using this structural information, webpage creators must provide it. The

Web Content Accessibility Guidelines 2.0 [23] provides guidelines to help developers create accessible pages. The guidelines require that "Information, structure, and relationships conveyed through presentation can be programmatically determined or are available in text," and that "Headings and labels describe topic or purpose" [23]. Headers are particularly important for navigation and understanding content on a webpage.

The plug-in worked similarly by designating certain parts of the code as key structural parts of the code (e.g. method and class declarations, control flow lines, etc.), which served similar purposes as the headers on a website. Although there are tools available that can move between nesting levels in IDEs today, to the best of my knowledge no one has studied whether they are useful for navigation by blind developers, and they do not provide the ability to switch between navigation while still maintaining the current position of the cursor, which is useful for blind developers.

6.3 StructJumper Design and Implementation

```
public class Calculator {
	private String display;
	public int add(int a, int b) {
		return a+b;
	}
	/**This method subtracts b from a*/
	public int subtract(int a, int b)  {
		return a-b;
	}
	public double exponent(int a, int b) {
		double answer = 1.0;
		if(b>=0) {
			for(int i = 0; i<b; i++) {
				answer = answer*a;
			}
		}
		else{
			for(int i = 0; i<b; i++) {
				answer = answer/a;
			}
		}
		return answer;
	}
}
```

Figure 6.2. Code for a simple calculator class, which is turned into the tree in Figure 6.3Figure 6.3.

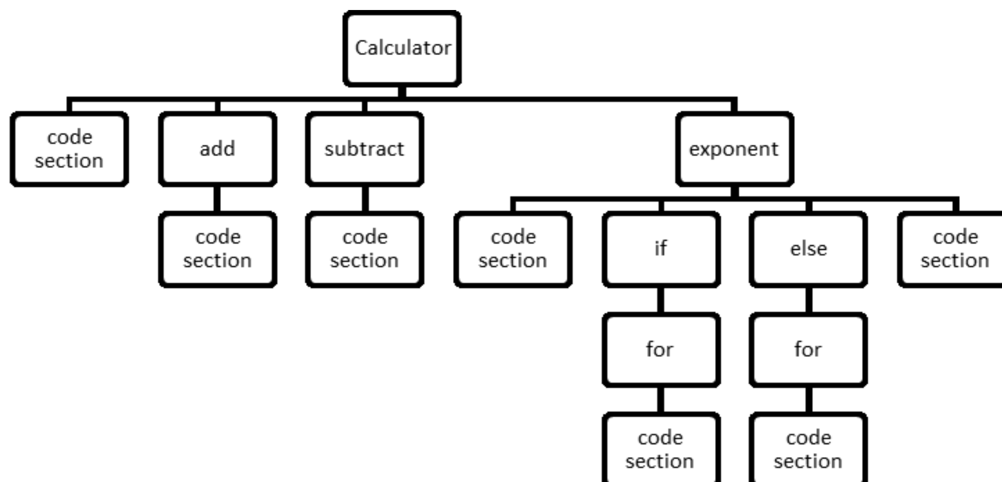


Figure 6.3. The tree created from the code in Figure 6.2. Code sections have no further nesting. Note in this image, the first code section corresponds to the code containing the member variable declared at the beginning of the Calculator class.

I created StructJumper, a plug-in for Eclipse. I chose to use Eclipse for a couple of reasons. Eclipse was a mainstream IDE that is used commonly by both blind and sighted programmers. As it is common for programmers to work in groups, having a common IDE was beneficial. Additionally, Eclipse was open source and had good support for creating and adding plug-ins.

For the plug-in, I was combining two concepts that had already been used in software development. This first concept was turning code into a tree structure. This has been done with Abstract Syntax Trees (AST). I was using a simplified version of an AST, as I did not want to overwhelm the programmer with too much information. Grouping code together at a certain nesting level was not a new concept and was frequently done by visual cues (e.g. indentation or highlighting). I was just adjusting this method to be in a format accessible to blind programmers.

The plug-in created a hierarchical tree of the code based on nested structure (Figure 6.2 and Figure 6.3). I broke nodes into two categories: code sections and statements that precipitate a change in nesting. Code sections were sequential lines of code that are all within the same level of

nesting. They could only be a leaf on a tree, not a parent to other nodes. A procedure call was not included as a separate node in the tree, but within the containing code section.

The tree was created in a separate window so that a programmer can use StructJumper both to navigate as well as to gain contextual information. This was further enabled by the key commands in Table 6.1. One of the major decisions I made was what should happen when a user entered and exited StructJumper.

Table 6.1 A table of the keyboard shortcuts that can be used to navigate in the tree and the code editor

Key	Action
Ctrl+F7	Switches between tree and editor and leaves the cursor/selected node at previous location (Eclipse Built-In Command)
Left Arrow	Go to parent
Right Arrow	Go to first child
Up Arrow	Go to previous sibling
Down Arrow	Go to next sibling
C	Go to node representing cursor location
T	Go to top of the tree
U	Update the tree
E	Switches to the editor and updates the cursor location to that of the current node

On entering, there were two options, (1) update the selected node to the one that represents the cursor location or (2) to leave it on the previously selected node. There are good arguments for either, but based on limitations of the screen reader, I decided to leave the cursor on the previously selected node (option 2).

On exiting there were also two options. (1) update the cursor location to that of the selected node or (2) leave it at its previous location. Work by Mealin and Murphy-Hill [57] showed that

being able to get information (such as a method name) while leaving the cursor in the current location was valuable to programmers and that many blind programmers created work-arounds to

Figure 6.4. This is an example of a portion of what the Package Explorer in Eclipse would show.

gain this functionality by using a text buffer. Therefore, I allowed for both options to be possible with 'E' updating the cursor location and 'Ctrl+F7' leaving the cursor in the previous location.

In Eclipse, the Package Explorer window had a similar layout, but only included class, fields and methods (Figure 6.4). As I wanted to make sure that the tool used the same mental model as all the other parts of Eclipse, I used that model for the tool as opposed to one similar to Figure 6.3. While this was inspiration for the layout and key selection choice, I made one change in how the arrow keys work. In the Package Explorer, if a user pressed the down arrow key when the selection was on "TreeParent" (Figure 6.4) then the selection moved to "children." To avoid moving the selection onto a further nested statement without the programmer being aware, the programmer must use the right arrow to move to a child. If the user pressed the down arrow, the selection would move from "TreeParent" to the next item at the same nesting level, "ViewContentProvider."

When the user was in navigation mode, the screen reader read relevant and unique cues (such as method names) first and then provided the rest of the information, so that users could quickly navigate and skim through code as suggested by Stefik et al. [79]. To present the most important information first, I reordered the presentation of several lines of code that are in the tree.

For a method, I first present the name, followed by the input, return type, and then any modifiers or annotations. For a class declaration, I first present that it is a class, followed by the name, then what it extended and implemented, followed by the modifiers. When necessary, keywords were inserted to distinguish the end of one type of item and the start of the other. For instance, the keyword "return type" would be added in between the input and return type to more easily distinguish the difference. Other lines of code were read as is, as they already have the important keywords first. For example, the subtract method in Figure 6.2 would be read "*subtract, Input: int a, int b, Return type: int Modifiers: public Comments: /**This method subtracts b from a*/.*" The plug-in was written in Java, and currently only parses Java code.

6.4 Experiment Design

To evaluate StructJumper, seven blind programmers completed three tasks, while using StructJumper and without the tool. After they had completed the tasks, I asked them questions about their experience.

6.4.1 *Participants*

I conducted the study with seven blind programmers, one of whom was female. I sent recruitment emails to an international mailing list of blind programmers and included a request to forward it to anyone that they thought would be interested. I also shared it with contacts that may be interested in the study or knew people that may be interested. The average age of the participants was 24.1

(SD = 4.9). The programmers reported an average of 7.8 years of programming experience (SD = 3.9), with a minimum experience of 3.5 years. Additionally, the participants reported an average of 2.8 years of experience with Eclipse (SD = 2.6) and 3.8 years of experience with Java (SD = 2.8). The minimum experience for Java and Eclipse was 0.5 years.

6.4.2 *Set-Up*

The study was conducted remotely, allowing each participant to use their own computer set-up. The participants used a variety of screen readers including JAWS, NVDA, and Window Eyes. By conducting the study remotely, the participants could use the settings with which they are comfortable, such as talking speed or amount of punctuation to speak. Using the screen sharing abilities of Skype and Google Hangouts, I could watch and record as the participants completed the tasks and track their progress.

6.4.3 *Procedure*

Before the study, the participants were asked to fill out demographic information and install StructJumper. Participants were not given access to the code until minutes before I started the study.

The study was divided into three parts, completing a series of tasks with the tool, completing a series of tasks without the tool and the post-session interview. There were two different code bases and each code base had three tasks that were similar to each other. The participants completed the tasks on one code base using the tool and one code base without the tool. The order of the code bases and whether they used the tool first or second was counterbalanced.

As common advice to improve programming skills is to read other people's code, I selected two trending repositories from GitHub. The code bases were selected as they each had a long file (600-800 lines of code), which was well commented, on which navigation would not be a trivial task. The two repositories chosen were ZXing[13], which scans QR codes, and the other was MPAndroidChart[14], which creates charts and graphs for Android applications. The ZXing file selected is code that searchs the image for FinderPatterns which are markers in the corners of QR codes. The MPAndroidChart file is the code that creates Pie Charts in Android applications.

As the users were unfamiliar with the StructJumper, the users were first given a short tutorial on how to use the tool. They were given a description of the tree created and an overview of the key commands that could be used with the tool. Then, they could practice using the tool on a toy code base that did a variety of matrix calculations. Once they felt familiar with the tool, they were asked to follow a series of directions to check if they knew each of the key commands.

Before completing the tasks, users were given a chance to familiarize themselves with the code. The participants could spend up to 15 minutes becoming familiar with the code. Users were given the option to use StructJumper to familiarize themselves as well as their own methods.

Once they were familiar with the code, the participants completed three tasks. The tasks were selected to get at the two goals of the tool: improving understanding of nesting information and improving navigation within code. There were two navigation tasks, which had non-obvious so that the users would need to navigate more to determine the correct answer. There was one context task, which was nested deeply so that it was possible for participants to either miss a

[13] https://github.com/zxing/zxing
[14] https://github.com/PhilJay/MPAndroidChart

condition or to add a condition to their answer. The tasks were similar for each code base. For the StructJumper tasks, participants were asked to use the tool, but did not have to use it exclusively.

Mealin and Murphy-Hill [57] mentioned that search was a technique that some blind programmers used to navigate in the code. To simulate exploring code in which the user may or may not know keywords to look for, I phrased the two feature location tasks different. One of the feature location task's answer could be found by using search on the keywords included in the question (referred to as the *With Keywords* task). The other feature location task purposefully did not use keywords that would allow the answer to be found using search (referred to as the *Without Keywords* task). In this way, I could investigate the differences in the use of StructJumper when search was effective or was not effective for navigation.

The third task, regarding which conditions were necessary for a line to execute, involved a deeply nested line of code (referred to as the *Conditions* task*)*. In both code bases, the line was within three if statements or for loops and there was at least one other if statement on the same level as an if statement necessary for the line to execute.

The three tasks for the ZXing code were:

(1) *With Keywords:* Find the location in the code where we skip more than the normal number of rows of the image in our search for finders patterns.

(2) *Without Keywords:* Find the location in the code where after we have found all the potential finder patterns, we determine which are most likely the actual finder patterns.

(3) *Conditions:* What are the conditions necessary for line 463 to execute?

The three tasks for the MPAndroidChart code were:

(1) *With Keywords:* Find the location in the code where the text for each slice of the pie chart is added.

(2) *Without Keywords:* Find the location in the code where the size of all the chart slices are determined.

(3) *Conditions:* What are the conditions necessary for line 300 to execute?

The tasks were timed and the answers were recorded for later analysis. The task time started when I had finished reading the questions and ended when the participant had stated their answer and stopped looking through the code.

I graded the answers on a 3-point scale. Participants could receive partial credit on the *With Keywords* and *Without Keywords* tasks based on my judgement of how related or similar the section of code they indicated was to the actual answer. I awarded 2 points if they found a section with a similar outcome or was related to the correct section of code, 1 point if the connection to the correct part of the code was it was not immediately obvious, and 0 points if it was not at all related or similar to the correct answer.

For the *conditions* tasks, the participants were awarded full points if they correctly identified all the conditions and did not add any conditions. If any conditions were missing or erroneously added, a point was subtracted per condition. If a participant had more than 3 errors, they were just given a 0. It was not possible to get negative points.

Once they had completed the three tasks, the participants were asked to rate their experience completing the tasks on a seven point semantically anchored scale. They were asked:

(1) How easy the tasks were to complete: 1 – Very Hard to 7 – Very Easy

(2) How frustrating the tasks were to complete: 1 – Very Frustrating to 7 – Not at all Frustrating

(3) How well they knew where they were in the code while completing the tasks: 1 – No idea where they were in the code to 7 – Always knew where they were in the code

After both sets of tasks had been completed, the participants were asked to reflect on the differences in their experience completing the tasks both with and without StructJumper.

6.4.4 *Design and Analysis*

I used a 2x2 within-subjects factorial design with factors of the code base and whether StructJumper was used. Each participant completed three tasks for each code base. I presented the tasks in the same order for each code base, but the order was counterbalanced for each participant using a Latin square. Participants completed a total of 6 tasks for a total of 42 tasks completed altogether.

While analyzing task completion time, I used a mixed-effects model analysis of variance with a fixed effect of *Tool*, with *Participant* modeled as a random effect. For the semantically anchored scale data, I looked at the descriptive statistics.

6.5 RESULTS

To analyze the results, I used task completion times, task scores, and reported semantically anchored scale values. I found that participants were faster using StructJumper and had a better experience while using the tool as they were less frustrated and were more aware of where they were in the code.

6.5.1 *Task Completion Time*

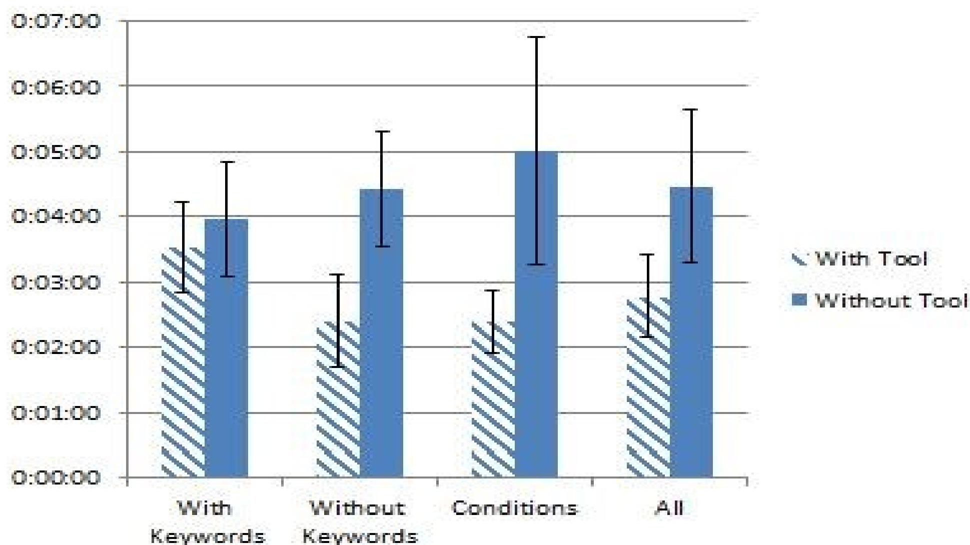


Figure 6.5. This chart shows the average completion time that it took participants to complete the three tasks broken down by type. The bars represent the standard error.

Participants completed all the tasks in an average time of 3 minutes and 38 seconds. The participants average time was 1m 47s faster with StructJumper (mean = 2m 47s, SD = 1m 41s) than without (mean = 4m 28s, SD = 3m 7s), which is a 38% decrease in average time with StructJumper. While *Tool* did not have a statistically significant effect on *Time*, there was a trend in this direction ($F(1,6) = 5.783$, $p = .053$). The largest difference in time for the tasks came on the *conditions* task (see Figure 6.5) where the average time was 2m 37s faster with StructJumper (mean = 2m 24s, SD = 1m 16s) than without (mean = 5m 1s, SD = 4m 34s).

6.5.2 *Task Score*

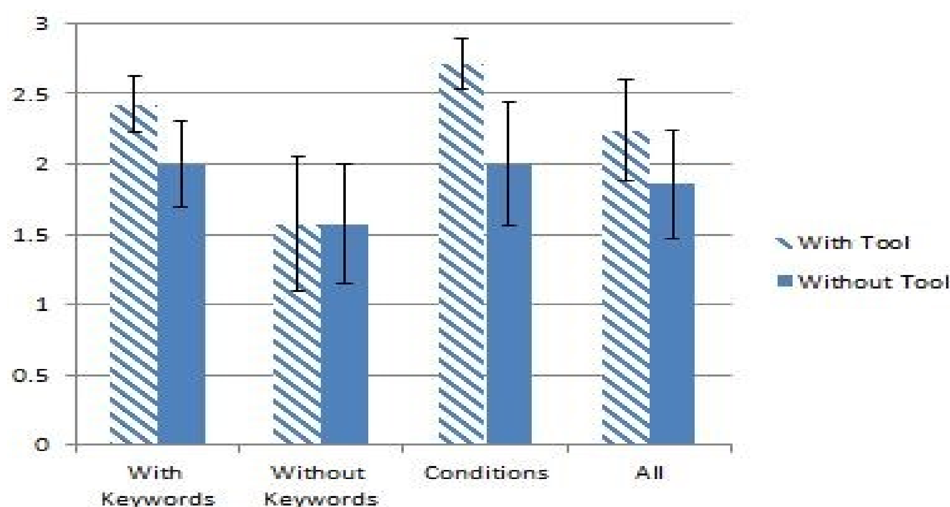


Figure 6.6 This chart shows the average score for all participants on the three tasks broken down by task. The bars represent the standard error.

The average score for participants was 2.0 (SD = .99). There was no significant effect of tool on the score ($F(1,6) = 1.038$, n.s.). The average score with StructJumper was 2.2 (SD = .94), and 1.9 (SD = 1.01) without. The largest difference in task score was on the third task, (Figure 6.6), where participants received higher scores using StructJumper (mean = 2.7, SD = .49) than without (mean = 2.0, SD = 1.15). It was not surprising that participants performed better on this task, as providing the nesting context was one of the main goals for StructJumper. When participants did the conditions task without the tool, two participants missed at least one condition and one participant added an extra condition, and one participant made both mistakes. With StructJumper, only two participants made errors, where one missed a condition and one added a condition.

On the feature identification tasks of *With Keywords* and *Without Keywords,* many of the participants found related or similar sections of code. Participants achieved full credit on 10 out of the 28 feature identification tasks.

For example, one of the tasks asked the participants to find the section of code where the code skipped more than the normal number of rows in the search through the image. For this task,

many participants found sections of code that determined how many rows of the image to skip instead of where the actual skip happened.

Another task asked participants to find where the most likely finder patterns were identified from all the potential finder patterns that were found. One participant found a section of code that identified if a single section of the image is likely a finder pattern.

Participants may have found related sections of code as opposed to the correct answer because the tasks purposefully contained as little information about the correct code section as possible. I made this decision in attempt to mimic a real-life situation where the programmer is searching for a certain feature or action in a newer or unfamiliar code base. This choice may have added difficulty for users to locate the correct section of code, and thus caused users to find a section that is similar or related.

6.5.3 *Participant Experience*

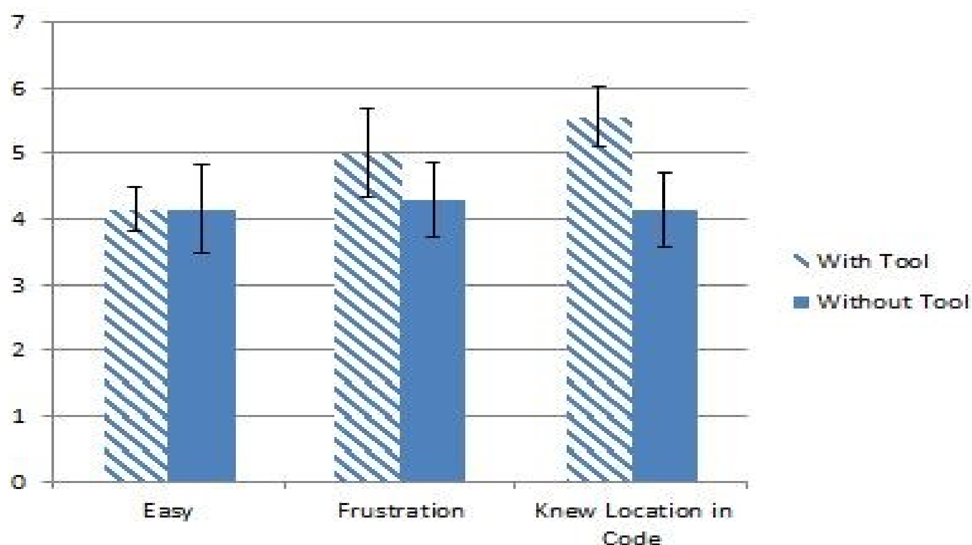


Figure 6.7. This chart show the average score for the participants for the semantically anchored questions. A higher value is better for all three questions. The bars represent the standard error.

I gathered insights on participant experience by asking three semantically anchored scale questions after each set of three tasks. I asked about how easy they found the tasks to complete, how frustrated they felt completing the tasks and how well they knew where they were in the code.

For how easy they found the tasks to complete, there was no difference in the average. The average score was 4.1 for both. However, the standard deviation was larger with the tool, than without the tool (SD = .83 vs. SD = 1.64).

When asked about their frustration when completing the tasks, the average was higher with the tool. For this semantically anchored scale, a higher number was less frustrating. The average was 4.3 without the tool (SD = 1.67) and 5.0 with the tool (SD = 1.4).

The largest difference however, was when the participants were asked about how well they knew where they were in the code (see Figure 6.7). The average was higher with the tool on this question as well. Once again, a higher number is better as it meant they knew more often where

they were in the code (1 was they never knew where they were and 7 was they always knew where they were). The average without the tool was 4.1 (SD = 1.12) and the average with the tool was 5.6 (SD = 1.40).

One thing that stood out about the responses for the question about how well they knew where they were in the code was the number of people who felt like they always knew where they were (response of 7) or almost always knew where they were (response of 6). Without the tool, nobody indicated they always knew where they were and only one person indicated that almost always knew where they were. Conversely, with the tool, two people always knew where they were and another three people almost always knew where they were.

6.6 QUALITATIVE RESULTS

After they had completed the sets of tasks with and without StructJumper, I asked the participants to reflect on completing the tasks, their ability to complete the tasks, their knowledge of where they were in the code, and their understanding the code. I asked if they would use StructJumper, and if so, for what types of tasks. Five of the participants indicated that they would use the tool.

To analyze the qualitative results, I looked at the interviews for concrete examples provided by the participants on how StructJumper changed the experience of completing the tasks and for examples on how the participants indicated that they would use the tool. Themes that were brought up by multiple participants were included in the results.

6.6.1 *Quicker and Easier*

As found in the results, there was a trend that StructJumper may have reduced the task completion time. In fact, six participants said that felt that they could navigate through the code faster and easier:

It's much easier than reading code. It's far more efficient because I'm reading relevant information. I don't have to read the complete code.

They mentioned that they could find the relevant information from the tool and skip between methods. When they were deep within a method, they could get information about where they were more quickly than without StructJumper.

6.6.2 *Better Understanding of the Layout*

Six participants indicated that the tool helped them with their understanding of how the code was laid out and how the statements were related to each other. Participants also felt that it helped them get a broad overview of the code.

This was particularly seen in the *conditions* task. As one participant was asked to complete the *conditions* task without StructJumper, they said:

I don't know how to see that in Eclipse. For that matter, in fact, I don't know how to see it in any of the IDEs I've worked in. Short of reading the code using brute force.

Another participant mentioned as they started the *conditions* task that this was one case where they would definitely want to use the tool as it made it a lot easier. This was also a common type of task that participants mentioned in the interviews that that the tool would be helpful for.

Participants also indicated that StructJumper was useful for gaining a broad understanding of the code. One participant indicated that they would be likely to use this to skim the code a few times before reading the code line by line when first introduced to a large, new code base.

6.6.3 *Lack of Cues*

Two participants mentioned that StructJumper was helpful when there is a lack of cues to indicate how far a statement is nested in the code. There were two cases of this mentioned by participants.

For instance, one participant mentioned Python. The participant mentioned that in languages like Java, there are cues, such as curly braces, to indicate the start and end of a nesting level. These cues make knowing how far nesting a statement is possible. Without these cues, the participant indicated that it is much harder to know where the statement was. Therefore, expanding this concept to a language like Python could potentially have a large impact on a blind programmer.

Another participant indicated a similar sentiment about the lack of cues. To have the screen reader speak cues such as the braces, the settings of the screen reader need to be set to speak all the punctuation. However, the participant said:

> *I really hate changing the punctuation verbosity of my JAWS, on my screen reader, to most or all. So it's always set at none. So I don't even know braces and stuff like that. And it's just that since I read through the code so many times and I understand it so well that I can figure out, ok well here has to be a brace or here is where the indentation changes. So I never use a feature ever to actually notify me. It's really annoying. So that one thing that this tool really helps with. I know, ok, well this condition, it's within the other condition, within that loop. So it kind of helps that way.*

As participants were completing the tasks, I saw some change the verbosity of the punctuation level as they were completing the tasks as they may only use that level of punctuation verbosity for programming and it may be the case that others would prefer not to have to have the screen reader read all the punctuation all the time.

6.6.4 *Change in Focus*

One of the prompts I asked the participants to reflect on was whether the tool affected their ability to understand the code. A few brought up that it removed the number of things that they would have to focus on as they were going through the code, which may make it easier to understand. For example, some participants mentioned that it allowed them to focus less on the little details of how to navigate or keep track of where they are in the code. One participant said:

> *You know, the navigation part, you know, without it, I was more focusing on that probably. How do I navigate, how do I get to the next thing, what keywords can I use to easily jump to where it needs to go? So, you know, without having to do that, you know, maybe I was focusing more on understanding what the code actually does.*

Another participant indicated that they would use the tool when they were trying to understand the code. They said that it was useful when:

> *Trying to keep track of what level you're on basically… If I'm reading through code and trying to remember how many right braces you have remaining, how the different conditionals are related to each other.*

It allowed them to see the relationship between blocks of code more easily. For instance, one participant indicated that is made it easier for them to know which conditionals a statement in the code it was nested under.

6.6.5 *Unfamiliar Code*

Multiple participants indicated that this tool would be more helpful for unfamiliar code than for code they know well. Many indicated that this was because, for code in which they are familiar, they already know the keywords they can use to jump to a section of code or what statements are under which conditionals. StructJumper aims to help provide this information, so it helps them learn when they first see a code base.

In familiar code, the biggest benefit this tool may add is the ability to navigate more quickly to that line, as mentioned by one participant. The tool can be used to skip lines and the user can quickly skip through code as they know each where the section is nested. This could be better than a keyword search as frequently the keywords may not be unique to that section of code only, and far better than moving through code line by line.

6.6.6 *Programming Use*

There were some participants who mentioned use cases for the tool that were more related to coding and finding errors in code. Some participants mentioned that this might be good at finding errors such as improperly matched braces. Another area that a participant suggested it might be useful to use it to skim over the logic for errors, such as looking at what the switch cases are to determine why the correct case is not getting called.

One participant mentioned that in his work, he might receive feedback from customers about the functionality of the application. He could then use the tool to navigate to a section of code where the functionality in question is and then fix the problem or make the requested change.

I am interested in investigating the use of this tool for programming by conducting follow-up studies.

6.7 DISCUSSION

Overall, the participants seemed to like having access to the tool as it improved the experience for them. Based on the interviews, there were two broad categories that the improvements helped with, providing fasters ways to look up information and by decreasing the amount of information that needed to be held in memory.

These improvements could potentially have impact on the actions of users who use this for coding which may not have been seen during the evaluation as it only focused on navigational and nesting tasks. By decreasing the difficulty of looking up information, users may look information such as nesting up more often. Additionally, by not needing to remember as much information, users may be able to focus better on the coding. Therefore, it would be interesting to learn how participants use this tool to perform different coding tasks as well. Some participants mentioned could be used in debugging to try and find logical errors as well as navigation to find problem sections of code. In addition, a potential future study could look at the use of StructJumper in the wild. This would allow me to understand how its use might change with familiar programs versus unfamiliar programs.

While participants felt it made them faster, there was not a significant effect of the use of StructJumper on the time it took users to complete the tasks. I believe the lack of significant effect of the tool on time comes from some variation in the participants and the small sample size. For 8 of the 21 tasks, participants were slower with the tool than without. Of the 8 slower tasks, 2 were on C*onditions* tasks, 3 were on the *With Keywords* task and 3 were on the *Without Keywords* tasks.

I attribute this variation to a variety of reasons. In one case, the participants only found one of the conditions without the tool and with the tool found all three and spent more time double-checking them. Two of the participants that were slower on some tasks with the tool did not use

anything like the package explorer or outline. They were efficient with a line-by-line navigation approach. All but one other of the participants used either the package explorer or the outline and may be more accustomed to using a similar tool.

For some participants, it was not clear what made them slower with the tool than without the tool. It may be that they were more efficient with their current method. It could also be that they just found the navigation task that they received while using the tool more difficult.

This tool would likely be most useful for code that used meaningful method and variable names. If the names were more generic, it would be harder for a user unfamiliar with the code to determine which statements they should look further into. Meaningful names could allow a user to guess what is happening in a statement or method and therefore determine if is the section they are looking for.

StructJumper provided useful navigation and structural information about the code that could make it a valuable addition to an IDE, but it only addressed some of the issues faced by blind programmers. Other areas of programming also present difficulties to blind programmers that need to be addressed, such as class diagram creation and debugging. There has been some research in these areas, such as the work done to improve debugging by providing the same structural information [77,78,83], but further research is needed.

Finally, it would be interesting to research whether this tool is beneficial to sighted programmers and investigate how their use of the tool varies from blind programmers. Much of the benefit of the tool was overcoming the difficulties that screen readers have in jumping many lines of code, which can be done more easily with sight and visual clues like indentation. However, there may be benefits for sighted programmers to only look at the included lines of code and ignore other code sections as they are collapsed on the tree.

6.8 SUMMARY

I created StructJumper, an Eclipse plug-in that allows blind programmers to quickly navigate through the code and see the how specific statements are nested within the code. I ran a user study with seven blind programmers and found that there is a trend that the participants were faster completing the tasks with StructJumper.

I also found that participants were positive about the tool and that they would be interested in continuing to use the tool. The participants found it quicker to navigate through the code and, thought that StructJumper provided valuable information about the conditionals that a line of code is nested within.

Chapter 7. DYNAMIC GRAPHS

In this chapter, I am investigating how I can make dynamic graphs more accessible. As there are many types of graphs, in this chapter, I focus on those graphs that are made up of nodes and edges. I consider a graph to be dynamic when there are multiple versions of the graph as it undergoes changes. These changes can be shown through an animation where only one graph is visible at a time or as part of a static image showing the changes side by side. In my investigation into dynamic graphs, I sought to answer *RQ4: How can we make it easier for someone who is blind to understand changes in graphs?* A student's experience in the introductory course will have an impact on whether they will continue studying the subject. As many introductory courses use a programming environment with code visualizations, this can be potential barrier that would deter a student from studying computer science. And dynamic graphs do not just occur in code visualizers. They also occur in text books, slides, and other materials. Since they play such a large part in the curriculum for computer science, it is important that blind students have an effective way of accessing these graphs. Building on prior work that looked at static graphs, I investigated different modalities for handing the change between the different versions of the graph. The results of my user study show that participants could use both modes to answer the questions regarding changes in the graph and wanted to have access to multiple modes so they could pick a mode depending on the task they are trying to complete. [15]

[15] The work in this chapter was done in collaboration with Richard Ladner

7.1 INTRODUCTION

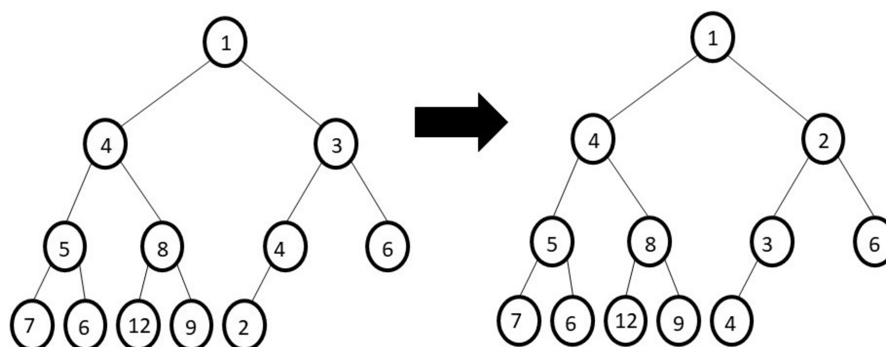


Figure 7.1. A sample image that could occur in a slide or textbook when teaching the insert operation of min-heap. This image shows the before and after of the percolate up operation after node 2 has been inserted into the min-heap. This image illustrates the use of dynamic graphs to teach CS concepts.

Dynamic graphs are frequently used in computer science to help teach concepts to students. They appear in many use cases from lecture slides (Figure 7.1) to interactive code visualizations and in all levels of courses. As they are used so frequently through computer science curriculums, it is important that blind students have access to this material.

For the lecture slides or textbooks, someone can create an alternate format of the visualization ahead of time, but that is not possible for the code visualizers that reflect the code a student has written. As far as I am aware, none of the development environments with code visualizers developed a method for blind students to access the visualizations. The accessibility statement for jGRASP stated that while they seek to support assistive technology use for the IDE, they did not support access to the visualizations as automatically generating useful text representations was not possible [45]. While it is not possible to auto-generate text summaries of the graphs, I believe that instead, one possible solution is to allow the blind students to navigate the data structures within the visualizations.

This required investigating the best methods of creating accessible dynamic graphs. While many researchers have investigated creating accessible static graphs, few have looked at dynamic graphs.

In this chapter, I describe an investigation of techniques that allowed a blind student to quickly identify changes in a graph. Sighted students who are looking at graphs can identify a change in a specific section of a graph by looking at the same location in the two versions of the graph. However, approaches that replicate a similar approach for blind students have not been investigated.

With the tool, I aim to evaluate the following research questions:

(1) How can we design interactions when a user is navigating between multiple versions of a dynamic graph?

(2) Does creating temporal connections based on location improve the understanding of the changes in a graph?

To help answer my research questions, I created a prototype of a dynamic graph tool which had two modes. The previous location mode treated the two graphs as independent and therefore the navigation was not linked between the different versions of the graphs. The relative location mode assumed that the graphs were not independent and linked the two versions of the graph based on the node locations. I evaluated the tool with 7 users who identified as blind or low vision and found that participants could answer questions about the changes in the graph. I then interviewed them to gain insights into their preferences and approaches for the modes.

My contributions are:

(1) The dynamic graph tool prototype with two interaction modalities for handling focus changes between the versions of the graphs.

(2) The results from my evaluation which showed participants could answer questions about the changes in the graphs and wanted access to a variety of modes to handle the focus change.

(3) Insights into how designers should create accessible dynamic graph tools for blind users.

7.2 RELATED WORK

Beyond the work presented in Chapter 2, section 4.2 on efforts to make graphs accessible to blind students, there has been much prior work on the visualizations of algorithms for sighted students.

In a survey of 255 CS educators, Isohanni and Järvinen found that approximately half of the courses the educators taught used visualizations, with 21.6% of courses using them regularly during the course [42]. The animations can be used for many purposes to support student learning. They can be pre-made to show specific algorithms (e.g. [26]) or they can be dynamically generated to show a programs output (e.g. [36,38]). In a meta-analysis of studies done on the effectiveness of algorithm visualizations, Hundhausen et al. found that the visualizations were most effective when they were used actively by students (e.g. predicting outputs, creating their own data sets to run the visualizations on, etc.) [41].

As these visualizations are commonly used in courses and have been shown to be effective when properly used, it is important to provide access to these animations to blind students. While there had been prior work that provide suggestions on how to provide access to graphs, little work has studied the design of interactions when there are multiple versions of a graph such as in a visualization, which is what I seek to investigate in this chapter.

7.3 System Design

To test the two interaction modalities, I created a dynamic graph prototype. The goal of this prototype was to develop enough of the features needed to evaluate the interaction modalities. Features such as the creation of the graphs are currently not implemented, as the goal for this study was to provide design guidelines to developers of interactive textbooks, code visualizers, or other similar technologies for implementing the navigation of data structures to make them accessible to people who are blind. Thus, at this point, all graphs used in the study are hard coded.

In the sections below, I provide an overview of the functionality that was included in the prototype and the two modes which differ in how they handle the temporal changes.

7.3.1 *Overview of the Prototype*

The basic functionality of the prototype was that a user could navigate around the graph using the keyboard. The initial focus was placed on a node. From there the user could move to the nodes connected to the current node via in edges or out edges. In the case of undirected graphs, the list of nodes reached via in edges and out edges was the same. Users could also switch between versions of a graph and focus would be handled according to the mode they were in. Currently, in the evaluation I only look at graphs in two temporal states, past and future, of the dynamic graph. However, the approaches described could potentially work with additional temporal states. Table 7.1 shows the complete list of actions that could be taken in the prototype.

Table 7.1. The list of keyboard shortcuts for the prototype and their association actions and what is spoken when the action is taken. I assumed that the starting point is the graph in Figure 7.2a, and that the user is in the previous location mode and the user reached Brad via the out edges of Tom.

Keyboard Shortcut	Action	Spoken Cue
Left / right arrow	Cycles through the nodes in the list of edges that the user has selected	*Right Arrow:* Node Jill. 1 in edges and 0 out edges. Node 3 of 3. Connected via edge from Tom. *Left Arrow:* Node Alex. 1 in edges and 2 out edges. Node 1 of 3. Connected via edge from Tom.
Shift + left / right arrow	Moves focus to the past/future version of the graph	*Shift + Right Arrow:* Graph 2 of 2 Current Node Tom. 0 in edges and 4 out edges. *Shift + Left Arrow*: No past graph
i	Moves focus to the list of nodes connected via in edges to the node with focus	Node Tom. 0 in edges and 3 out edges. Node 1 of 1. Connected via edge to Brad.
o	Moves focus to the list of nodes connected via out edges to the node with focus	Node Chris. 1 in edges and 0 out edges. Node 1 of 1. Connected via edge from Brad.
p	Returns focus to the node that provided the list of edges that the user is currently navigating through	Node Tom. 0 in edges and 3 out edges.
Tab / Shift + Tab	Cycles through the nodes	*Tab:* Node Jill. 1 in edges and 0 out edges. *Shift + Tab:* Node Alex. 1 in edges and 2 out edges.
s	Provides a summary. Gives which graph it is, how many nodes and edges and the user's current location	Graph 1 of 2. 7 Nodes and 6 Edges Current Node Brad. 1 in edges and 1 out edges.
r	Repeats the user's current location	Node Brad. 1 in edges and 1 out edges. Graph 1 of 2
m	Switch modes	Relative Location Mode
Ctrl + f	Search functionality	Type the name of the node you want to jump to and press enter

As the tool was a prototype, it was self-voicing. It was set up to use the user's own screen readers if they have the Java Access Bridge enabled. It did this by continually updating the accessible description of the window. For users who did not have the Java Access Bridge enabled, the tool used FreeTTS[16] to speak the information.

When the tool spoke, it provided the user information about the node that currently had focus. For the node with focus in Figure 7.2, graph a, the following would be spoken, assuming that the user was looking at the out edges of Tom:

Node Brad. 1 in-edges and 1 out-edges. Node 2 of 3. Connected via edge from Tom.

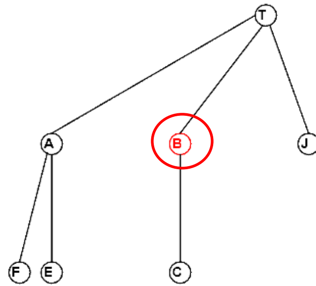
The first piece of information was about the node that has focus. It listed the name, value (if present), and information about connections ("Node Brad. 1 in-edges and 1 out-edges"). This information could change based on node or graph type. For instance, an undirected graph would not need to separate in and out edges or for some graphs, nodes have an associated value.

The second piece of information it provided was about the context of how the user reached that node. It tells the user where in the list the focus was ("Node 2 of 3") and how the focus got there ("Connected via edge from Tom"). If the user was looking at in edges, it would say "via edge to Tom" instead of "via edge from Tom." For the undirected graphs, the information is simplified and just says "connected to Tom." If the node is reached by a method other than an edge (e.g. tab or search), then the second half of information is not included.

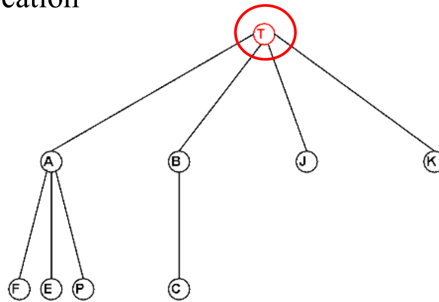
[16] https://freetts.sourceforge.io/

7.3.2 *Modes*

a)



b) Previous Location



c) Relative Location

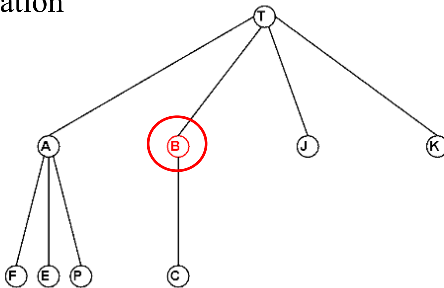


Figure 7.2. This shows the differences in how focus (highlighted in red and circled) moves in the two modes. The graph in a) is the present graph and the graphs in b) and c) show where the focus would be in the future for the two modes. In b), we are assuming that this is the first visit to the graph, so it goes to the initial placement in the graph. In c), it goes to the same relative location.

In this section, I show the differences between the two modes that I compare in this chapter. The modes varied in how they handled the focus placement when a user switched between different versions to explore the temporal changes of a graph. The two modes are highlighted visually in Figure 7.2.

7.3.2.1 *Previous Location Mode*

In previous location mode, the two graphs were treated as independent. Therefore, when a user switched between the past and future, the state of the navigation of the current graph is saved and the state of the other graph is loaded. In the case that the user has never visited the other graph, it will go to the initial state. In Figure 7.2, I show how the focus would move when the user moved to the future graph for the first time. We see that while the focus in the tree was in the second level for the first graph (a), the focus was at the top for the second graph (b).

7.3.2.2 *Relative Location Mode*

In this mode, I assume that the graphs are not independent and therefore, the tool should use the location of the focus in the current graph to determine the location of the focus in the other graph. I used techniques that sighted users might use to identify changes in side by side images as an inspiration for the relative location mode. One technique is to search through the image in a systematic way by looking at one location of an image and then looking at the same location in the other image to identify if anything had changed and continuing that back and forth through the image. This is the technique that I based the relative location mode on by linking the corresponding location in both graphs, blind users could use a similar technique to identify changes.

For graphs where there is a defined structure, the tool can harness that structure to link the locations of two versions of a graph. For instance, with a binary tree, we can define a traversal path without reference to specific nodes (e.g. we can define the traversal path by its path from the root using references to the left or right child and specific values of nodes do no matter). Similarly, with a linked list, we can define a node based on its positional location in the list (e.g. first item or fourth item). In these cases, when a user moved to the past/future version the graph, the focus should be placed on the node that was in the same location as the node with focus in the current

graph. In Figure 7.2, in both the versions of the graph (a and c), the focus was in the same location, the second child of the root.

For cases where there was not a defined structure, the focus was sent to the same node in the other version of the graph. This assumed that nodes had an identity which can be used to determine if the node existed in the other graph.

7.3.3 *Changes in Response to Pilot*

I solicited feedback on the initial prototype. The participant I tested the early prototype with reported advanced graph expertise and expert software development expertise. There were two main changes that were made in response to the pilot.

One critique was that the navigation initially included edges. A user would move from a node to a list of edges. Then from the edge, they could decide which node to move to. The pilot participant indicated that this created an unnecessary step in navigation which added additional cognitive load. They did not care about the edges except that they connected them to other nodes. After the pilot, I switched the list of edges to be a list of nodes that were reachable via the edges that were previously in the list. They could still receive information about edges, such as the weight of the edge, as the final version includes information about how that node was reached.

The other change was the keyboard shortcuts. Initially, the out and in edges were reached by the up and down arrows. By using only the arrow keys for navigation, the pilot participant felt that it implied a navigation approach that was not actually supported by the keys semantics. By switching to “i" and “o”, I hoped that users would associate the key with the list they produce based on edges and not a hierarchy or spatial relationship.

7.4 EXPERIMENT DESIGN

To evaluate the effects of the two modes on a user's ability to complete tasks and their experience while completing the task, I had them complete a series of tasks with each mode. After completing the tasks, the participants were asked questions about their experience.

7.4.1 *Participants*

I had 7 people complete the study, 6 identified as blind and 1 identified as low vision. Participants were recruited using email lists for blind programmers (both international and US-based) and snowball sampling. All participants used a screen reader and reported having some experience with graphs and software development.

I had six males, and one participant who did not identify as either male or female. The average age for the participants was 37.1 (SD = 14.8). All participants had experience with graphs: six participants rated their expertise as intermediate and one rated their expertise as advanced. Additionally, as some concepts came from programming (e.g. linked list), I had the participants rate their expertise with software development. Two participants rated their expertise as novice, four as intermediate and one as advanced.

7.4.2 *Set-up*

I conducted the study remotely over a screen sharing service, either Skype or Google Hangouts, depending on the participants preference. Using this service, I could watch the participants complete the tasks. Additionally, the tool would log everything that was spoken by the screen reader, which allowed me to analyze the actions taken by the participants.

The participants were sent the software, a reference guide, which detailed the two modes of the tool and the keyboard shortcuts, and a document with the tasks. Participants were not

required to download the documents and most did not and solely relied on me if they forgot the task or keyboard commands.

Six participants used the self-voicing mode with the text-to-speech engine and one participant started using the self-voicing mode, but switched to their own screen reader after one task.

7.4.3 *Procedure*

Participants completed the study in a single session. Before the session, I collected participants' demographic information and their level of experience with graphs and computer science. During the session, the participants completed a series of tasks with each mode. At the end of the session, participants reflected on the experience of completing the tasks with each of the modes.

Participants completed eight tasks, four with the previous location mode and four with the relative location mode. The first task for each mode was always a practice task to allow the participants to become familiar with the tool and the mode. The participant would first just explore the graph and become familiar with the tool. Then they would be provided the task. During the exploration and task, I would answer all questions and there was often a dialog with the participant to clarify how it was working, what the information presented meant, etc.

Then I had the participants complete three types of tasks based on the scenarios where they might encounter graphs that are changing dynamically. Those three scenarios and their associated tasks are outlined below. The tasks were done in the same order for each mode and every participant. They were ordered in increasing complexity of the graphs. Below I list the tasks. As the graphs were different for each set of tasks, I provide one set of tasks as an example as all that changes in the tasks in the node names. The tasks were:

(1) *A node has been added to a linked list. Determine after what node it was inserted in the list.*

(2) *I provide the academic family tree for Jane, which shows who Jane has advised, who Jane's students have advised, and so forth. The tree is shown at two points in time. Which of Paul's students started after the first tree?*

(3) *On this undirected graph, identify which of the nodes connected to node cat have changed values.*

The first two tasks revolve around the scenario where a student is using a code visualizer to understand the changes that happen to a data structure when they run their code. In this scenario, there were two cases: 1) the student was trying to understand what changes were happening to the data structure and did not know where the change will happen and 2) the students knew where the change should occur relative to the first graph and needed to check if their understanding was correct and the change happened properly.

The third task was based off the scenario where a program was visualizing a graph algorithm. In this case, the graph structure was often unchanging (i.e. the nodes and edges remained the same), but had associated values that may be changing. For instance, when running Dijkstra's Algorithm[17], the shortest path thus far, the previous node in the shortest path found thus far, and whether the shortest path has been found were the values that may be changing, but the nodes and edges remained the same.

For each mode, the same set of three tasks were done. I provide the one of the set of tasks above to illustrate the tasks. For the other mode, the structure of the task remains the same. The only differences were that there were slightly different graphs and therefore the names of nodes

[17] https://en.wikipedia.org/wiki/Dijkstra's_algorithm

mentioned in the tasks were updated to reflect the graph. The ordering of the modes and sets of tasks were counterbalanced.

For each of these tasks, the task was explained and the participants had a chance to ask any clarifying questions regarding the task. They were then provided the key command that would load the task. The timing began when the graph loaded and stopped when the participants provided the final answer. During the task, I answered questions regarding the tool or further clarification of the task. I did not answer questions regarding whether the participant was on the right track or how to complete the tasks. At the end of the task, I recorded the time and answer.

I graded the answers on a one-point scale. As tasks one and two for each mode only had a single answer, they were both either 0 or 1. Task three had multiple values that changed (two in one version of the task and three in the other). For this task, participants could get a 0, 0.5 or 1. A score of 0 was if they got none of the nodes that changed. Participants received a 0.5 is if they got it partially correct, with some but not all the changes identified or if they erroneously indicated that a node had changed value.

After completing the tasks for each mode, the participants were asked to rate their agreement with four statements on a seven-point Likert scale where one was strongly disagree and seven was strongly agree. The four statements were:

1. The tasks were easy to complete
2. The tasks were frustrating to complete
3. I had to hold a lot of information in memory
4. I could easily determine what had changed between the two graphs

After completing the tasks with the two modes, participants were asked to reflect on the experience completing the tasks with the two modes. I wanted to gain an understanding of their preferences, approaches, and the effects of the mode on what they held in memory.

At the end of the session, I asked participants to email the log file that was generated to me.

7.4.4 *Design and Analysis*

I used a 2x2 within-subjects factorial design with factors of the set of tasks and mode. Each set of tasks had the same three types of tasks, though with slightly different graphs. Participants completed one set of tasks per mode, for six tasks total. I presented the tasks in the same order within the task set for all participants, but counterbalanced the order in which the mode and set of tasks were presented. Therefore, the seven participants each completed a total of six tasks for a total of 42 tasks completed altogether. Due to the small sample size, I used descriptive statistics to look at the time, accuracy, actions, and Likert data.

7.5 RESULTS

In this section, I present the results of the study. I show that participants were faster with the relative location mode with more than half the tasks and the accuracy was high with both modes.

7.5.1 *Data Cleaning*

There were a few issues that arose during the user study which affected the data. All bugs in the software that were identified during the user studies were fixed before the next participant. However, theses bugs affected the data, so I will describe each of the issues and how I handled them.

There were issues with the log file which allows me to see the actions taken. When the program started up, it created a log file in the same folder that the program is saved. There was one log file that was missing as the participant ran the program directly from the Drive and therefore the log file did not save. A second participant was missing a single task worth of data as they needed to restart the program after the first task and it overwrote the file before the email with the first log sent. Finally, participants were asked not to continue navigating after providing their final answers, but some did not listen or needed to reconfirm the names of the node when asked to repeat their answers due to the audio breaking up. In those cases, the additional movements after providing their answer were removed from the log file.

Additionally, there was a small bug in the prototype that was fixed after p2. This bug made it possible for a user to accidentally switch focus to a graph from a previous task without meaning to. This happened on two separate tasks for one participant. For one task, it only happened once and 9 of 56 actions were on the wrong graph. The participant could return to the correct graph without needing to reload the task. Therefore, I have removed the actions that were taken when the participant was exploring the wrong graph and have thrown out the time data. The second time the participant encountered the error, the participant both encountered the bug and was switching between the modes within the task, so all the data associated that task for that participant has been thrown out.

Finally, a second bug affected the search feature, searching the display name instead of the spoken name. It only affected the second task and only one participant tried to use the search on this task. As this only affected a single task for the participant, I have discarded the time data for that task and have removed the attempts at using the search feature from the data logs.

7.5.2 *Time*

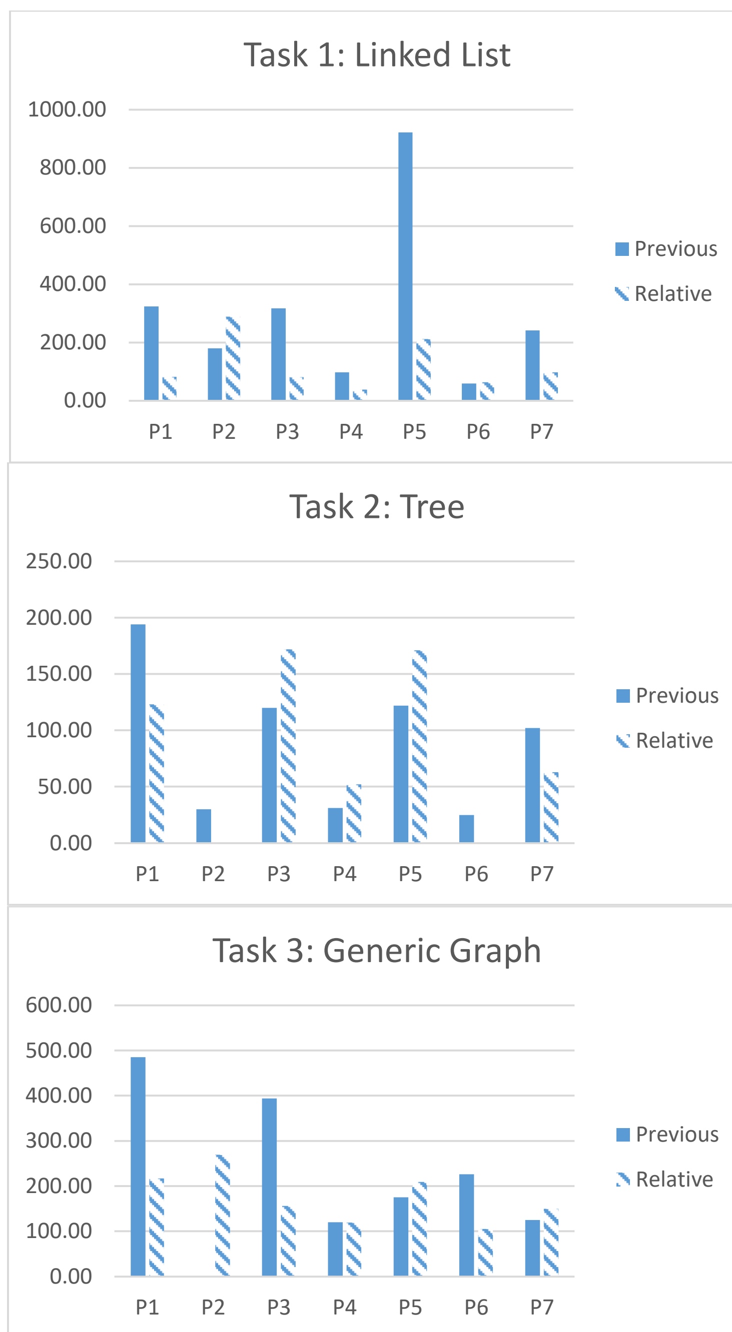


Figure 7.3. These graphs show the time taken in seconds to complete the tasks with each of the modes by participant. Each bar represents a single task. The three graphs are for each type of task with the solid bar representing the time with the previous location mode and the striped bar with the relative location mode. The missing bars are due to the data points that needed to be removed as described in 7.5.1.

The average time it took to complete the tasks was 178 seconds (SD = 160.0). Participants took an average of 140 seconds (SD = 73.7) with the relative location mode versus 214 seconds (SD = 207.9) with the previous location mode. Users were faster using the relative location mode in 11 out of the 18 sets of tasks (due to the thrown-out data points, three sets of tasks only had one mode and cannot be compared). The breakdown of the time it took each participant on each task is shown in Figure 7.3.

7.5.3 *Accuracy*

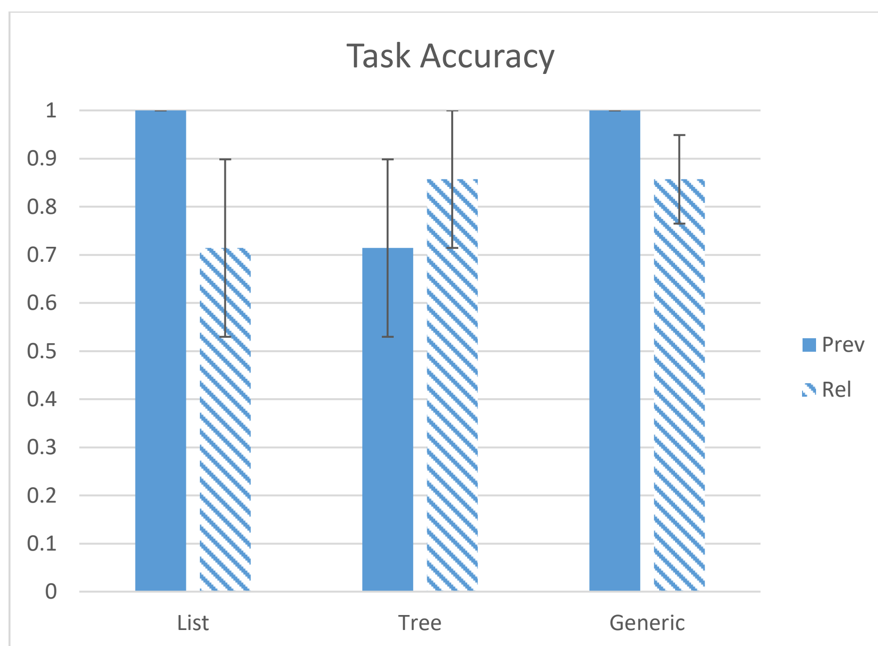


Figure 7.4. This shows the average accuracy for each completed task. Participants could get a max of 1. The error bars represent the standard error.

The participants' answers were compared to the correct answer and then graded. For the first two tasks, the participants could only get 0 or 1. For the third task, there was a multi-part answer so participants could get 0, 0.5, or 1. Overall participants were able to answer the questions correctly.

The average accuracy was for the tasks was 0.85 (SD = 0.34). Participants had an average of 0.90 (SD = .31) for the previous location mode versus .81 (SD = .37) for the relative location mode.

Participants missed two questions in the previous location mode, both on the tree task and five questions on the relative location mode, two missing the linked list, one the tree task, and two getting the generic graph task partially correct. The incorrect answers appeared to be from either misinterpreting the information from the tool or forgetting a node. The breakdown of the average accuracy by task type is shown in Figure 7.4.

7.5.4 *Actions*

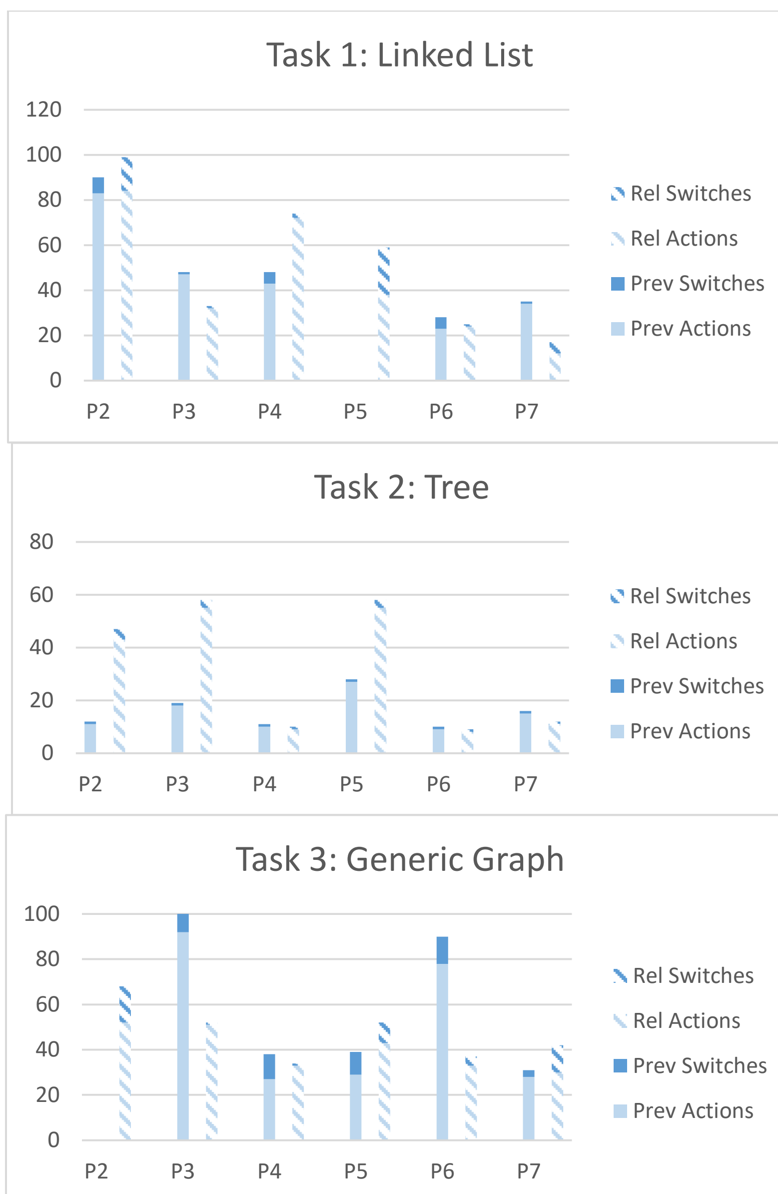


Figure 7.5. These graphs show the number of actions taken by each participant to complete the tasks. The graph switches are represented by a darker blue. Each bar represents a single task. The three graphs are for each type of task with the solid bar representing the previous location mode and the striped bar the relative location mode. The missing bars/participants are due to the data points that needed to be removed or were missing as described in 7.5.1.

For each participant, I had the log file of what was spoken by the screen reader. Using this, I analyzed how many actions the user took. Actions included moving to a new node, repeating the

current node, etc. I also analyzed how many times users switched between the past and present as the information spoken when changing graphs is unique.

Participants took an average of 42.1 actions (SD = 26.3). They averaged 40.3 (SD = 29.2) with the previous location mode versus 43.7 (SD = 24.2) with the relative location mode. Participants switched graphs an average of 5.0 times (SD = 5.3). They averaged 4.4 (SD = 4.1) with the previous location mode and 5.6 (SD = 6.3) with the relative location mode. Graphs that show the breakdown of actions within a graph and graph switches by task are shown in Figure 7.5.

7.5.5 *Participant Experience*

After each set of tasks, I asked participants to rate their agreement with 4 statements on a 7-point Likert scale, with 1 as strongly disagree and 7 as strongly agree. The exact statements are provided in Section 4.3 of this chapter. These statements looked at how easy the tasks were, how frustrating the tasks were, how much information had to be held in memory, and how easy it was to understand the changes.

The average Likert scores are provided in this table:

Table 7.2. Summary of the Likert scores

	Previous Location Mode	Relative Location Mode
Easy	5.6 (SD = 1.72)	5.3 (SD = 1.60)
Frustration	2.6 (SD = 1.27)	3.4 (SD = 1.72)
Memory	3.1 (SD = 1.77)	3.7 (SD = 1.50)
Changes	5.7 (SD = .95)	5.8 (SD = .90)

Overall the results for the two modes were similar. I saw little differences in the averages, with three of the questions favoring the previous location mode and one, about understanding the

changes, favoring the relative location mode. Figure 7.6 shows the breakdown of the scores for each Likert question.

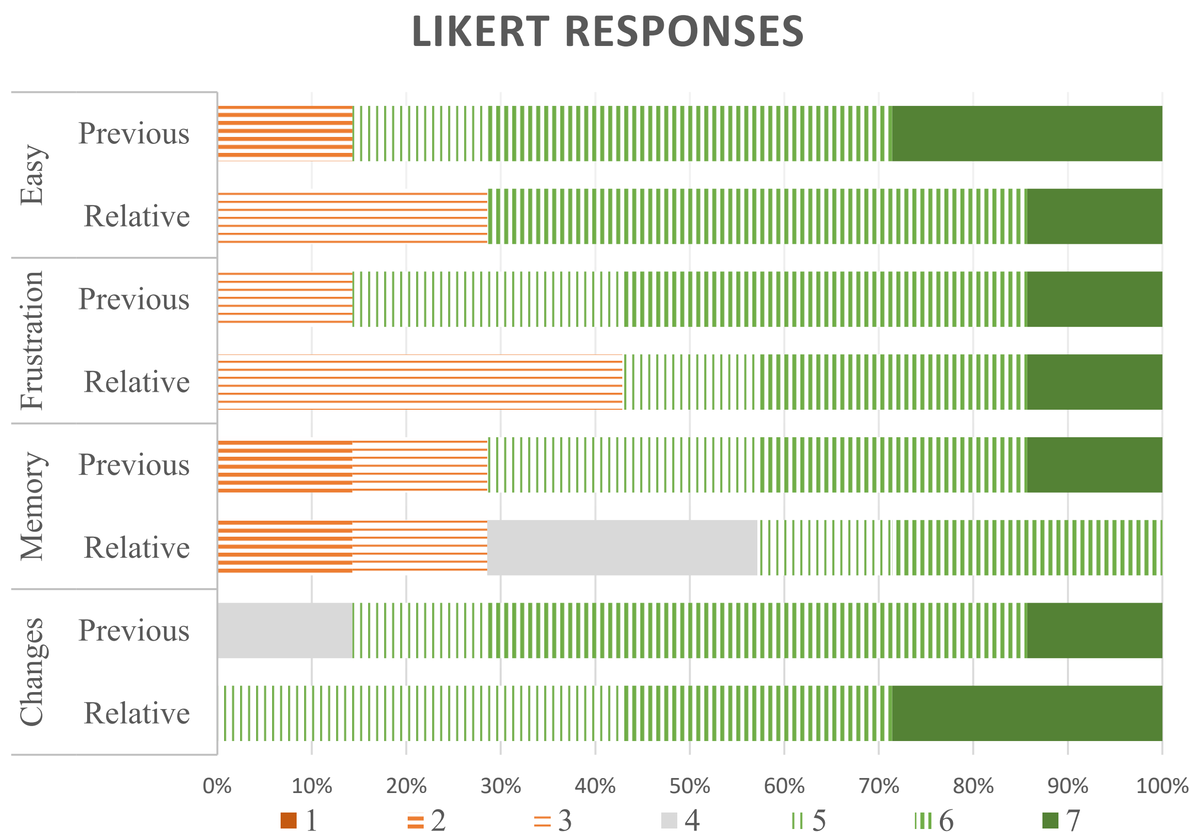


Figure 7.6. This figure shows the percentage of participants with each Likert response. Agreements is shown with green vertical lines, disagreement with orange horizontal lines and solid gray is the middle value. The less white visible, the more extreme the agreement/disagreement. The values of the Likert question relating to memory and frustration are swapped so that the higher numbers are better for all four questions.

7.6 Qualitative Results

To analyze the post-session interviews, I transcribed the recordings of the interviews and then noted down the themes that the participants brought up. Relevant themes that were brought up by multiple participants are discussed below.

7.6.1 *Preferences*

During the post-session interview, I asked the participants about their preferences for the different modes. The most common answer I got was that it depended on the task. However, even those that had a preference wanted access to both modes. One participant said:

> *The more generic the better obviously. The more modes there are, the more avenues people will have to use them to their advantage. – p2*

Multiple participants made comments that experience would have an effect. During the interviews, participants would often add qualifications about their statements regarding their preferences or their experiences that their opinion may have just been because they were still learning the tool or mode.

7.6.2 *Lost Context*

Many of the participants found it easy to determine what had changed with the relative location mode, but it came with a cost of losing their context in some situations.

The mental model that some participants had when exploring the graph did not match the model that was used by the tool in the relative location mode. If B is a neighbor of A, then when many participants reached B from exploring the edges of A, they were keeping the context that they were exploring the neighbors of A in memory. Then when they switched graphs, they were just on B and not in a list of edges. This was an intentional choice in the design of the tool as if B is no longer a neighbor of A or A or B no longer exist in their current locations, then it is unclear how to handle to transfer that state to the future graph. Rather than have the graph behave

differently in a wide spectrum of cases, it was more consistent to have it place focus on a node and not in a list of edges.

But many participants found the loss of the context frustrating for tasks like task 3:

> *It was great to jump to the same node, but actually when you jump back, you lost that context. So, you had to faff around a bit, finding your place again. – P5*

As task 3 required participants to look at all the nodes connected to a specific node, they would have to take extra steps getting back to the list of the nodes connected to the specified node and remember where they were in the list.

7.6.3 *Saved State*

Multiple participants mentioned that the previous location mode allowed them to save their state, which they liked. As the previous location mode allows users to resume the state they left it in, I found that users would use that to their advantage in many ways.

Often the main piece of information that participants were saving was where they were in the list of edges when they were exploring. This was information that was lost in the relative location mode, so many participants mentioned that this made it easy for them to switch back and forth and continue looking through the list of nodes they were currently looking at. Other participants would purposefully move the focus on the graph they were currently on to the node that they wanted to be on when they came back to the graph.

7.6.4 *Memory*

I found that participants had different opinions on which mode required remembering the most information and it often depended on what they found easier to hold in memory.

There were two things that participants held in memory that were unique to the specific modes. In the previous location mode, participants would have to hold in memory the state they left the other graph. In the relative location mode, because the user lost their context when they switched, participants needed to remember how to get back to the state that they were in last.

I saw some participants adopt different strategies with the modes so that they would need to hold less in memory. For the relative location mode, one participant was able to decrease significantly what they were holding in memory for the linked list task:

> *[For the previous location] mode I actually tried to have a general understanding of the entire graph, particularly for the linked list. ... But for the [relative location] mode for the linked list, I actually did it very blindly. I didn't try to keep it memory at all. – P7*

They were able to do this by checking the node at the same location in the future graph for every node until they came across one that was different in the future graph than the past graph. When that occurred, they knew they had found the inserted node. This was a different strategy than what they used for the previous location mode. In that mode, they held the entire list in memory and then compared the future version to the version in their memory.

For the previous location mode, participants would often hold the state of the other graph in memory so they would have the context when switching back to that graph. Because that did increase the cognitive load, some participants would increase the number of times they would switch graphs:

> *I think the fact that it saved, I started depending on – instead of trying to keep each location synchronized in my mind better or at the same time, I would just rely on being able to switch back and check. – P6*

By switching between the graphs frequently, participants could keep the graphs in a similar state, decreasing what needed to be kept in memory as they explored.

7.7 DISCUSSION

Based on watching the participants complete the tasks and the feedback in the post-session interviews, there are a few changes that I would make for future versions.

One is based on the complaint of the context being lost in the relative location mode. This complaint was common while completing the third task. This is partially because unlike the other tasks, in the third task participants did not tend to navigate within the future graph. They would switch to the future to look at the value and then instantly switch back. When they returned to the original graph they had the most issues with losing context as they would then have to recreate the state that was lost when they moved to back to the original graph (e.g. get back to the 4th item in the list of out nodes from cat). In this case, a peek functionality might be useful. This would allow the users to see the node at the same location, but not lose their context of being in the list of out edges.

Multiple participants indicated that a third mode, which connects the graphs temporally by node ID, would be useful as well. Some participants thought initially that that was how the relative location mode worked as for two of the three graphs there were no nodes that changed location. For the one task that did, the linked list, many of the participants did not experience this much. Many participants simply memorized the linked list before switching to the future graph, so they did not experience the mode taking them to a different node than the one they left from or did not register it as they were focused on moving to the front of the list to compare it to the list in their memory. Those who wanted this third mode thought that it may be useful when the graph is

changing frequently and could be used to quickly determine if a node is still present and how its neighbors changed.

While I was trying to isolate the effect of the modes on the participants experiences, I did not look at how I could optimize for specific graph types, nor did I try to look at other ways to present the understanding of changes in the graph.

For this prototype, I treated all the graphs the same in terms of what was spoken by the screen reader. Therefore, while the information was correct, it was more general than it could have been. If this were to be implemented in a code visualizer or interactive textbook, the exact type of data structure is likely to be known and more specific terminology could be useful. For instance, in the tree task, using terminology like parent and children could improve the understanding of the relationships. One participant got confused with the generic terminology as they were on node X and received the feedback that it was the only node that was connected via an in edge to Y. This was providing the information that Y only had one in edge (from X), but the participant interpreted it as Y was the only node connected to X. Rephrasing this information to say that X was the only parent of Y may be clearer. Additionally, extra information such as level of the tree or position in a linked list may be useful for conceptualizing the information.

Another possible area to explore would be to add information about the changes. In situations like interactive textbooks or lecture slides, the exact change would be known and could be summarized which will help a provide a user additional context. They could use that information to determine where to navigate and better incorporate the changes into their mental model of the graph. For code visualizers, certain information can be able to be automatically determined, such as nodes added or nodes removed. The usefulness of the summaries may vary depending on how much a graph has changed as too much information may be overwhelming.

7.8 SUMMARY

In this chapter, I investigated how to make dynamic graphs more accessible. I present two interaction modalities for handling the focus as users switch between the past and future versions of a graph. One mode treats the interaction of the two graphs as independent and the other mode uses the location of the focus in the current graph to determine where to place focus in the other version of the graph.

The results of the user study show that users could identify changes in the graphs with both modes, however there were differences in what they held in memory for each mode. I found that users want to have access to both modes so they can select the best mode for the task.

Chapter 8. CONTRIBUTIONS AND FUTURE WORK

In this chapter, I provide a summary of the previous chapters as well as my contributions. I then discuss the limitations of my work and highlight some areas for future work based on the themes that arose in this dissertation.

8.1 SUMMARY OF CHAPTERS

In Chapter 1, I motivated the need to improve the educational experiences of blind students in computer science. To empower users with disabilities to create their own solutions, we need to improve the educational experiences of these students. I focus on this goal in two ways: 1) understanding the barriers that blind students face in computer science education and 2) creating tools that allow blind students to access visual information that is used in computer science.

In Chapter 2, I highlight related work in the areas relating to blind programmers. This work falls in to four categories: 1) practices and challenges, 2) education, 3) programming tools, and 4) diagrams. Sections 1 and 2 provide insights into the barriers that blind programmers face and my work in chapters 3 and 4 fills in a gap in knowledge that exists in these areas. Sections 3 and 4 present artifacts that have been created and the empirical knowledge which has been gained from their evaluations, which I have used to design the artifacts I present in Chapters 5-7.

In Chapter 3, I present my work investigating *RQ1: What are the barriers that can prevent someone who is blind from studying computer science?* To answer this question, I did a survey and follow-up interviews blind students who had completed their degree in computer science or a related field. The survey provided an initial overview of the barriers that blind students face. I then did follow-up interviews with 10 of the 15 survey respondents to gain more information about the barriers they mentioned on their surveys. From these surveys and interviews, I found that barriers

included access to materials, lack of support from the faculty, and access to technology. Because inaccessible technology was common, students were hesitant to explore new technologies and would put less effort into inaccessible assignments.

In Chapter 4, to provide additional information answering *RQ1: What are the barriers that can prevent someone who is blind from studying computer science?* I follow-up on one of the barriers mentioned in the survey and interview, inaccessible technology. In this chapter, I evaluated 6 popular integrated development environments (IDEs) and code editors for basic out of the box accessibility. In this investigation, I found that all environments had problems with even basic accessibility and half were completely unusable.

In Chapter 5, I present my work investigating *RQ2: How can we provide access to graphics for people who are blind and do not know Braille?* In this chapter, I present the design and evaluation of Tactile Graphics with a Voice, which is a system that provides access to labels on tactile graphics using a smartphone application and QR codes. I evaluated the system, which had three modes with different levels of feedback and different strategies for handling multiple QR codes. In the evaluation, I found that participants could successfully answer the questions about the diagrams with just under 90% accuracy. I found that the modes worked equally well and users had varied preferences.

In Chapter 6, I present my work investigating *RQ3: How can we make it easier for blind programmers to contextualize their location and navigate through code?* To answer this question, I developed StructJumper which harnessed the indentation of the code through a modified abstract syntax tree. The evaluation of StructJumper showed that there was a trend that participants were faster using StructJumper to answer questions about the code that required navigating to locations in the code and determining the context of specific lines of code. The participants found that

StructJumper made it easier to navigate and determine the context of a line of code and decreased the cognitive load.

In Chapter 7, I presented my work investigating *RQ4: How can we make it easier for someone who is blind to understand changes in graphs?* To answer this question, I designed and developed a dynamic graph tool that had two modes to handle the focus change when moving between different versions of the graph. One mode places the user at the same location as their focus in the current graph and the other mode places focus where the user was last in that graph. In my evaluation of this tool, I found that users could identify the changes in the graph and wanted access to multiple modes for handling the focus changes. I also present the insights I gained into the design of dynamic graph tools.

8.2 Summary of Contributions

In this dissertation, I present multiple contributions to the fields of the HCI and Accessibility. The contributions are artifacts that increase access to three types of visual information found in computer science education and empirical findings regarding both the design of artifacts presenting visual information as well as findings regarding the barriers that blind computer science students face. In the sections below, I elaborate on my contributions.

8.2.1 *Artifact Contributions*

In this dissertation, I present three artifacts as part of my contribution: Tactile Graphics with a Voice, StructJumper, and the dynamic graph tool. In this section, I will describe each of the artifacts I have presented.

Tactile Graphics with a Voice was a system that had two parts: embossed graphics with QR code labels and a smartphone application for creating the labels. For the graphics, I determined

that it would take little work to integrate with the Tactile Graphics Assistant [44], which automates as much as possible the process of creating usable tactile graphics from scanned textbook images. I developed an algorithm that would automate the placement of the QR codes on the graphics and decreased the amount work needed compared to the naïve method of placing them in the same location as the original text.

The smartphone application allowed users to scan the QR codes and the results of the scan are announced and stored in a list that the user can reference at any point. As taking a picture can be a difficult task for a user, the application I developed had three different modes providing different levels of feedback and interaction styles. The application had a silent mode, verbal mode, and finger pointing mode. The verbal and finger pointing mode would provide audio cues guiding a user to scan a QR code. The verbal and finger pointing mode varied in the precise cues given based on how they handled multiple QR codes. As graphics may have many labels, and therefore many QR codes, the smartphone may have multiple QR codes visible at once. TGV had two different ways of indicating the intended QR code. The silent and verbal mode required the user to isolate a single QR code to scan. The finger pointing mode on the other hand determined which QR code to scan by selecting the QR code closest to a user's finger.

StructJumper was an Eclipse plug-in that created a modified abstract syntax tree. Instead of including every line of code in the tree, it would add the lines of code that precipitate an indentation change (e.g. the control flow statements) and all other lines of code are collapsed into leaves on the tree. This was tree is available in a separate window and provided the user the option to return to their original cursor placement in the code or update the cursor location.

The final artifact was a dynamic graph tool. This tool allowed users to navigate to the graph using the keyboard. The unique aspect of this graph exploration software was the modes to handle

focus changes when switching between versions of a graph. There were two different modes to handle the focus change. The first mode, the previous location mode, treated the two graphs as independent and therefore the state of the focus was saved when a user leaves the graph and was resumed when they returned. The other mode, the relative location mode, assumed that that the graphs were not independent so the tool should use the location of the user's focus in their current graph to determine the location of the focus in the future graph. For graphs where there is a defined structure to the graph, focus was placed on the node at the same location. Otherwise, it was placed on the same node in the other version of the graph.

I designed and developed these artifacts with a few common principles. The first is that they were all developed for mainstream technology or to provide design guidelines for mainstream technology. There were a couple of reasons for requiring all the artifacts to be designed for mainstream technology. For TGV, one of the biggest reasons I was focused on using mainstream devices is because of the cost. Prior solutions existed for people who did not know Braille to access the labels, but they required custom hardware and proprietary software (e.g. Talking Tactile Tablet or Talking Tactile Pen). These solutions can be expensive whereas a smartphone may be something that a student already owns and therefore will not require them to buy and carry extra devices.

There are also social impacts on the use of assistive technology. Using different technology can draw unwanted attention to a student's disability [72]. Additionally, if a student is using a different technology than their peers in class, then they may not be able to receive help from their peers when they run into challenges and may not be able to collaborate as easily.

When I was designing StructJumper and the dynamic graph tool, the goal was to make underlying visual structure explicit to blind users. Tools often use visual cues to show the relationships of information and these visual cues are not in any way translated into a non-visual

manner for blind users. Examples of these cues is increasing the indentation level of a line of code to show nesting and keeping the layout of a graph the same so users can find the equivalent node in multiple versions of a graph. The tools I created made explicit links between data to provide blind users access to the information that sighed users get visually.

The insights that were gained by evaluating these artifacts are discussed in the next section.

8.2.2 *Empirical Contributions*

In both my qualitative work and my design and evaluation of the artifacts described above, I found many insights about the experiences of blind students in computer science and how to best design for these students. I describe these findings below.

Accessibility barriers decrease the motivation of students to learn and explore in computer science. In my interviews with blind graduates who had completed their degrees in computer science or a related field, I found that inaccessible materials had a large effect on a students' motivation. I found that because students had encountered so many inaccessible development environments in the past, and the ones that were accessible were difficult to learn, they were hesitant to explore new environments or features within an environment. I also found that when students had assignments that required a sighted student to tell them if it worked, they were not excited by the assignment and did not put in as much effort to complete the assignment and therefore did not learn the related concepts as well.

Blind users need access to multiple modes. In the evaluations of both Tactile Graphics with a Voice and the Dynamic Graphs prototype, I saw interest in having access to multiple modes. Users preferences may change as they gain expertise, when they are in specific situations, or based on the tasks they are trying to complete. I saw desire for multiple modes throughout the work that I present in this dissertation. In the formative work for TGV, I found that users often want to

decrease the amount of feedback they hear in situations where they may need to be quiet or to be able to listen to something else as well. In the TGV evaluation, I saw that half the participants had the mode that was initially their least favorite become their favorite mode after six sessions, showing the effect expertise has on preference. Finally, in the dynamic graph study, I saw that every participant indicated that, even if they preferred one mode to the other, they wanted access to both modes so they could decide based on the task. Therefore, it is important to provide access to multiple modes so that the user can select the best mode for the task and situation.

Design tools so blind users can easily determine the context. As screen readers are limited to providing information about one location at a time, it is important to allow a user to know the context of their current location. In the evaluation StructJumper, one of the benefits that the participants saw was that they could quickly check contextual information such as what conditionals a statement was inside. In the dynamic graphs tool evaluation, the importance of maintaining context was raised in the post-session interviews by many participants. Participants were frustrated when the relative location mode would lose part of their context when switching graphs. In the previous location mode, they would often keep the state of the focus of other graph in their memory so that when they switched to that graph, they would know the context and would not have to navigate around to get it.

Decrease the amount of information blind users need to hold in memory. As screen readers only allow blind users to access one piece of information at a time, it can require them to hold other information in memory to understand the context and complete tasks. Designers need to be cognizant of the extra information that blind users will need to hold in memory and should design tools that decrease the amount of information that needs to be held in memory. I tried to decrease what needed to be remembered with all the artifacts that I presented. Tactile Graphics

with a Voice made sure to not only read aloud the result of the QR code scan, it also stored it in a list that the user could go back and reference so they did not have to remember every label they had already scanned. StructJumper allowed a blind user to only look at relevant lines of code when determining which conditionals a statement is nested under. In contrast, without StructJumper, a user will need to remember the conditionals for a longer period of time as they go through all the lines of code and extra conditionals may be held in memory until a until the user determined they did not apply to the statement. Finally, in the evaluation of the dynamic graph tool, I saw that as the two modes required remembering different information, participants varied in their opinions of which mode decreased the memory requirements based on what information they felt was easier to hold in memory. While the work that I present has presented all the information via text, the use of other types of audio, such as spearcons and earcons, may be able to further reduce the memory requirements.

8.3 Support of Thesis Statement

In this section, I will summarize the evidence presented in my dissertation which supports my thesis statement: *Blind students face many barriers in their education, including access to technology and visual information, which can decrease their motivation to study computer science and other technical fields. To improve the access to visual information, we can use mainstream technologies to augment its accessibility, which can support: 1) Increased understanding of the structure and relationships of information and 2) Decreased cognitive load.*

In Chapters 3 and 4, I provide support for the first part of my thesis statement: "Blind students face many barriers in their education, including access to technology and visual information, which can decrease their motivation to study computer science and other technical fields." These chapters highlight some of the barriers that blind students faced while studying

computer science and provide multiple examples of how it decreased the students' motivation to study computer science.

The second half of my thesis statement, "To improve the access to visual information, we can use mainstream technologies to augment its accessibility, which can support: 1) Increased understanding of the structure and relationships of information and 2) Decreased cognitive load." is supported in Chapters 5-7. Each of the artifacts presented in these chapters is created on mainstream technology or with mainstream technologies in mind.

The design of the tools presented in Chapters 5-7 worked to support making the relationships and/or structure of information clear to blind users. Tactile Graphics with a Voice (TGV) does this based on the label placement. One of the issues with Braille labels is that their length, which can span the entire page, can make it harder to determine what the label is referring to. QR codes are more compact, which allows us to fit more labels on the page and will hopefully allow a user to more easily determine what part of the image the label corresponds with. StructJumper sought to make the nesting relationships of lines of code clear with a tree structure. In my evaluation, I found that the structure provided allowed users a better understanding of the relationship of lines of code. Finally, the dynamic graph tool provided ways for users to compare and understand the changes that had occurred in a graph. In my evaluation, I found that both modes provided users with the ability to determine the changes in a graph. The relative location mode, which connects the same location in both graphs, allowed users to quickly see if a change had occurred in that location.

In my evaluations of the tools I created, there was support that StructJumper and the dynamic graph tool decreased the cognitive load for the users. I saw that StructJumper was able to decrease user's cognitive load in many ways. StructJumper required user to remember less

information when determining what lines of code a statement is nested under. Additionally, it provided additional cues to the programmers about the relationships of lines of code that were easy to access, so that a user did not have to try and remember or deduce all the information and could quickly look it up. Finally, StructJumper allowed some users to focus less on their navigation strategy, allowing them to focus more on the code.

The dynamic graph tool was not compared to a control, rather two modes were tested to understand their effects. Both modes were found to be useful for participants. However, most participants seemed to prefer one mode or the other based on which they found easier to use. The two modes required remembering different information to complete the tasks and therefore when using this tool outside of the evaluation, users could choose the mode that they found decreased their cognitive load for the specific task.

8.4 LIMITATIONS

In this section, I will highlight some of the limitations for the studies I present in this dissertation. For all the projects presented, there were small sample sizes. With larger sample sizes I would be able to be more certain about the conclusions I have drawn in the studies presented in this dissertation.

In Chapter 3, I discussed the barriers that were faced by blind students in computer science and related fields. However, I only talked with people who had successfully completed their degree. I believe that the issues faced by those who were successful were also issues faced by those who did not succeed. But there may be issues faced by the students who did not succeed that were not raised in the interviews.

In Chapter 4, I did an evaluation of the accessibility of IDEs and code editors. However, the evaluation was done by a novice screen reader user with NVDA. While NVDA is gaining in

popularity, JAWS is still the most popular screen reader [86] and they do not work identically. So, for users who are using JAWS or are expert screen reader users, the experience may not be the same and some of the challenges may not exist. Additionally, screen readers can augment the accessibility of software through add-ons. For instance, some users have developed an add-on for Visual Studio[18] which fixes some of the issues I mentioned in Chapter 4, Section 2.2. Finally, I only evaluated IDEs and editors on the Windows platform. The accessibility of IDEs and editors on other platforms, such as Mac or Linux, may be more or less than those of on Windows.

In Chapter 5, I evaluated Tactile Graphics with a Voice with a limited set of graphics. These graphics were created with an embosser and had no color. I took advantage of the lack of color on the graphics in the design of the algorithm to recognize the finger and instead of looking just for skin tone pixels, it looked for any colored pixels, which was beneficial as the finger is still recognized when a user is wearing nail polish. However, this limits the types of graphics that the finger pointing mode will work for as there are embossers that use both ink and embossing, adding colors to the diagrams. Further work would be needed to examine the tradeoffs of being able to recognize the finger when wearing nail polish versus not being able to use colored graphics.

In Chapter 6, I evaluated StructJumper by having participants navigate around code bases both with and without using StructJumper. One of the limitations of this work is that it only considers navigation within unfamiliar code. As programmers will spend much of their time navigating through code that they have been working on and is familiar to them, this study does not look at one of the main use cases. I do not yet know if the benefits I saw in this study will carry over to navigation in familiar code.

[18] https://github.com/mohammad-suliman/visualstudioaddon

In Chapter 7, there were two main limitations to the study. All but one of the participants used the self-voicing mode of the dynamic graph tool prototype instead of their own screen readers. This meant that they were not using all their settings that they have optimized. This can include voice, speed, use of a Braille display and other similar preferences. Expert screen reader users can browse at up to 500 words per minute [16], which is significantly faster than the TTS engine I used in the self-voicing mode. For some users, they found that using the slower voice meant that it was harder for them to remember as much information as there were longer pauses between pieces of information. Additionally, multiple users misheard or requested clarification on the node names due to the voice being different.

The other main limitation of the study was that I limited the tasks to small changes. When more of a graph is changing, this may change how well a user is able to understand the relationship between the two graphs and it may bring changes in their preferences for mode and the effects of the modes.

8.5 Future Work

In my investigation into the barriers faced by blind students in computer science, I found many problems that are still unresolved. While I have sought to address some of the known problems, there are still many more that need additional research. I highlight a few areas below that would benefit from additional research.

8.5.1 *Creation of Accessible Materials*

As we saw in Chapters 3 and 4, one of the largest problems that blind students face is lack of access to the materials and technology that are necessary to succeed in the computer science. While further research is needed to improve the accessible formats of these materials, there are currently

existing accessible formats that are not used. Therefore, one area that needs further investigation is how to increase the creation of accessible materials. I think that are two areas that should be further explored to increase production of accessible materials: automation and education.

8.5.1.1 *Automation*

One of the current problems with the creation of accessible materials is that they are time consuming to create. There has been some work in automating the creation of accessible images either through automating the conversion to tactile graphics [44] or by automatically generating alt text via computer vision [87]. However, the automatic generation of these resources is still at a point where additional work would be needed. The tactile graphics assistant works best when working with an entire book and still requires an average of 10 minutes of work per image [44] and automatically generated alt text is frequently wrong.

With the automation of alt text, researchers have found that users tend to over trust the generated text and will make up stories to make the alt text make sense in reference to the caption when the alt text is wrong [55]. In education, having wrong alt text can be a huge detriment to the student as they may draw wrong conclusions about the material. However, the automatic generation of alt text may still be useful when combined with additional human oversight as it may decrease the total amount of work needed to make images accessible. For each of these processes, there is still plenty of room for improvement of the automated process.

Additionally, most of the automation has focused on taking an existing image and creating an accessible version of it from the visual image. Another approach that may be fruitful in the future is to design tools that will create an accessible version of the image at the same time the image is created.

8.5.1.2 *Education*

Another area that could lead to the production of more accessible materials is education. Currently many developers do not know how to create accessible software. This is an area in which more companies need trained developers and therefore they have created the TeachAccess initiative[19]. One of the goals of this initiative is to increase the number of classes teaching accessibility and they do this by providing some basic curriculum materials that educators can use.

Additionally, researchers have looked at the best practices for teaching accessibility [66]. However, the researchers also found that only 3 of the 18 faculty that they interview regarding their accessibility teaching felt that accessibility was integrated throughout the curriculum. For more developers to consider accessibility when building their next tools, it is important for more schools to integrate accessibility throughout their curriculum so that more students know the material well. Additional research on how we can encourage educators to teach accessibility is needed.

8.5.2 *Creating Visuals*

In this dissertation, I presented work on making visual information accessible to blind students. However, beyond accessing this information, they often need to create it as well. This is a relatively unexplored area that would be a good direction for future work. I will discuss two specific areas where the need to create visuals is common in computer science education.

8.5.2.1 *Interfaces and Layouts*

In my survey and interviews with blind graduates of computer science programs, I found that creating interfaces and layouts for websites or posters was difficult for them. Some students were

[19] http://teachaccess.org/s

excused from creating them or would use a sighted peer to help them evaluate their layouts. One student mentioned that they would use they touch screen device to explore the interface so that they could gain a spatial understanding of the layout.

One of the problems is that many GUI interface builders rely on the drag and drop interface, which is difficult for a blind user to use. At this point, the one common way for a blind student to make an interface is through explicitly coding where interface elements go. Researchers have investigated improving scripting languages to make it easier to code the placement of the interface elements [32]. But requiring blind developers to explicitly code all their interfaces is still more work. Additional research into how to support interface/layout design for blind students is still needed.

8.5.2.2 *Graphs*

The creation of graphs can be difficult for blind students. In my survey and interviews, I found that many students were excused from this task and would just use a notation or text to indicate what the graph should look like. When graph creation occurred in group projects, the blind student often could not participate in that part of the project and would do more work in another area instead.

There has been some research focused on this area, such as Balik et al. who used a grid like system to allow blind users to create graphs that are visually readable [10]. The system was designed so it could be used by both blind and sighted users. However, collaborative graph creation software for blind and sighted users has not been studied much and further research in this area is needed.

8.6 CONCLUDING REMARKS

The overarching goal of my dissertation has been to support blind students in their pursuit of learning computer science. Through the initial work I presented in Chapters 3 and 4, I have sought to identify the barriers that we need to remove to increase the number of blind students in computer science. Additionally, I sought to provide access to information that is not currently available.

However, in my evaluations, I have realized that we need to aim to not just provide access to the materials, but we need to do so in a way that decreases the amount of information that we require blind students to hold in memory. In the evaluations of the tools I created, the amount information that blind students needed to hold in memory played a large role in determining the students experience and opinion of the tool. In the future, I aim to continue reducing the amount of information that blind students must be able to hold in their memory to successfully access information.

BIBLIOGRAPHY

1. About | The Alliance for Access to Computing Careers. Retrieved April 21, 2016 from http://www.washington.edu/accesscomputing/about
2. Khaled Albusays and Stephanie Ludi. (2016). Eliciting Programming Challenges Faced by Developers with Visual Impairments: Exploratory Study. *Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering - CHASE '16*, ACM Press, 82–85. http://doi.org/10.1145/2897586.2897616
3. Frances K Aldrich and Linda Sheppard. (2001). Tactile graphics in school education: perspectives from pupils. *The British Journal of Visual Impairment*, *19*(2), 69–73.
4. Ameer Armaly and Collin McMillan. (2016). An Empirical Study of Blindness and Program Comprehension. *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16*, ACM Press, 683–685. http://doi.org/10.1145/2889160.2891041
5. Chieko Asakawa and Takashi Itoh. (1998). User Interface of a Home Page Reader. *Proceedings of the Third International ACM Conference on Assistive Technologies - Assets '98*, ACM Press, 149–156. http://doi.org/10.1145/274497.274526
6. Catherine M. Baker, Lauren R. Milne, Ryan Drapeau, Jeffrey Scofield, Cynthia L. Bennett, and Richard E. Ladner. (2016). Tactile Graphics with a Voice. *ACM Transactions on Accessible Computing*, *8*(1), 1–22. http://doi.org/10.1145/2854005
7. Catherine M. Baker, Lauren R. Milne, and Richard E. Ladner. (2015). StructJumper: A Tool to Help Blind Programmers Navigate and Understand the Structure of Code. *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems - CHI '15*, ACM Press, 3043–3052. http://doi.org/10.1145/2702123.2702589
8. Catherine M. Baker, Lauren R. Milne, Jeffrey Scofield, Cynthia L. Bennett, and Richard E. Ladner. (2014). Tactile Graphics with a Voice: Using QR Codes to Access Text in Tactile Graphics. *Proceedings of the 16th International ACM SIGACCESS Conference on Computers & Accessibility - ASSETS '14*, 75–82. http://doi.org/10.1145/2661334.2661366
9. Catherine M. Baker, Lauren R. Milne, Jeffrey Scofield, Cynthia L. Bennett, and Richard E. Ladner. (2014). Tactile Graphics with a Voice Demonstration. *Proceedings of the 16th International ACM SIGACCESS Conference on Computers & Accessibility - ASSETS '14*, ACM Press, 321–322. http://doi.org/10.1145/2661334.2661349
10. Suzanne P. Balik, Sean P. Mealin, Matthias F. Stallmann, and Robert D. Rodman. (2013). GSK: Universally Accessible Graph SKetching. *Proceeding of the 44th ACM Technical Symposium on Computer Science Education - SIGCSE '13*, ACM Press, 221. http://doi.org/10.1145/2445196.2445266
11. Suzanne P. Balik, Sean P. Mealin, Matthias F. Stallmann, Robert D. Rodman, Michelle L. Glatz, and Veronica J. Sigler. (2014). Including Blind People in Computing Through Access to Graphs. *Proceedings of the 16th International ACM SIGACCESS Conference on Computers & Accessibility - ASSETS '14*, ACM Press, 91–98. http://doi.org/10.1145/2661334.2661364
12. Jeffrey P. Bigham, Maxwell B. Aller, Jeremy T. Brudvik, Jessica O. Leung, Lindsay A. Yazzolino, and Richard E. Ladner. (2008). Inspiring Blind High School Students to Pursue Computer Science with Instant Messaging Chatbots. *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education - SIGCSE '08*, ACM, 449. http://doi.org/10.1145/1352322.1352287

13. Jeffrey P. Bigham, Chandrika Jayant, Andrew Miller, Brandyn White, and Tom Yeh. (2010). VizWiz::LocateIt - Enabling Blind People to Locate Objects in Their Environment. *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops, CVPRW '10*, 65–72. http://doi.org/10.1109/CVPRW.2010.5543821
14. Mahesh Kumar Biradar. (2015). Popularity of Programming Languages | Information SuperHighway on WordPress.com. Retrieved May 22, 2016 from https://maheshbiradar.wordpress.com/2015/07/28/popularity-of-programming-language/
15. P. Blenkhorn and D.G. Evans. (1998). Using speech and touch to enable blind people to access schematic diagrams. *Journal of Network and Computer Applications*, *21*(1), 17–29. http://doi.org/10.1006/jnca.1998.0060
16. Yevgen Borodin, Jeffrey P. Bigham, Glenn Dausch, and I. V. Ramakrishnan. (2010). More than Meets the Eye: A Survey of Screen-Reader Browsing Strategies. *Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A) - W4A '10*, ACM Press, 1. http://doi.org/10.1145/1805986.1806005
17. Stacy M. Branham and Shaun K. Kane. (2015). The Invisible Work of Accessibility: How Blind Employees Manage Accessibility in Mixed-Ability Workplaces. *Proceedings of the 17th International ACM SIGACCESS Conference on Computers & Accessibility - ASSETS '15*, ACM, 163–171. http://doi.org/10.1145/2700648.2809864
18. Ivan Burazin. (2015). Most Popular Desktop IDEs & Code Editors. Retrieved May 22, 2016 from https://blog.codeanywhere.com/most-popular-ides-code-editors/
19. Sheryl Burgstahler. (2011). Universal Design. *ACM Transactions on Computing Education*, *11*(3), 1–17. http://doi.org/10.1145/2037276.2037283
20. Sheryl Burgstahler and Richard Ladner. (2006). AccessComputing: From Research to Practice | The Alliance for Access to Computing Careers. Retrieved May 3, 2016 from http://www.washington.edu/accesscomputing/resources/accesscomputing-research-practice
21. Matt Calder, Robert F. Cohen, Jessica Lanzoni, Neal Landry, Joelle Skaff, Matt Calder, Robert F. Cohen, Jessica Lanzoni, Neal Landry, and Joelle Skaff. (2007). Teaching Data Structures to Students who are Blind. *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education - ITiCSE '07*, ACM Press, 87. http://doi.org/10.1145/1268784.1268811
22. Matt Calder, Robert F. Cohen, Jessica Lanzoni, and Yun Xu. (2006). PLUMB:: An Interface for Users Who Are Blind to Display, Create, and Modify Graphs. *Proceedings of the 8th International ACM SIGACCESS Conference on Computers and Accessibility - ASSETS '06*, ACM Press, 263–264. http://doi.org/10.1145/1168987.1169046
23. Ben Caldwell, Michael Cooper, Loretta Guarino Reid, and Gregg Vanderheiden. (2008). Web Content Accessibility Guidelines (WCAG) 2.0. Retrieved August 10, 2017 from https://www.w3.org/TR/WCAG20/
24. Robert F. Cohen, Arthur Meacham, and Joelle Skaff. (2006). Teaching Graphs to Visually Impaired Students Using an Active Auditory Interface. *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education - SIGCSE '06*, ACM Press, 279. http://doi.org/10.1145/1121341.1121428
25. Richard Connelly. (2010). Lessons and Tools From Teaching a Blind Student. *Journal of Computing Sciences in Colleges*, *25*(6), 34–39. http://dl.acm.org/citation.cfm?id=1791129.1791137

26. David Galles. (2011). Data Structure Visualization. Retrieved July 30, 2017 from http://cs.usfca.edu/~galles/visualization/Algorithms.html
27. Parham Doustdar. (2016). The Tools of a Blind Programmer – Parham Doustdar's Blog. Retrieved April 21, 2016 from https://www.parhamdoustdar.com/2016/04/03/tools-of-blind-programmer/
28. EdSharp. Retrieved August 11, 2017 from https://github.com/EmpowermentZone/EdSharp
29. Ahmed Elgammal, Crystal Muang, and Dunxu Hu. (2009). Skin Detection - a Short Tutorial. *Encyclopedia of Biometrics*, 1–10. http://pdf.aminer.org/000/312/166/finding_facial_features_using_an_hls_colour_space.pdf
30. Leo Ferres, Gitte Lindgaard, and Livia Sumegi. (2010). Evaluating a Tool for Improving Accessibility to Charts and Graphs. *Proceedings of the 12th International ACM SIGACCESS Conference on Computers and Accessibility - ASSETS '10*, ACM Press, 83. http://doi.org/10.1145/1878803.1878820
31. Joan M. Francioni and Ann C. Smith. (2002). Computer Science Accessibility for Students with Visual Disabilities. *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education - SIGCSE '02*, ACM Press, 91. http://doi.org/10.1145/563340.563372
32. Kenneth G. Franqueiro and Robert M. Siegfried. (2006). Designing a Scripting Language to Help the Blind Program Visually. *Proceedings of the 8th International ACM SIGACCESS Conference on Computers and Accessibility - Assets '06*, ACM Press, 241. http://doi.org/10.1145/1168987.1169035
33. Nicholas A. Giudice, Hari Prasath Palani, Eric Brenner, and Kevin M. Kramer. (2012). Learning Non-Visual Graphical Information using a Touch-Based Vibro-Audio Interface. *Proceedings of the 14th International ACM SIGACCESS Conference on Computers and Accessibility - ASSETS '12*, ACM Press, 103. http://doi.org/10.1145/2384916.2384935
34. Daniel Goransson. (2011). New VoiceOver Features in iOS 5. Retrieved August 11, 2017 from http://axslab.com/articles/new-voiceover-features-in-ios5
35. Berchie Woods Gordon-Holliday, L.E. Yunker, G. Vannatta, and F.J. Crosswhite. (1999). *Advanced Mathematical Concepts: Precalculus with Appplications*. Glencoe/McGraw-Hill.
36. Philip J. Guo. (2013). Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education. *Proceeding of the 44th ACM Technical Symposium on Computer Science Education - SIGCSE '13*, ACM Press, 579. http://doi.org/10.1145/2445196.2445368
37. Stephen J. Hartley. (1994). Animating Operating Systems Algorithms with XTANGO. *Proceedings of the Twenty-fifth SIGCSE Symposium on Computer Science Education - SIGCSE '94*, ACM Press, 344–348. http://doi.org/10.1145/191029.191164
38. T. Dean Hendrix, James H. Cross, and Larry A. Barowski. (2004). An Extensible Framework for Providing Dynamic Data Structure Visualizations in a Lightweight IDE. *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education - SIGCSE '04*, ACM Press, 387. http://doi.org/10.1145/971300.971433
39. Bill Holton. (2013). Voiceye: A Breakthrough in Document Access. *AFB AccessWorld Magazine*. Retrieved August 11, 2017 from http://www.afb.org/afbpress/pubnew.asp?DocID=aw140605

40. Ayanna M. Howard, Chung Hyuk Park, and Sekou Remy. (2012). Using Haptic and Auditory Interaction Tools to Engage Students with Visual Impairments in Robot Programming Activities. *IEEE Transactions on Learning Technologies*, *5*(1), 87–95. http://doi.org/10.1109/TLT.2011.28
41. Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. (2002). A Meta-Study of Software Visualization Effectiveness. *Journal of Visual Languages & Computing*, *13*(3), 259–290. http://doi.org/10.1006/jvlc.2002.0237
42. Essi Isohanni and Hannu-Matti Järvinen. (2014). Are Visualization Tools Used in Programming Education? By Whom, How, Why, and Why Not? *Proceedings of the 14th Koli Calling International Conference on Computing Education Research - Koli Calling '14*, ACM Press, 35–40. http://doi.org/10.1145/2674683.2674688
43. Chandrika Jayant, Hanjie Ji, Samuel White, and Jeffrey P. Bigham. (2011). Supporting Blind Photography. *The Proceedings of the 13th International ACM SIGACCESS Conference on Computers and Accessibility - ASSETS '11*, 203–210. http://doi.org/10.1145/2049536.2049573
44. Chandrika Jayant, Matt Renzelmann, Dana Wen, Satria Krisnandi, Richard Ladner, and Dan Comden. (2007). Automated Tactile Graphics Translation: In the Field. *Proceedings of the 9th International ACM SIGACCESS Conference on Computers and Accessibility - Assets '07*, 75. http://doi.org/10.1145/1296843.1296858
45. jGRASP Accessibility Statement. Retrieved April 24, 2017 from http://www.jgrasp.org/accessibility.html
46. Maggie Johnson. (2015). The Computer Science Pipeline and Diversity: Part 1 - How did we get here? Retrieved May 7, 2016 from http://googleresearch.blogspot.com/2015/07/the-computer-science-pipeline-and.html
47. Shaun K. Kane and Jeffrey P. Bigham. (2014). Tracking @stemxcomet: Teaching Programming to Blind Students via 3D Printing, Crisis Management, and Twitter. *Proceedings of the 45th ACM Technical Symposium on Computer Science Education - SIGCSE '14*, ACM Press, 247–252. http://doi.org/10.1145/2538862.2538975
48. Shaun K. Kane, Brian Frey, and Jacob O. Wobbrock. (2013). Access Lens: A Gesture-Based Screen Reader for Real-World Documents. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '13*, 347–350. http://doi.org/10.1145/2470654.2470704
49. Andrea R. Kennel. (1996). Audiograf: A Diagram-Reader for the Blind. *Proceedings of the Second Annual ACM conference on Assistive Technologies - ASSETS '96*, ACM Press, 51–56. http://doi.org/10.1145/228347.228357
50. Richard E. Ladner. (2015). Design for User Empowerment. *interactions 22*, 24–29. http://doi.org/10.1145/2723869
51. Steven Landau, Steven Holborow, and Jane Erin. (2004). The Use of the Talking Tactile Tablet for Delivery of Standardized Tests. *Annual International Technology and Persons with Disabilities Conference - CSUN '04*. http://www.csun.edu/~hfdss006/conf/2004/proceedings/324.htm
52. Steven Landau and Joshua Miele. (2010). Talking Tactile Apps for the Pulse Pen: STEM Binder. Retrieved August 11, 2017 from https://docs.google.com/presentation/d/1Ylscpk6QKX7Y5WCW3CFnhZDJJlL4X5ki8gHdcJvBp70/present#slide=id.i0
53. Stephanie Ludi. (2013). Robotics Programming Tools for Blind Students. *28th Annual*

International Technology and Persons with Disabilities Conference Scientific/Research Proceedings, California State University, Northridge. http://scholarworks.calstate.edu/handle/10211.3/121968

54. Stephanie Ludi, Jamie Simpson, and Wil Merchant. (2016). Exploration of the Use of Auditory Cues in Code Comprehension and Navigation for Individuals with Visual Impairments in a Visual Programming Environment. *Proceedings of the 18th International ACM SIGACCESS Conference on Computers and Accessibility - ASSETS '16*, ACM Press, 279–280. http://doi.org/10.1145/2982142.2982206
55. Haley MacLeod, Cynthia L. Bennett, Meredith Ringel Morris, and Edward Cutrell. (2017). Understanding Blind People's Experiences with Computer-Generated Captions of Social Media Images. *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems - CHI '17*, ACM Press, 5988–5999. http://doi.org/10.1145/3025453.3025814
56. Collin McMillan and Amanda Rodda-Tyler. (2016). Collaborative Software Engineering Education Between College Seniors and Blind High School Students. *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16*, ACM Press, 360–363. http://doi.org/10.1145/2889160.2889188
57. Sean Mealin and E Murphy-Hill. (2012). An Exploratory Study of Blind Software Developers. *Proceedings of 2012 IEEE Symposium on Visual Languages and Human-Centric Computing - VL/HCC '12*. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6344485
58. Suranga Nanayakkara, Roy Shilkrot, Kian Peen Yeo, and Pattie Maes. (2013). EyeRing: A Finger-Worn Input Device for Seamless Interactions with our Surroundings. *Proceedings of the 4th Augmented Human International Conference on - AH '13*, 13–20. http://doi.org/10.1145/2459236.2459240
59. National Federation of the Blind Jernigan Institute. (2009). *The Braille Literacy Crisis in America: Facing the Truth, Reversing the Trend, Empowering the Blind*. Retrieved August 11, 2017 from https://nfb.org/images/nfb/documents/pdf/braille_literacy_report_web.pdf
60. National Center for Science and Engineering Statistics National Science Foundation. (2017). *Data Tables: Women, Minorities, and Persons with Disabilities in Science and Engineering*. Retrieved May 23, 2016 from http://www.nsf.gov/statistics/2015/nsf15311/tables.cfm
61. Nemeth Braille. Retrieved August 11, 2017 from http://www.braillebug.org/nemeth_braille.asp
62. Nomensa. (2006). *United Nations Global Audit of Web Accessibility*. Retrieved May 1, 2017 from http://www.un.org/esa/socdev/enable/documents/fnomensarep.pdf
63. Son Lam Phung, Abdesselam Bouzerdoum, and Douglas Chai. (2003). Skin Segmentation Using Color and Edge Information. *Proceedings - 7th International Symposium on Signal Processing and Its Applications, ISSPA 2003*, *1*(July), 525–528. http://doi.org/10.1109/ISSPA.2003.1224755
64. Son Lam Phung, Abdesselam Bouzerdoum, and Douglas Chai. (2005). Skin Segmentation Using Color Pixel Classification: Analysis and Comparison. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *27*(1), 148–154. http://doi.org/10.1109/TPAMI.2005.17
65. John R. Porter and Julie A. Kientz. (2013). An Empirical Study of Issues and Barriers to

Mainstream Video Game Accessibility. *Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility - ASSETS '13*, 1–8. http://doi.org/10.1145/2513383.2513444

66. Cynthia Putnam, Maria Dahman, Emma Rose, Jinghui Cheng, and Glenn Bradford. (2016). Best Practices for Teaching Accessibility in University Classrooms. *ACM Transactions on Accessible Computing*, *8*(4), 1–26. http://doi.org/10.1145/2831424
67. (2011). QR Code Minimum Size. Retrieved August 11, 2017 from http://www.qrstuff.com/blog/2011/11/23/qr-code-minimum-size
68. T. V. Raman. (1996). Emacspeak---Direct Speech Access. *Proceedings of the Second Annual ACM Conference on Assistive Technologies - Assets '96*, ACM Press, 32–36. http://doi.org/10.1145/228347.228354
69. Audrey C. Rule, Greg P. Stefanich, Robert M. Boody, and Belinda Peiffer. (2011). Impact of Adaptive Materials on Teachers and their Students with Visual Impairments in Secondary Science and Mathematics Classes. *International Journal of Science Education*, *33*(January 2015), 865–887. http://doi.org/10.1080/09500693.2010.506619
70. Jaime Sánchez and Fernando Aguayo. (2005). Blind Learners Programming Through Audio. *Extended Abstracts on Human Factors in Computing Systems - CHI '05*, 1769–1772. http://dl.acm.org/citation.cfm?id=1057018
71. Linda Sheppard and Frances K Aldrich. (2001). Tactile graphics in school education: perspectives from teachers. *The British Journal of Visual Impairment, 2001*, (19:3), 93–97. http://doi.org/10.1177/026461960101900303
72. Kristen Shinohara and Jacob O. Wobbrock. (2011). In the Shadow of Misperception: Assistive Technology Use and Social Interactions. *Proceedings of the 2011 Annual Conference on Human Factors in Computing Systems - CHI '11*, ACM Press, 705. http://doi.org/10.1145/1978942.1979044
73. Size and Spacing of Braille Characters. Retrieved July 25, 2017 from http://www.brailleauthority.org/sizespacingofbraille/sizespacingofbraille.pdf
74. Ann C. Smith, Joan M. Francioni, Mohd Anwar, Justin S. Cook, Asif Hossain, and M. Fayezur Rahman. (2003). Nonvisual Tool for Navigating Hierarchical Structures. *Proceedings of the 6th International ACM SIGACCESS Conference on Computers and Accessibility - ASSETS '04*, 133. http://doi.org/10.1145/1029014.1028654
75. Ann C. Smith, Joan M. Francioni, and Sam D. Matzek. (2000). A Java Programming Tool for Students with Visual Disabilities. *Proceedings of the Fourth International ACM Conference on Assistive Technologies - ASSETS '00*, ACM Press, 142–148. http://doi.org/10.1145/354324.354356
76. (2017). Stack Overflow Developer Survey 2017. Retrieved July 18, 2017 from https://insights.stackoverflow.com/survey/2017#developer-profile
77. Andreas Stefik, Roger Alexander, Robert Patterson, and Jonathan Brown. (2007). WAD: A Feasibility study using the Wicked Audio Debugger. *15th IEEE International Conference on Program Comprehension (ICPC '07)*, IEEE, 69–80. http://doi.org/10.1109/ICPC.2007.42
78. Andreas Stefik, Christopher Hundhausen, and Robert Patterson. (2011). An empirical investigation into the design of auditory cues to enhance computer program comprehension. *International Journal of Human-Computer Studies*, *69*(12), 820–838. http://doi.org/10.1016/j.ijhcs.2011.07.002
79. Andreas M. Stefik, Christopher Hundhausen, and Derrick Smith. (2011). On the Design of

an Educational Infrastructure for the Blind and Visually Impaired in Computer Science. *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education - SIGCSE '11*, ACM Press, 571. http://doi.org/10.1145/1953163.1953323

80. Ender Tekin and James M. Coughlan. (2010). A Mobile Phone Application Enabling Visually Impaired Users to Find and Read Product Barcodes. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *6180 LNCS*(PART 2), 290–295. http://doi.org/10.1007/978-3-642-14100-3_43
81. TOPIDE Top Integrated Development Environment index. Retrieved May 23, 2016 from https://pypl.github.io/IDE.html
82. Marynel Vázquez and Aaron Steinfeld. (2012). Helping Visually Impaired Users Properly Aim a Camera. *Proceedings of the 14th International ACM SIGACCESS Conference on Computers and Accessibility*, 95–102. http://doi.org/10.1145/2384916.2384934
83. Paul Vickers and James L Alty. (2002). When bugs sing. *Interacting with Computers*, *14*(6), 793–819. http://doi.org/10.1016/S0953-5438(02)00026-7
84. Bruce N. Walker and Lisa M. Mauney. (2010). Universal Design of Auditory Graphs: A Comparison of Sonification Mappings for Visually Impaired and Sighted Listeners. *ACM Transactions on Accessible Computing*, *2*(3), 1–16. http://doi.org/10.1145/1714458.1714459
85. WebAIM: Screen Reader User Survey #4 Results. Retrieved August 11, 2017 from http://webaim.org/projects/screenreadersurvey4/
86. WebAIM: Screen Reader User Survey #6 Results. Retrieved June 19, 2017 from http://webaim.org/projects/screenreadersurvey6/
87. Shaomei Wu, Jeffrey Wieland, Omid Farivar, and Julie Schiller. (2017). Automatic Alt-text: Computer-generated Image Descriptions for Blind Users on a Social Network Service. *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing - CSCW '17*, ACM Press, 1180–1192. http://doi.org/10.1145/2998181.2998364

APPENDIX A

Educational Experiences of Blind Programmers Survey and Interview Script

Survey
Criteria: Undergraduate degree in computer science or a related field and used a screen reader while completing that degree.
The following 4 questions refer to your undergraduate classes in computer science or related fields:

1. What aspects of these classes were not fully accessible and what accommodations were made when you came across these aspects?
2. Were you ever provided an alternative assignment? If so, what are some examples of alternative assignments you were given and what it replaced.
3. Were you ever excused completely from an assignment? If so, what are some examples of assignments you were excused from.
4. How many of your undergraduate computer science courses had materials or assignments that were not fully accessible?
    a. None had inaccessible portions
    b. A few had inaccessible portions
    c. About half had inaccessible portions
    d. Most had inaccessible portions
    e. All had inaccessible portions

The following 4 questions refer to any learning related to computer science that was outside of formal classes (at any point in time).

5. Which of these have you used to learn a computer science topic?
    a. Have not taught myself any topics
    b. Books
    c. Online tutorials
    d. Online courses
    e. Forums or Q&A sites (e.g. Stack Overflow)
    f. Blogs
    g. Other people
    h. Other:
6. Which of these have you used (or tried to use) and come across materials that were not fully accessible?
    a. Have not taught myself any topics
    b. None of the resources were inaccessible
    c. Books
    d. Online tutorials
    e. Online courses
    f. Forums or Q&A sites (e.g. Stack Overflow)
    g. Blogs
    h. Other people
    i. Other:

7. What were some examples of how the materials were not fully accessible?
8. What did you do when you came across something that was not fully accessible?

The following 9 questions relate to your computer science background

9. What was your introduction to computer science or programming?
10. Did you take any computer science classes before you started your undergrad (e.g. AP Computer Science)? If so, which ones?
11. How much programming outside of classes had you done before your first undergrad computer science class?
12. What programming language were you first introduced to?
13. What is your preferred programming language?
14. What IDE or text editor were you first introduced to for programming?
15. What is your preferred set-up (IDE, screen reader, braille device, accessibility features, etc.)? If you have multiple depending on the language or project, please list them all and the scenario where you would use each set-up.
16. What parts of the programming languages or IDEs that you commonly use are not fully accessible?
17. What was the largest barrier you faced in learning computer science?

The following 6 questions are demographic questions.

18. What is your age?
19. What is your gender?
20. What is your highest level of education?
21. How many years of computer science experience do you have?
22. What country you did get your undergraduate degree in?
23. What year did you graduate with your undergraduate degree?

Follow-up Interview Protocol – The interview was based primarily on the responses in the survey therefore there was not a single protocol. Users were asked to expand on the answers in the survey and as it was a semi-structured interview, additional topics were explored. In general, the topics explored were:

- Computing Background
- Accessibility of classes
    - Materials
    - Assignments
    - Specific classes that were more difficult due to accessibility
    - Group projects
    - Faculty support
- Accessibility of resources for self-teaching
- Use of mailing lists or other similar resources

APPENDIX B

Code Book for Analysis of Interviews in Educational Experiences Project

Code Abbr	Full Code	Long Description (if needed)
F-	Actively Impeding	
F	Passive Lack of support	
F+	Support	
MI	Missing info	Did not learn some information;
EW	Extra work	Having to do extra work -> find a different way to do something, additional steps, etc.
Diff	Different work	
Mot	Lack of interest/motivation	
UA	unsure if tool is accessible	
InM	Inaccessible Materials	
Alt	Alternate Materials	
NM	No Materials	
G-	Harmed Grades/learning	
Dia	Diagrams	
PM	Poor(incorrect/filled with errors) materials	
SE	shared experience/mentoring	
cop	coping	
altexp	expertise of assistants	
just	justification	
sug	advice/suggestions	
access	accessible tool/materials	
math	math	
dep	dependent on help from others to do/check	

APPENDIX C

Tactile Graphics Formative Survey

1. Name
2. Age
3. Gender
4. Highest level of education
5. How would you describe your ability to see? (Blind/Low Vision)
6. Since the terms blind and low vision can have different meanings to different people, how would you describe (*in your own words or in medical terms*) your vision level / ability to see?
7. At what age did you begin to experience vision difficulties? (*That is, have you been blind since birth, is this a degenerative condition that began at a certain age, etc.*)
8. What is your level of proficiency in reading Braille?
9. How long have you known or been learning Braille (in years)?
10. How often do you read or write Braille?
11. Have you ever used tactile graphics before? If so, in what capacity?
12. How often do you use tactile graphics?
13. If you have used tactile graphics, in what format were the text labels? Check all that apply.
    a. The text labels were in Braille
    b. There were no labels
    c. The tactile graphic was placed on top of a touch screen
14. Which accommodations do you typically use? Check all that apply.
    a. Braille Note Taker
    b. Screen Reader on a Desktop/Laptop
    c. Screen Reader on a Smartphone
    d. Braille Reader
    e. Tactile Graphics
    f. Other:
15. Do you have a smartphone? If so, what type? Check all that apply.
    a. No Smartphone
    b. iPhone
    c. Android
    d. Windows
    e. Blackberry
    f. Other:
16. Which of the following applications have you used that require you to use a camera? Check all that apply.
    a. Camera (i.e. taking pictures of friends and family)

b. Currency Reader (e.g. Looktel Money Reader)
c. Color Identifier (e.g. Color ID)
d. Photo Interpreter (e.g. VizWiz)
e. Light Detector (e.g. Light Detector by EveryWare Technology)
f. Barcode Reader (e.g. Digit-Eyes)
g. Identifying Objects (e.g. TapTapSee)
h. Other:

17. How often do you use an application that requires the camera?
18. If you haven't used an application that requires the camera, why not?
19. Have you ever used an application that gave you feedback to help you aim the camera? If so, what was the feedback and was it helpful? If not, do you think feedback would be helpful?

For the remaining six questions in this section, please think of an application that you use regularly that requires aiming the camera and relate all your answers to that application.

20. What is the application?
21. Did you use any strategies for aiming the camera?
22. Was there anything you found to be particularly difficult or frustrating in using the camera?
23. Did you have any difficulties with other aspects of camera use not related to aiming the camera (i.e. covering the camera with your finger, problems with lighting conditions or the flash)?
24. Do you believe you have become better at using the camera in this application over time?
25. If yes, how long did it take you to become adept at using the camera?

APPENDIX D

Script for Follow-up Interviews to Formative Survey for Tactile Graphics

1. Highest level of education:
2. Level of vision:
3. Describe familiarity with braille (note physical limitations):
4. Describe how you use braille during education classes and at work:
5. Describe how you use braille on a day-to-day basis:
6. Describe your opinion of braille. How important is it in your life?
7. How important is it for other people who are blind or low vision?
8. Describe your familiarity with tactile graphics:
9. Describe instances in education classes or at work when you use them.
10. How were they labeled?
11. What were they made of? (E.g. things from the environment such as pipe cleaners or puff paint or embossed on thermoform paper or by a braille embosser.)
12. Describe instances in your everyday life that you use tactile graphics.
13. How are they labeled?
14. What are they made of?
15. If you could change anything about how you used tactile graphics during education classes, what would you change? (E.G. more tactile graphics, less tactile graphics, etc.)
16. What is your opinion of tactile graphics (useful, not useful.)
17. Describe other accommodations you use during education classes and at work. These can include technologies such as screen readers or braille notetakers that you used, or other aids such as a white cane or creative solutions you developed.
18. Describe your familiarity with using a camera.
19. How long have you been using a camera?
20. What types of cameras do you use? (iPhone, digital camera, etc.)
21. Explain the process of taking a picture.
22. Do you use any smart phone applications that use the camera?
23. What are they?
24. How often do you use each?
25. How long did it take you to become adept at using the application?
26. Are there any applications that you downloaded, but choose not to continue using due to the poor design of the application? If so, what were the problems that forced you to stop using the application? How long did you use the application before choosing to stop using it?
27. Give an example of how you use each application that uses the camera.
28. What would you like to change about your ability to use cameras?
29. What suggestions do you have for people developing accessible technologies to make it easier for people who are blind or low vision to take pictures?

30. Have you used applications that give feedback to help aim the camera? If so, what were they? How helpful were they? Is there anything you'd change about them?
31. Are there any other types of feedback that you would want?

OUR STUDY:
We are working on a project entitled tactile graphics with a voice. We are replacing the Braille text labels on tactile graphics with QR codes (do you know what QR codes are? they are similar to barcodes, textual information is stored in a QR code. When the QR code is scanned, the information stored in it is relayed.) For this project, we are storing information about tactile graphics such as equations on the QR codes. We are doing this project because QR codes take up less room on the graphics than Braille and to accommodate people who do not read Braille. As part of this project, we are working on a smartphone application that gives feedback for scanning the QR codes. This application uses the camera on the smartphone to detect if there is a partial QR code on the screen and gives feedback to the user to move the camera.

32. Do you have any thoughts on this project?

Types of feedback that we could give would be verbal (e.g. move "left/right/up/down"), tonal (beeps more quickly as QR code gets closer to center) or haptic (phone vibrates as the QR code is centered or you can feel on the screen where the QR code is located).

33. Is there some feedback that you think would be more helpful?
34. More annoying?
35. Easier to use?
36. Easier to learn?
37. Do you think some of the feedback might be helpful in the beginning, but less useful after you become accustomed to scanning codes?
38. Do you think that there are any situations that you would choose to turn the feedback off?

In some images, there might be two QR codes close together, and you wouldn't know which one the camera is scanning. One way to allieviate this problem is to point to the QR code that you would like and have the camera detect your finger. What do you think about this solution?

39. Is there any other solution that might work better?
40. Do you think tactile graphics can be useful if they do not have braille?
41. For people who know and use braille?
42. Why?
43. For people who do not know braille?
44. Why?

APPENDIX E

Tactile Graphics with a Voice Study

Initial Survey

1. Age
2. Gender
3. Highest Level of Education
4. How would you describe your ability to see? (Blind/Low Vision)
5. Since the terms blind and low vision can have different meanings to different people, how would you describe (*in your own words or in medical terms*) your vision level / ability to see?
6. At what age did you begin to experience vision difficulties? (*That is, have you been blind since birth, is this a degenerative condition that began at a certain age, etc.*)
7. What is your level of proficiency in reading Braille?
8. How often do you read or write Braille?
9. How often do you use tactile graphics?
10. Do you have a smartphone? If so, what type? Check all that apply.
    a. No Smartphone
    b. iPhone
    c. Android
    d. Windows
    e. Blackberry
    f. Other:
11. How often do you use a smartphone application that requires the camera (e.g. object identifier, currency reader, etc.)?
12. How would you rate your ability to scan a QR code using your smartphone? (1-No experience – 5 Expert)

Post Session Survey

5. Session Number
6. Please rank your preference for the feedback methods (1 is high and 3 is low):
7. Please rate how much you liked using the different types of feedback: (1-Strongly like to 7-Strongly dislike)
8. Please rate how helpful you thought the different types of feedback were. (1 – Very helpful to 7 – Very unhelpful)
9. Please rate how easy to use you thought the different types of feedback were (1 – Very easy to use to 7 – Very hard to use)

10. Please rate how easy to understand you thought the verbal feedback was: (1 – Very easy to understand to 7 – Very hard to understand)
11. Please share any thoughts on what you liked and didn't like about the different types of feedback.

APPENDIX F

StructJumper Questions

Initial Survey

1. Age
2. Gender
3. Do you identify as blind or low vision?
4. How many years of programming experience do you have?
5. How many years of experience with Eclipse do you have?
6. How many years of experience with Java do you have?

Post-code base questions:

1. On a scale from 1 to 7, rate how easy you found the task to complete, with one being very difficult, and seven being very easy
2. On a scale from 1 to 7, rate how frustrating you found the task to complete, with one being very frustrating, and seven being not at all frustrating
3. On a scale from 1 to 7, rate whether you felt you had a good idea where you were in the code with one being no idea and 7 being-always knew

End of Session Interview (Semi-Structured)

1. Reflect on how the experience of navigating through the code was different with the tool than without the tool.
   a. How did the tool affect your ability to complete the tasks
   b. How did the tool affect your ability to know where you were in the code
   c. How did the tool affect your ability to understand the code
   d. How did it change how you do your initial skimming or orient yourself ;
2. How does it compare to other tools already use?
   a. Ordering
   b. amount of info
3. Would you use the tool? For what?

APPENDIX G

Dynamic Graph Questions

Screener Survey

1. Gender
2. Age
3. How would you describe your vision? (Blind/Low Vision)
4. Since the terms blind and low vision can have different meanings to different people, how would you describe (*in your own words or in medical terms*) your vision level / ability to see?
5. At what age did you begin to experience vision difficulties? (*That is, have you been blind since birth, is this a degenerative condition that began at a certain age, etc.*)
6. What assistive technology do you use?
7. What is the highest level of education you have completed
8. Software development expertise
9. Graph expertise (Note we are referring to the graphs made up of vertices and edges, not charts and line graphs)

Post-code base questions:
On a scale from 1 to 7, rate how much you agree with each statement with 1 being strongly disagree, 4 neither agree nor disagree and 7 being strongly agree

1- The tasks were easy to complete
2- The tasks were frustrating to complete
3- I had to hold a lot of information in memory
4- I could easily determine what had changed between the two graphs

End of Session Interview (Semi-Structured)
Reflect on how the experience of answering questions about the graphs was different with the two modes.

- Which mode did you prefer and why?
- If you were able to switch between the modes while completing the tasks, how would that affect your approach?
- Did your approach vary the different modes? If so, how?
- How did the two modes affect your ability to complete the tasks?
- How did the two modes affect your understanding of the graphs changes?
- What information did you hold in memory with each of the modes?
- Is there anything else that was different of the experience for the modes?

APPENDIX H

Notes on Running Remote Studies with Blind Participants

- **Have the user share their screen with yours**: There are some services that allow a user to take control of your screen allowing you to have the software on your computer. However, a user cannot use their screen reader with this method making it unusable. Instead, you will need to send the software to the user and have them share their screen with you. You may need to walk them through this process, so where the options are in the menus, if it will ask for additional information (e.g. confirmation, select a screen), etc. is useful to know ahead of time.
- **Provide users with options on which screen sharing service to use**: Accessibility of software can vary based on the combination of screen readers and browsers. Additionally, users may not want to use a specific product due to poor previous experiences. Therefore, provide the user with multiple options so that they can choose a service that they are more comfortable with.
- **Determine how you will follow along with the user's actions:** For some tasks, it can be difficult to follow along with the what the user is doing. Depending on the user's settings, you may not be able to hear a user's screen reader over the call, which can make it harder to follow along with what is happening. It can help to make sure that your software has clear visual focus for you to follow or a log to go back and see their actions.

VITA

Catherine Baker is from the Paul G. Allen School of Computer Science & Engineering at the University of Washington and is advised by Richard Ladner. Her research interests are in the areas of Human-Computer Interaction and Accessibility. More specifically, her research focuses on creating technology for blind students studying computer science. She is a NSF Graduate Research Fellow (2013-2016). Catherine received her MS in Computer Science & Engineering from the University of Washington (2014) and her BA in Computer Science and Mathematics from DePauw University (2012).