Vocal Programming for People with Upper-Body Motor Impairments

Lucas Rosenblatt[1], Patrick Carrington[2], Kotaro Hara[3], Jeffrey P. Bigham[2]
[1]Brown University, [2]Carnegie Mellon University, [3]Singapore Management University
lucas_rosenblatt@brown.edu, {pcarring, jbigham}@cs.cmu.edu, kotarohara@smu.edu.sg

ABSTRACT

Programming heavily relies on entering text using traditional QWERTY keyboards, which poses challenges for people with limited upper-body movement. Developing tools using a publicly available speech recognition API could provide a basis for keyboard free programming. In this paper, we describe our efforts in design, development, and evaluation of a voice-based IDE to support people with limited dexterity. We report on a formative Wizard of Oz (WOz) based design process to gain an understanding of how people would use and what they expect from a speech-based programming environment. Informed by the findings from the WOz, we developed VocalIDE, a prototype speech-based IDE with features such as Context Color Editing that facilitates vocal programming. Finally, we evaluate the utility of VocalIDE with 8 participants who have upper limb motor impairments. The study showed that VocalIDE significantly improves the participants' ability to make navigational edits and select text while programming.

CCS Concepts

• **Human-centered computing~Human computer interaction (HCI)** • *Human-centered computing~Interaction techniques* • *Human-centered computing~Accessibility systems and tools*

Keywords

Speech recognition; programming tools; upper-limb impairment; Cerebral Palsy

1. INTRODUCTION

The most widely used method for text entry on computers is using a physical keyboard, often with the QWERTY layout though alternative layouts are sometimes used [7][22]. While the use of physical/software keyboards are prevalent, they are not accessible for people with limited upper body movements [27][33]. The problem is exacerbated when people try to do typing intensive tasks, such as coding software programs. The accessibility barrier may be preventing people with motor impairments from entering the software industry. In fact, only 4% of professional programmers have physical disabilities [17]—a rate lower than the 8.2% of the general population who have "difficulty with physical tasks relating to upper body functioning" [6]. This disparity between the distributions of the general population and the "coding" population suggests that those with upper limb impairments are under-represented in the developer community.

W4A '18, April 23–25, 2018, Lyon, France
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5651-0/18/04...$15.00
https://doi.org/10.1145/3192714.3192821

Supporting people with motor impairments to type through assistive technologies may facilitate people to enter the coding industry and reduce the disparity in coding population. Prior work has shown that soft keyboards or specialized trackballs can allow a user to *type*, but these systems can be inefficient, difficult to learn, and expensive [24]. More importantly, the primary target of these tools is general computer use rather than the specialized domain of *coding*—a task that involves much more structured typing, numerous symbols, and less flexibility for errors.

There have been a few recent notable hands-free computer programming tools. Tavis Rudd's dictation-based python programming system as well as Ben Meyer's vocal programming system VoiceCode[1] are two examples of existing vocal programming systems. While purported to be useful for people with Repetitive Stress Injuries (RSI)[2] or for people with motor impairments, neither system is specifically designed to be used by those with upper limb motor impairments, and neither has been studied with this population. Most relevant to the current work is a tool designed by Begel and Graham. They designed and developed voice-based coding and noted that it has potential benefits including a reduction in tasks that could lead to or exacerbate RSI, improved access for people with existing motor impairments, and the potential to further explore and understand of the process of coding and speech recognition in specialized domains [2][3]. Voice is an input modality used by many people who cannot type on a physical keyboard, and so lends itself to this domain [15]. However, an evaluation of voice-based coding tools with people who have limited dexterity is missing.

In this research, we first explored the design space for vocal programming environments and re-evaluated its utility, then designed, developed, and evaluated our prototype vocal programming system, *VocalIDE* (Figure 1). The work involved three parts. First, we conducted a Wizard of Oz study (WOz) with ten participants *without* motor impairments who each completed a series of programming tasks. Each participant gave vocal instructions to a researcher, who controlled a text editor based on a predefined protocol. Second, based on results from the WOz study, we designed and developed VocalIDE, a vocal programming editor. Finally, we conducted a second study with eight participants who have upper limb mobility impairments to assess the usability of our prototype.

[1] Meyer, B. Accessed May 23, 2017. *Advanced voice-control Platform*. https://voicecode.io/.
[2] Rudd, T. http://pyvideo.org/pycon-us-2013/using-python-to- code-by-voice.html. Using Python to Code by Voice. PyVideo.org, Accessed May 23, 2017

The main contributions of this work are threefold: (i) insights into the language inclinations of programmers who used (WOz) speech-based coding environment, (ii) the functional prototype system for speech-based coding written in JavaScript, and (iii) evaluation conducted with people who have upper body motor impairments that shows the feasibility and utility of using a system such as ours, suggesting people can use the system to effectively edit code.

2. RELATED WORK

In this section, we describe related work in programming, accessibility, speech-based interfaces, and speech-based *programming* interfaces.

2.1 Programming and Code Editing

Software development process involves multiple activities incorporating both mental and physical tasks. LaToza *et al.* [21] introduced a taxonomy of activities associated with development, describing nine different activities: designing, writing, understanding, editing, unit testing, communicating, overhead, other code, and non-code activities. Among the nine activities, the physical activities of writing and editing code—tasks that are traditionally performed using text editor or integrated development environment (IDE) with a keyboard—would put the biggest burden on people with limited upper body movement.

Yet, writing and editing is inevitable in software development process. In fact, Desilets [9] provides a good interpretation and discussion of the study by Singer *et al.* [31] describing the importance of code editing tasks in the development process. They described how 8 out of the 9 programming activities described by Singer presented similar challenges for programming-by-voice as code editing challenges. This provides further evidence of the importance of editing and navigation tasks in programming. VocalIDE supports both writing and editing, although focuses especially on code editing.

2.2 Text Input Methods for People with Upper-Limb Motor Disabilities

Much of accessibility research on motor impairments has focused on understanding how users with motor impairments interact with computers through different methods, including speech [8][14][15], touch [12][33], and gesture [38]. Of particular prevalence is improving or augmenting pointing-based interactions. Wobbrock *et al.* developed a gesture-based alphabet input method, that could be used through multiple input styles including a stylus [41], a trackball [37], wheelchair joysticks, and touchpads [42]. These gesture-based techniques alter the interaction style allowing the user to choose the appropriate input device based on their abilities and preferences. We focus on speech interaction because, for those who have difficulty using a keyboard but little to no difficulty speaking, it could be a much faster method for input compared to typing [28].

Prior research has explored speech based interaction methods including cursor control [8][14], drawing [15] and text entry [34]. For example, Sears *et al.* studied how accurately people can correct text using dictation interfaces [11][29]. Their studies showed that correcting text in dictation interfaces can achieve promising results [11][29]. However, the added complexity of switching between different input approaches introduced new interaction cost of correcting and editing, making correction and editing of dictated text remain tedious and slow [34]. Thus, we suspect these methods are not suited for people with motor impairments.

2.3 Speech-Based Programming Systems

Programming, as described, is an expert task; it will always require some user education, whether self-guided or instruction-based, in order for interactions to produce results (*e.g.*, sample commands, understanding of editing mechanisms, *etc.*). In both Rudd and Meyer's solutions, a new user must also learn a host of new, non-natural vocal commands to be able to effectively use the software. Those same systems require an unreasonable level of precision to effectively program. This creates additional cognitive work, especially for those who are temporarily injured or for users where a speech impairment is present. Both systems require a great deal of knowledge and (keyboard driven) setup. Neither was designed with persistent disabilities in mind, and neither system was designed empirically (involved no user study).

Research on programming by voice has produced both guidelines and tools for vocal programming. Most existing projects focus on the potential of vocal programming to assist experienced programmers who may experience RSI [1][2][3][9][10]. Existing programming-by-voice systems, VoiceGrip [9] and VoiceCode [10], allow the user to dictate using pseudocode, rather than having to dictate each character. However, neither example included a formal evaluation of the system but [9] suggested that the system was available for use and had been used by active programmers. Begel and Graham [2] explored programming by voice by having experienced programmers verbalize a portion of a java program. Spoken Java and the SPEED editor were developed to enable speech-based programming and were implemented as an eclipse plugin and evaluated with experienced programmers [3]. The system and evaluation provide positive insight into programming by voice and usability issues that need to be overcome by new systems. Results suggest that, for experienced programmers, the Spoken Java system may not desirable for everyday use but would be a usable alternative.

Few projects have focused on vocal programming for people with persistent motor impairments. Notably the Myna system was developed and tested for use with children with motor impairments [35][36].

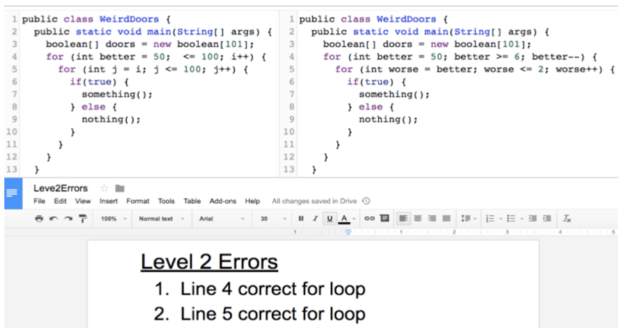
After reviewing the current available speech-based interaction systems, as well as Rudd and Meyers work, we realized that these systems could benefit individuals with upper body motor impairments, but that few to no exploratory evaluations had been conducted. What would the best vocal programming system look like for someone with an upper limb mobility impairment? How can we design such a system effectively, and how can we evaluate it empirically? We began by first observing programmers attempting to code using natural speech, curious to see how these observations would influence our system design.

3. STUDY 1: PRELIMINARY WIZARD OF OZ STUDY

We conducted a Wizard of Oz study (WOz) with people *without* disabilities to gain an understanding of user instincts when giving vocal commands to a computer. The key questions included: What commands are programmers inclined to give a computer using their voices in order to write and edit code? What are important friction points and the most difficult tasks to complete?

3.1 Participants

The study was conducted with 10 participants (5 female) with coding experience but without motor impairments. Note that the recruited population is not the target user group of the intended final product (*i.e.,* VocalIDE that targets people with limited dexterity). However, working with coders without disabilities was a good first iteration of the design process, because: (i) it allowed us to tease out design bugs and understand frequently used commands for vocal coding (whether one has or does not have a disability); and (ii) the population was relatively easy to recruit, allowing us to quickly get feedback on the initial design of the tool (which we will evaluate with participants who have upper limb impairments in Study 2). The participants were recruited from a pool of summer research interns (computer science undergraduate students) at Carnegie Mellon University via social media. The average age of the participants was 20 years (SD=1.15). The participants had 2.7 years (SD=0.95) of programming experience on average. All had taken at least one programming course.



```
1  public class WeirdDoors {
2    public static void main(String[] args) {
3      boolean[] doors = new boolean[101];
4      for (int better = 50;  <= 100; i++) {
5        for (int j = i; j <= 100; j++) {
6          if(true) {
7            something();
8          } else {
9            nothing();
10         }
11       }
12     }
13   }
```

```
1  public class WeirdDoors {
2    public static void main(String[] args) {
3      boolean[] doors = new boolean[101];
4      for (int better = 50; better >= 6; better--) {
5        for (int worse = better; worse <= 2; worse++) {
6          if(true) {
7            something();
8          } else {
9            nothing();
10         }
11       }
12     }
13   }
```

Leve2Errors

Level 2 Errors

1. Line 4 correct for loop
2. Line 5 correct for loop

Figure 1. Study 1 participants were asked to correct the code on the left until it matched the code on the right using vocal instructions.

3.2 Method

The Wizard of Oz (WOz) method allows researchers to demonstrate a working system by having a human "wizard" simulate the functionality or intelligence by interacting with the user through either a real or mock interface [26]. Following a similar procedure to Begel's exploration of experienced programmers vocalizing Java [2], we asked participants to direct a "human computer" (the first author) to correct code that was provided to them. The participant's screen mirrored the display of the "human computer's" screen, who operated the text editor according to a literal interpretation of participant's speech. A list of errors was provided to control for variation in debugging time and strategy.

The participants were asked to work on six coding problems. In each problem, the participant was presented with two blocks of code side by side (Figure 2) where the answer was shown on the right side of the coding window and the left displayed a similar code snippet that contained some errors. Each problem contained varying types of errors to diversify difficulty levels. Our goal was to elicit as many design requirements and types of interactions/vocal commands as possible. The majority of problems (five out of six) that we administered were the editing task rather than other types of tasks (*e.g.,* writing a code snippet from scratch—a type of a problem one would see in a coding interview)—because (i) editing existing code is a large part of programming process [9], [31], and (ii) it reduces the effects of programming difficulties that is not necessarily related to interaction challenges in entering codes with voice commands.

The six problems were created following an analysis of common Java code and errors and had varying difficulty levels. We scraped the 50 most in-linked algorithm pages from `rosettacode.org`, combining the code to form a common Java corpus. We ran further text analysis to create a standard of common Java syntax (*i.e.*, what syntax/code blocks are most common and in what order). Jackson, Cobb, and Carver [18] provide an approximation of the 20 most common Java errors. We used this as a starting point to create realistic, common errors on each level of the study. This process ensured that levels were representative of important/common challenges a programmer faces when writing/editing Java code. The levels began with "easy," smaller code edits, and progressed in difficulty by including more text generation, selection and navigation until level six, which was only code generation.

3.3 Result

The participants used different commands when vocal programming, although they shared common approaches. The most common words after excluding stop words like articles (*e.g.,* "the", "to") were "right," "line," "space," and "after." Commonly appearing words were navigational in nature, suggesting that a majority of participants spent their words on referencing locations in the code. For closer text analysis, we use lexical density (LD) (a measure of words' significance in a text, or how many words contribute to the overall meaning of a text) [19]. The overall lexical density of the participant corpus was 6.9%, which is far less than average speech (one study suggests that speech interviews tend to have a lexical density of ~45%) [19]. Even though we expected a limited vocabulary due to the finite nature of coding possibilities, our analysis of transcribed participant speech suggested that the users were inefficient with natural speech vocal commands. A number of individuals typed letter by letter, while some participants provided new feature insights that we had not yet thought of (for example: combine "search" and "type" into one command "change").

4. Working Prototype: VocalIDE

Drawing from prior research (*e.g.* [2]) and the results of our WOz study, we developed VocalIDE, a voice-to-code editor, using JavaScript. VocalIDE was designed and developed to accommodate a user with an upper body motor impairment that prevents them from quickly and/or accurately entering text on a computer via keyboard. This user may have used accessibility technology in the past to help with text entry, but has struggled with writing code due to the constraints of programming (tricky syntax, edits, odd language *etc.*). This user can speak clearly, and would benefit from a system that makes text entry easy while specifically "listening" for programming syntax as input. This system would then make editing this input via tools easy, increasing the speed and accuracy of a user's interactions.

VocalIDE is a web application that allows users to write and edit program code using a set of vocal commands. This is enabled by two system components: browser-based automatic speech recognition (ASR) and a rule-based syntax parser. The speech-based coding workflow starts from turning on ASR. VocalIDE records and recognizes users' speech using WebKitSpeechRecognition, which is natively available in WebKit-based modern browsers (*e.g.*, Google Chrome). The interpreted speech is then passed to a rule-based syntax parser. The parser accepts `command` *`parameter`* where `command` represents a keyword reserved for the system (*e.g.,* `type`, `select`) and *`parameter`* represents an arbitrary additional argument given by a user.

Look to Table 1 for the summary of VocalIDE's functionality.

4.1 Commands

Writing and editing code is done via the following commands: *text entry, navigation, text selection*, *replacement, deletion, undo,* and *snippet entry*—a set of basic functionality of vocal text editing identified in the WOz study.

- ***Text Entry:*** Users can enter new text by vocalizing the command `{type | write | add}` followed by a user specified parameter. For example, the user can say "`type` *`open parenthesis i space less than space one close parenthesis`*" to enter "(i < 1)". The parameter could be a word, alphanumerical letter, special character (*e.g.,* '(', '+', '.', ',', '\', '}', ')') or space/tab/return.
- ***Navigation:*** A user can reserve keywords to navigate between lines of code with the commands `{go to (line) | move to (line) | line}` and a line number. For example, saying "`move to line` *`three`*" moves the cursor to the third line of the code, as does just "`line` *`three`*". Alternatively, a user can say `{up | down | left | right}` to move the cursor just like one moves around a cursor by pressing arrow keys. A user can stack these commands – 5 "*`up`*" commands will move the cursor up 5 lines, as will "*`up 5 lines`*"
- ***Text Selection:*** Selecting instances of words or phrases can be done by saying `select` and then the word or phrase. The word or phrase that matches and is nearest to the cursor is then selected – saying next can toggle between repeated words or phrases. Spaces are ignored, so multiword selections are possible. A user can also say "`statement`" or "`block`" to be presented with options to select between specific syntax (for example, the text within a set of parentheses). The user can also say "`line select`" to select a line, or "`select word`" to select the word in which the cursor currently resides.
- ***Replacement:*** The replace command is unique in a sense that it can only occur after text selection. The replace syntax is as follows: `{ change | replace }` *`parameter`* `{ to | with }` *`parameter`*. For example, a user might say "`replace array with array open bracket close bracket`" and the system would replace the nearest occurrence of "array" to the cursor position with "array[]". Specialized commands like replace were added based on observations from the WOz study, where participants would often use similar semantic shortcuts. *Replace* expects the user to have a specific change in mind and so is prone to error (articles like "a, the etc." are eliminated, but other words are not).
- ***Deletion:*** The delete command allows a user to either delete a character or delete a selected region of text. The delete syntax is as simple as `{ delete | remove }`. By default, delete command deletes a character behind the current position of the cursor. But if a text region is selected prior to deletion, the selected text gets deleted.
- ***Undo:*** The undo command rolls back any change made to the code in the prior step. Spoken syntax is simply: `undo`.
- ***Smart Snippet Entry:*** In addition to the above basic functionality, VocalIDE implements what we call a *smart snippet entry*. In the system, a set of keywords are reserved to help a user entering a block of text. For instance, if a user issues a command that contains the keyword "variable," VocalIDE creates a new variable snippet with filler text (*e.g.,* "`new variable`" creates "`var x = _`" where the variable could be an array, char, boolean, string, integer, *etc*). Similarly, if a user vocalizes other common code blocks such as a "`make a for loop`", "`make an if statement`", "`make a while loop`", "`make a new function`", the system will auto generate the corresponding block for the user. Note, however, this feature was only partially implemented for at the time of testing. What it does not do well is interpret commands such as "`make a new variable foo with value 3`" – given this the system will still generate a blank filler variable with no value, ignoring the part of the command to initialize the variable (*i.e.* "`with value 3`").

Text transcribed by ASR is assessed incrementally from the first word of the text. If none of the commands or words reserved for the programming language match the first word, the word is ignored, and the next word is evaluated. VocalIDE prioritizes longer commands in evaluation; the more complex commands (multi-token, like "`make an if statement with a greater than sign`") are evaluated first, followed by less complex commands (*e.g.* "`go right 3 times`").

Depending on the interpreted command type, the list of numbers is checked in order. Note that a command like "`go to line 4 then right 5 times`" will be carried out correctly, but that a command like "`go right 5 times on line 4`" will instead go to line 5 and right 4 times. These semantic differences are difficult to prepare for, and need to be addressed in further system development.

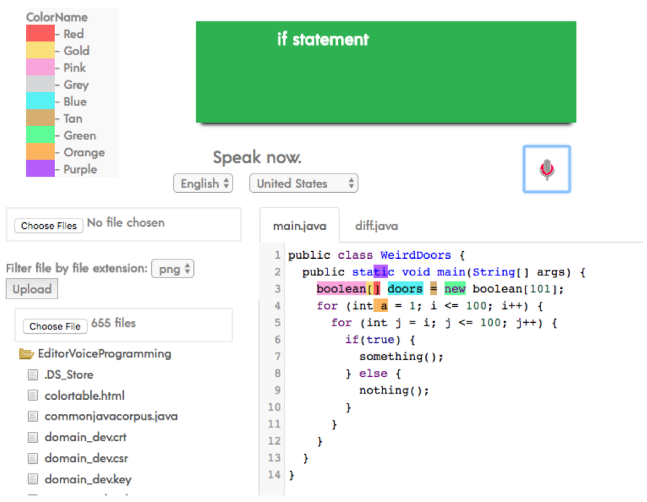
Table 1. A summary of the four main components of VocalIDE: the main speech interpreter, keyword parsing, smart snippets, and color context edits.

Functionality	Description
Main Interpreter	Users can enter new text by word or by letter, enter syntax (+, ., \, }, etc), navigate using line numbers or cursor commands, and select instances of words or phrases.
Keyword Parsing	VocalIDE ignores most articles and filler words, attempting to search a command string for pertinent or significant command information
Smart Snippets	User commands that contain specific keywords are sometimes interpreted as a "snippet" (i.e., for, if, while, variable, array, etc). A filler element of that type is inserted.
Color Context Edits (Figure 3)	CCE follows the cursor position, and highlights individual syntax and words on the cursor's line. CCE can also be used to select large bodies of text based on syntax.

4.2 Color Context Editing

The WOz study revealed that users are inefficient when navigating text using basic vocal commands (*e.g.,* `down, left`) because it only moves one character/one line at a time. Therefore, tasks like selecting a word/words (to edit or delete) using multiple execution of navigation commands is time consuming and potentially error prone (because more command entry could introduce more command parsing error in the ASR step). To address this problem, we implemented Context Color Editing (CCE).

Context Color Editing is always on, following the cursor position, and highlights individual syntax (brackets, parentheses, periods, etc.) and words on the cursor's line (space separated collections of characters), as well as on the lines above and below the cursor position (Figure 4). To let a colored word(s) to be selected, the user needs to vocalize the corresponding color (like "`red`") for that color's highlighted text (the text that happens to be highlighted in "red" at that moment) to become the active selection. CCE is limited, as of now, to 9 total color selections (4 before the cursor location, 3 after, 1 above and 1 below). It is trivial to add more

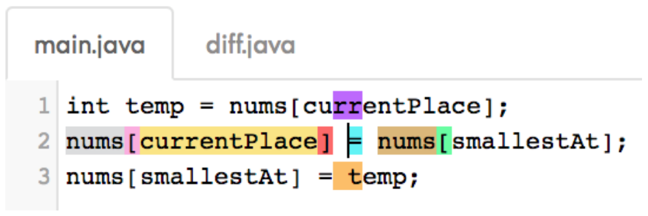


```
public class WeirdDoors {
  public static void main(String[] args) {
    boolean[] doors = new boolean[101];
    for (int a = 1; i <= 100; i++) {
      for (int j = i; j <= 100; j++) {
        if(true) {
          something();
        } else {
          nothing();
        }
      }
    }
  }
}
```

Figure 2. Screenshot of full system – top right green box displays recent commands, editor bottom right, file system bottom left, and CCE key is top left.

color selections, although less trivial to find more easily differentiable colors with short names. The goal of CCE is to allow users to be more efficient with their commands, replacing cumbersome sentence structures with short, quickly spoken colors.

Context Color Editing was iteratively designed within the research team. We chose to use color-based selection because the visual feedback makes the system easy to remember. We found, through an informal assessment, that the introduction of CCE greatly improved close context editing ability of users (*i.e.,* members of research team) as they could quickly (re)select a word(s) with color commands.



```
int temp = nums[currentPlace];
nums[currentPlace] = nums[smallestAt];
nums[smallestAt] = temp;
```

Figure 1. Screen shot of CCE – highlighting quick selection possibilities for the user's convenience.

5. Study 2: VocalIDE Usability Evaluation

To assess the feasibility of using VocalIDE to edit code, we conducted a study with people with motor impairments. We recruited N=10 people through a local accessibility organization in Pittsburgh, PA. Each participant was selected by a supervisor at a local assisted living and support facility to make sure they had upper body impairments and did not have other disabilities (*e.g.,* visual impairments, cognitive impairments). As a feasibility study, we did not specify the level of dexterity that each participant had (but instead we measured their dexterity levels with Box and Block Test as we describe below). In addition to these criteria, we: (i) recruited those who use computing devices at least occasionally to make sure they know how to interact with text editing environment in VocalIDE, and (ii) could "speak clearly."

Two people did not complete the entire study and were not included in our evaluation. Of the remaining eight people, four were female. Their ages ranged from 19 to 50 years old (*mean*=30.6, *SD*=10.6). Three participants used computers less than once a week, two used computers once a week or more, and three used computers daily or more.

We note that the participants in this study were non-coders. This was primarily due to a lack of access to experienced programmers with motor impairments. We acknowledge that our study may have benefited from including experienced programmers with chronic motor impairments who fit the profile of our intended users; however, we argue that studying with the current population allowed us to study feasibility of using VocalIDE to generate and edit simple code structures sufficiently. We also point out that, with the exception of Wagner *et al.*'s Myna system [35][36], prior work on vocal programming has not involved people with motor impairments.

5.1 Method

All study sessions were conducted in a room in the local accessibility organization's office (Figure 5). In each session, one member of the research team explained the study process. The process involved 4 parts: (i) the Box and Block Test, (ii) the baseline current computer interaction test, (iii) the system evaluation using their voice, and (iv) an unstructured interview asking their experience in using both traditional user interface and VocalIDE. While our participants were working on the baseline and VocalIDE tasks, we observed the participants and took notes on what kinds of challenges they faced while coding.

Table 2. Participant profiles for usability evaluation study. SCI = Spinal Cord Injury, CP = Cerebral Palsy

PID	Age	Gender	Medical Diagnosis	Box/Block Test Score
P1	22	M	SCI	0
P2	26	F	CP	9
P3	28	M	CP	DNC
P4	27	M	Spinal Dysmorphism	DNC
P5	50	F	CP	25
P6	24	F	Did not Respond	30
P7	29	M	Stroke	17
P8	32	M	CP	20
P9	49	F	CP	24
P10	19	M	CP	21

5.1.1 Box and Block Test

To evaluate the participants' dexterity (and exclude them in the analysis step if necessary), we first asked them to complete the Box and Block Test[3]—one of existing standard tests to measure one's dexterity. The test uses a hinged wooden box that opens into

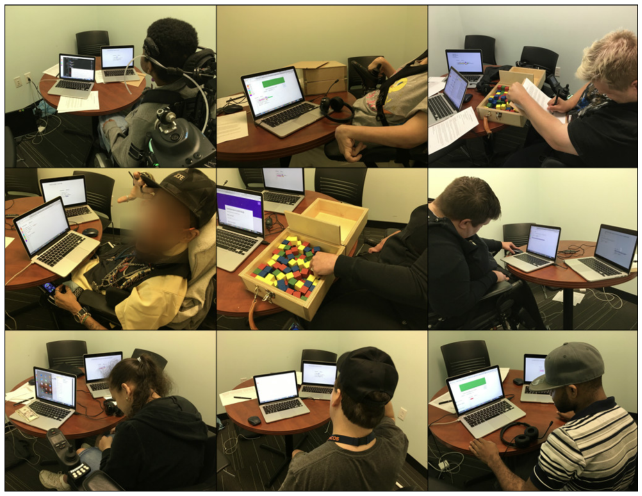


Figure 4. Participants using VocalIDE while completing usability evaluation. The center photo depicts a participant completing a Box and Block Test, which is a standard measure of dexterity.

[3] http://www.rehabmeasures.org/Lists/RehabMeasures/DispForm.aspx?ID=917

two halves, separated by a wooden divider. In either half of the box are 150, 2.5cm, wooden cubes of varying colors. The participant is asked to move as many cubes as they can, one at a time, over the divider into the other half in 60 seconds. Any block that travels across the divider in their hand counts toward their total score. They may not throw blocks across the divider, but if the block is carried across then bounces out of the box, it still counts towards their total. A higher score on the test indicates better gross manual dexterity. P3 and P4 could not complete the test due to limited upper body dexterity, thus they did not work on the Baseline and VocalIDE steps that we describe next.

5.1.2 Baseline Test using Keyboard or Preferred Assistive Device

We evaluate our participants' ability to edit code using a keyboard or other assistive technologies, which we treat as a baseline to assess if VocalIDE enables/improves participants' coding capacity. Each participant was asked whether they used any assistive technology to interact with a computer when entering text. If they responded yes, the researcher would assist setting up participant's any existing assistive technology on the test computer. Otherwise, they used a keyboard to enter text. The baseline condition was meant as a demonstration of using the participants' familiar method of text entry, whether that be with an assistive device or the keyboard. Maintaining this order allowed us to observe the participants' familiar input method prior to using VocalIDE.

We evaluate the participants' coding capacity along dimensions of coding processes described by LaToza *et al.* [21]. LaToza broke down coding interactions into nine categories (*e.g., writing, editing, unit-testing*). To assess the feasibility of coding, we focused on *writing* and *editing* while we ignored other categories like *designing* and *unit testing*. While the omitted activities are important, integral parts of software development, our goal was to see if VocalIDE can support writing code at all. We further broke down *editing* and *writing* into smaller sub-categories: adding text (ADD), removing text (REM), selecting text (SEL) and navigation within the body of the text (NAV).

The test consisted of three levels of tasks including variety of tasks that are involved in coding. Our goal in designing the levels was to include as many types of tasks in each level, but with varying difficulty (first level being easiest and the third level being the hardest). Similar to the Study 1, participants were given the correct code and the code snippet to edit in each task. Unlike the Study 1 setting where the correct code and code snippet were presented side-by-side, the correct code was shown on a PowerPoint slide on another computer's display. To give a hint of what to edit in the code snippet, the part that the participant was supposed to edit was highlighted on the slide. They were asked to complete each level by making the edits so that the editor's text appeared exactly as the text in the PowerPoint slide looked. The researcher recorded the breakdown of their successes, failures, and incompletes on a spreadsheet while they completed the task. A success was an edit that perfectly mirrored the correct version of a level (*i.e.* contributed to achieving the end goal edit). If the participant was unsuccessful after two attempts, the researcher asked them to move on to the next level.

The first level focused on small syntax edits, such as adding single characters, selecting and replacing single words, moving right, left, or down. For each edit, the researcher recorded a success, failure, and incomplete. A failure was when the user failed to use the system to make the correct edit, while an incomplete was when the user did not try to make the edit (we allowed this as every step of this process was voluntary – we did not want to impart unnecessary pressure on our participants using an unfamiliar system). Each edit was also classified as one of the following four categories: Navigational, Additional, Removal, or Selection. Participants' overall success was quantified based on their performance in these categories via their success and failure rates.

The second level focused on harder tasks. For example, Selection tasks required the participants to select a larger portion of the code (*e.g.,* line selection, multiline block selection), Additional and Removal tasks involved more complicated edits like copy/paste/cut, and rearrangement of code blocks. The third level focused on code generation, asking the participant to enter text/syntax from scratch in order to mimic a short while/if statement snippet. A screen recording of the test was captured throughout this step.

5.1.3 System Evaluation using VocalIDE

After completing the baseline to the best of their abilities, each participant was guided to work on a set of tasks with similar difficulty levels using VocalIDE. We first gave a short tutorial of using the VocalIDE system. The tutorial consisted of a short demonstration of the VocalIDE features described in the Section 4 of this paper. Following the tutorial, the participants completed three levels that consisted of the same types of tasks (*i.e.,* Navigational, Additional, Removal, or Selection) as the baseline test, but with different text and occasionally in a different order. Each task, however, required the same effort in the VocalIDE test as the baseline. For example, for level three, the variable name to enter might change, or the participant might have to move the text from line 2 instead of line 4.

5.1.4 Unstructured Interview

The unstructured interview was conducted at the end of the session to debrief the participant on their experience using both a traditional user interface and VocalIDE. The participant was asked to describe their experience using each interface and to discuss any perceived benefits and/or challenges of using each one.

5.2 Result

The results of our evaluation are presented in roughly the same format as the session described above.

5.2.1 Demographics and Box and Block Test

The results of the box and block test confirm that all of our participants had reduced dexterity when compared to normative data of the test [25]. The results are reported in Table 1 along with demographic information. Of the eight participants who participated in the Baseline and VocalIDE tasks, only three participants used assistive devices/software. One participant used a Bluetooth joystick to control the mouse, one participant used WordCue autocomplete, and another participant used IntelliKeys USB/Easyball.

5.2.2 Baseline vs. VocalIDE

To understand the effects of the task types and interface conditions to our participants' ability to perform the coding tasks, we used a generalized linear model (GLM) with logit link function. Note, we used GLM instead of the oft-used repeated measures ANOVA because the dependent variable was binary (*i.e.,* "completed" vs. "incomplete"). We had TaskType (*i.e.,* "ADD", "NAV", "REM", and "SEL") and Condition ("baseline" vs. "test/vocal side") as independent variables. We also had the interaction component to check whether the interface condition had varying effect on different types of tasks.

We had 8 participants who were able to complete both the baseline and the VocalIDE tests. P3 and P4 were unable to complete enough of the study to provide comparison. Each participant provided 40 data points for the baseline and the system test ("ADD", "NAV", "REM", and "SEL" binary completion stats and the time it took them to do so). This gave us a data set of 640 binary completion points.

We did not observe significant main effects of task types and interface condition, but we observed slight trend in the interface condition (p=0.078 < 0.1). This suggests that with VocalIDE may have positive impact in enabling/supporting people with limited upper body movement to work on coding tasks. The lack of statistical significance suggests more work with a larger population size is needed in the future.

We observed significant interaction effects between task types and interface conditions. This indicates that interface types had varying effects on completion of the types of tasks. In observing the success rate, the VocalIDE interface had a significant positive impact on completing NAV (p=0.02 < 0.05) and SEL (p=0.03 < 0.05) for our participants, but our interface did not improve how the participants could perform ADD or REM. The future work should further investigate the effects of the interface to each type of the tasks.

Because we could not measure the task completion time for incomplete tasks and the types of the tasks that our participants completed differed, we could not perform a statistical test similar to the task success rate analysis. Therefore, we report average time that our participants took to perform each task to get informal sense of task completion time. Our participants completed 300 tasks (150 baseline, 150 VocalIDE). The duration to complete each task in the baseline condition was 21.24 seconds, while the average task completion time in the VocalIDE condition was 13.81 seconds. Though informal, these results may suggest that VocalIDE could improve task completion time as the participants, overall, appeared to perform tasks faster in the VocalIDE condition than in the baseline condition where they used either keyboard or their choice of assistive text entry devices. Note, however, given the VocalIDE tasks were performed after the baseline tasks, the shorter average task completion time may be attributed to learning effect.

5.2.3 Unstructured Interview and Observation

We conducted the unstructured interview with all the participants (N=10) regardless of their participation in the coding tasks.

Assistive Technologies for Text Editing. Nine out of the ten participants reported that their ability to use a computer could be improved via some sort of better assistive technology, while the tenth participant was "not sure." When prompted to discuss more on current assistive interactive devices, the participants offered mixed reviews. One participant was excited to try a vocal system for general computer navigation. Another participant noted that they *"gets really tired after 30-60 minutes of typing"* due to the large mental effort that the interaction method requires. They were interested in *"anything that recognizes my voice, that would be able to copy down what I say with my voice."* Another participant was afraid that a computer voice system might have trouble understanding them because of their stutter. Therefore, they had not yet tried one. A participant that had tried a vocal computing system (Dragon Dictate) said *"it doesn't really work well... it did more to frustrate me than to help."* When pressed, the participant responded that the frustration was due to insufficient transcription accuracy and the consequent interactions/time required to fix the errors.

Interest in Voice Interaction and Programming. Eight of the ten participants said they were "interested" in a voice interaction with their computer, and 7 of the 10 noted that they often use their computer to enter and edit text. Four of the 10 participants said they were interested in programming or computer coding, but none of the participants said that they thought assistive technology existed to help them with programing.

Overall Reaction to VocalIDE. Overall, all participants positively reacted to using VocalIDE. For example, P2 who used a joystick and a virtual keyboard seemed to prefer VocalIDE over the assistive technologies in the coding tasks because it reduced the text entry/editing effort a lot. In the interview after using VocalIDE, they enthusiastically said *"[Entering text with VocalIDE] was much easier than the software keyboard."* P9 enjoyed speaking into the microphone and watching edits take place. They had never used a vocal system for computer navigation and thought it was fun. They stressed that they enjoyed using their voice with a computer, and would do it again.

Some preferred using traditional input devices. For example, P8 preferred using a keyboard, saying that they was *"used to it."* They did not want to change their text entry method even though they thought the VocalIDE system might be *"faster [for text entry]."* This suggests that there is non-negligible learning barrier for code editing by VocalIDE (and perhaps ASR-based text entry in general). P6 shared that they would use the VocalIDE *"if it were better at understanding"*—indicating that ASR's transcription was not accurate enough. This participant struggled particularly due to their accent, and found himself frustrated by certain unnecessary repetitions of commands. They stressed that they *"really liked the system"* and would *"use more often if it was better."* P10 also struggled with the system due to their accent.

The participants' comments reflected our motivation in facilitating people with disabilities to enter the software industry. P5 noted that *"[He had] always wanted to program a video game where someone's in a wheelchair and in high school – sort of like buffy the vampire slayer meets terminator…"* and *"people in wheelchairs are not well integrated into entertainment... I'd use any software that would help me build the video game."* They also said *"I can't wait to get your system when its on the market, it would be so helpful for writing stories. Now that I know I can voice type I will keep doing it."* P6 noted that *"I've always wanted to make a video game with code...now I can use my voice."* These quote suggests that VocalIDE (or perhaps any assistive technologies that enable people to code) has potential benefits in empowering people with disabilities to enter the software industry.

Insights from observation. While the participants were performing the tasks, we observed how our participants worked on the coding tasks. We were interested in advantages and disadvantages of VocalIDE. More specifically: Can the participants enter text in larger blocks much faster? Can they navigate the code more efficiently compared to the baseline input devices (*e.g.,* a keypad, joystick)? Especially advantageous was using the system for selection, which normally requires a click and drag (difficult for many of the participants), but with the system was made much easier to perform.

Response to Speech Differences - Accent and Stutter. As briefly mentioned above, by far the most frequently observed challenge that participants faced was misinterpreted commands due to ASR having difficulty recognizing speech differences (ASR responds best to clear, level toned speech, with no mumbling or affected intonations). Errors in ASR seemed to negatively affect the usability of VocalIDE. Most participants had at least one moment where they mumbled or spoke too softly. As a result, ASR made a transcription errors and the user had to repeat a command. The imperfect automated transcription was a more severe problem for some, because their disability not only affected their dexterity but also their speech. For example, P1's biggest impediment when using VocalIDE was their stutter, which seemed to become worse as their nerves increased. It should be noted that for P1, clarity was not the issue - it was the built in system timeout per command that caused VocalIDE to often misunderstand their commands. The system would correctly transcribe their words, but before they could say the next portion of the command (due to their stutter), the system would execute the half-formed command (usually resulting in either nothing or the wrong edit).

Challenges in Text Entry with Keyboard/Existing Technology. All participants who used a keyboard had difficulty in typing while working on the coding tasks. For example, for P1, pressing specific keys was quite challenging. They could only focus on one hand at a time, and had a success rate of pressing the right key, it seemed, of less than half. They really struggled with keyboard accuracy, and would often accidentally strike 2 or 3 keys at once.

Some features of VocalIDE seemed to help. For example, P7 described their issues with Dragon Dictate and their positive reactions to VocalIDE's use of CCE and Smart Snippets: *"I had to be so perfect with every word in Dragon, so deliberate…you don't have to be so perfect with your system. It's better for this."* They went on to say *"I thought [color selection] was super helpful - just to know it's gonna complete my thought [select what I want] for me was satisfying."* And *"For me, I want to say the least amount of things possible, the less you have to say the more[sic]. Dragon sucked because you had to learn a different language and everything had to be perfectly quiet."* This notion is consistent with users' sentiment to existing speech-based programming systems [[23]]. They also noted that "Like you had to do everything, but with this the shortcuts are super helpful." The participant went on to stress that "dragon was a nightmare" for someone with their *"other-ability"*.

6. Discussion and Future Work

The results show that VocalIDE could be used for coding and potentially improve the coding experience in some cases. Features like CCE seemed to be helpful as described in the Result section. Despite reactions such as this, speech recognition still seems to be the tallest blockade to efficient speech-based editing. Currently, our approach is dependent upon existing capabilities of speech recognition. However, as demonstrated workarounds can be implemented to overcome some challenges and as recognition improves, so would the usability of systems such as this.

The analysis suggested that our participants may perform Navigation and Selection tasks better with VocalIDE—two areas of editing that have proven difficult with dictation software [7][14]. Color Context Editing, in thinking about how we can make vocal interactions with a computer more efficient, seemed obvious post inception. Such a simple premise (color coded selection) provided a marked increase in usability. Small innovations such as these could improve all dictation software, saving untold man hours by shaving seconds off the mundane. More pertinent to this work, it can make certain tasks (selecting that annoying punctuation, for example) accessible to those who do not have the option to use a mouse or keyboard. Innovations like CCE could be implemented in common dictation software, such as Google Voice Dictate in Google Docs to name one.

Without creating an entirely new grammar for programming syntax, we were pretty sure that we would be able to significantly improve text generation. By having to reinterpret the English interpretation given to us by WebKitSpeech, we were quite limited in terms of accuracy given a range of inputs. For example, we would hope that the interpreter would never recognize a "wild loop" as distinct from a "while loop," but due to the constraints of the grammar we dealt with issues like this. In the future, we would hope to address this issue *with* a new grammar (i.e. a model of spoken syntax) for programming. This is similar to the approach by Begel and Graham [2][3] and Désilets [9].

7. Limitations

Our system design and evaluation were primarily limited to navigation and selection. Future work should also study other aspects of coding practices such as more accurate text generation given a limited coding syntax (especially punctuation) and better code snippet generation for editing than we were able to achieve. VocalIDE is both functional and feasible, but the improvements introduced in VocalIDE were mostly small innovations (like CCE and smart snippet generation to allow filler text edits). It appears there are many opportunities for innovation in this space going forward to support greater future accessibility in coding and text editing. More work is needed to assess and claim that VocalIDE addresses limitations faced by those with upper body limb impairments. But our quantitative and qualitative results are cautiously positive, and we are optimistic that the system can be expanded upon to address the challenges uncovered by the study. The current system remains limited by insufficient speech recognition accuracy and by the clumsiness of command timing, but this is a broader challenge in ASR. Command timing may be addressed through multi-modal input. For instance, incorporating a binary input device for issuing commands (*e.g.*, an easy to press button to indicate when a user is issuing commands *vs.* not issuing commands).

8. Conclusion

In this paper, we introduced VocalIDE, a system supporting vocal programming and demonstrated that it is feasible to use the system for programming. We contributed an empirically designed vocal programming system (one of the first), as well as a study of its usability and potential for individuals who have upper limb mobility impairments. We believe that VocalIDE and systems like ours can make programming and general computer use more accessible to anyone, regardless of physical ability.

9. ACKNOWLEDGMENTS

We thank our participants for enabling us to perform this research. We also thank CLASS for their support in our research.

10. REFERENCES

[1] Arnold, S. C., Mark, L., & Goldthwaite, J. (2000, November). Programming by voice, VocalProgramming. In Proceedings of the fourth international ACM conference on Assistive technologies (pp. 149-155). ACM.

[2] Begel, A. and Graham, S. L., (2005) Spoken programs,In Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), 2005, pp. 99-106.

[3] Begel, A., & Graham, S. L. (2006). An assessment of a speech-based programming environment. In Visual

Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on (pp. 116-120). IEEE.

[4] Brandt, J., Dontcheva, M., Weskamp, M., and Klemmer, S.R. *Example-centric programming: integrating web search into the development environment*. ACM, New York, New York, USA, 2010.

[5] Brandt, J., Guo, P.J., Lewenstein, J., Dontcheva, M., and Klemmer, S.R. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. ACM (2009), 1589–1598.

[6] Brault, M.W. July 2012. *Americans With Disabilities: 2010.* U.S. Department of Commerce.

[7] Curatelli, F. and Martinengo, C. (2010) Enhancing digital inclusion with an 9nglish pseudo-syllabic keyboard. *HCI in Work and Learning*, (2010).

[8] Dai, L., Goldman, R., Sears, A., and Lozier, J. Speech-based Cursor Control: A Study of Grid-based Solutions. *SIGACCESS Access. Comput*, 77-78 (2003), 94–101.

[9] Désilets, A. (2001). VoiceGrip: a tool for programming-by-voice. International Journal of Speech Technology, 4(2), 103-116.

[10] Désilets, A., Fox, D. C., & Norton, S. (2006, April). VoiceCode: an innovative speech interface for programming-by-voice. In CHI'06 Extended Abstracts on Human Factors in Computing Systems (pp. 239-242). ACM.

[11] Feng, J., & Sears, A. (2004). Using confidence scores to improve hands-free speech based navigation in continuous dictation systems. ACM Transactions on Computer-Human Interaction (TOCHI), 11(4), 329-356.

[12] Findlater, L., Moffatt, K., Froehlich, J.E., Malu, M., and Zhang, J. Comparing Touchscreen and Mouse Input Performance by People With and Without Upper Body Motor Impairments. ACM Press (2017), 6056–6061.

[13] Gorter, J.W., Rosenbaum, P.L., Hanna, S.E., et al. Limb distribution, motor impairment, and functional classification of cerebral palsy. *Developmental Medicine and Child Neurology 46*, 7 (2004), 461–467.

[14] Harada, S., Landay, J.A., Malkin, J., Li, X., and Bilmes, J.A. The Vocal Joystick:: Evaluation of Voice-based Cursor Control Techniques. ACM (2006), 197–204.

[15] Harada, S., Wobbrock, J.O., and Landay, J.A. Voicedraw: A Hands-free Voice-driven Drawing Application for People with Motor Impairments. ACM (2007), 27–34.

[16] Holmes, R. and Walker, R.J. Systematizing pragmatic software reuse. *ACM Transactions on Software Engineering and …*, (2012).

[17] IGDA. October 2005. Game Developer Demographics: Exploration of Diversity. U.S. Department of Commerce.

[18] Jackson, J., Cobb, M., and Carver, C. Identifying top Java errors for novice programmers. *Frontiers in Education*, (2005).

[19] Johansson, V. 2008. Lexical diversity and lexical density in speech and writing. Working Papers 53. 61-79 pages.

[20] Kim, M., Bergman, L., and Lau, T. An ethnographic study of copy and paste programming practices in OOPL. ... *Engineering*, (2004), 83–92.

[21] LaToza, T.D., Venolia, G., and DeLine, R. Maintaining mental models: a study of developer work habits. (2006).

[22] Leopold, J. L., & Ambler, A. L. (1997, September). Keyboardless visual programming using voice, handwriting, and gesture. In Visual Languages, 1997. Proceedings. 1997 IEEE Symposium on (pp. 28-35). IEEE.

[23] MacKay, D. Dasher–an efficient keyboard alternative. *Advances in Clinical Neuroscience and Rehabilitation*, (2003).

[24] MacKenzie, I.S. and Zhang, S.X. *The design and evaluation of a high-performance soft keyboard*. ACM, New York, New York, USA, 1999.

[25] Mathiowetz, V., Volland, G., Kashman, N., & Weber, K. (1985). Adult norms for the Box and Block Test of manual dexterity. *American Journal of Occupational Therapy*, *39*(6), 386-391.

[26] Maulsby, D., Greenberg, S., & Mander, R. (1993, May). Prototyping an intelligent agent through Wizard of Oz. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems* (pp. 277-284). ACM.

[27] Nicolau, H., Guerreiro, J., and Guerreiro, T. Stressing the Boundaries of Mobile Accessibility. 2014.

[28] Ruan, S.,, Wobbrock, J.O., Liou, K., Ng, A., and Landay. J.A. (2016) Speech Is 3x Faster than Typing for English and Mandarin Text Entry on Mobile Devices. arXiv:1608.07323 {cs.HC} (Aug. 2016).

[29] Sears, A., Feng, J., Oseitutu, K., & Karat, C. M. (2003). Hands-free, speech-based navigation during dictation: difficulties, consequences, and solutions. Human-computer interaction, 18(3), 229-257.

[30] Sears, A., Revis, D., Swatski, J., and Crittenden, R. Investigating touchscreen typing: the effect of keyboard size on typing speed. *Behaviour & ... 12*, 1 (1993), 17–22.

[31] Singer, J., Lethbridge, T., Vinson, N., and Anquetil, N. (1997). An examination of software engineering work practices. In Proceedings of CASCON 97, Toronto, ON, NRC, Ottawa, pp. 209–223.

[32] Suhm, B., Myers, B., and Waibel, A. Multimodal error correction for speech user interfaces. *ACM Transactions on Computer-Human Interaction (TOCHI) 8*, 1 (2001), 60–98.

[33] Trewin, S., Swart, C., and Pettick, D. Physical Accessibility of Touchscreen Smartphones. ACM (2013), 19:1–19:8.

[34] Vertanen, K. and MacKay, D.J.C. *Speech dasher: fast writing using speech and gaze*. ACM, New York, New York, USA, 2010.

[35] Wagner, A., Gray, J., (2015) An Empirical Evaluation of a Vocal User Interface for Programming by Voice, International Journal of Information Technologies and Systems Approach, v.8 n.2, p.47-63.

[36] Wagner, A., Rudraraju, R., Datla, S., Banerjee, A., Sudame, M., & Gray, J. (2012, May). Programming by voice: A hands-free approach for motorically challenged children. In CHI'12 Extended Abstracts on Human Factors in Computing Systems (pp. 2087-2092). ACM.

[37] Wobbrock, J. and Myers, B. *Trackball text entry for people with motor impairments*. ACM, New York, New York, USA, 2006.

[38] Wobbrock, J. O. (2014). Improving pointing in graphical user interfaces for people with motor impairments through ability-based design. In *Assistive Technologies and Computer Access for Motor Disabilities* (pp. 206-253). IGI Global.

[39] Wobbrock, J.O. and Myers, B.A. From letters to words: efficient stroke-based word completion for trackball text entry. ACM, New York, New York, USA, 2006.

[40] Wobbrock, J.O., Kane, S.K., Gajos, K.Z., Harada, S., and Froehlich, J. Ability-Based Design. *ACM Transactions on Accessible Computing 3*, 3 (2011), 1–27.

[41] Wobbrock, J.O., Myers, B.A., and Kembel, J.A. EdgeWrite: a stylus-based text entry method designed for high accuracy and stability of motion. ACM, New York, New York, USA, 2003.

[42] Wobbrock, J.O., Myers, B.A., Aung, H.H., and LoPresti, E.F. Text Entry from Power Wheelchairs: Edgewrite for Joysticks and Touchpads. ACM (2004), 110–117.