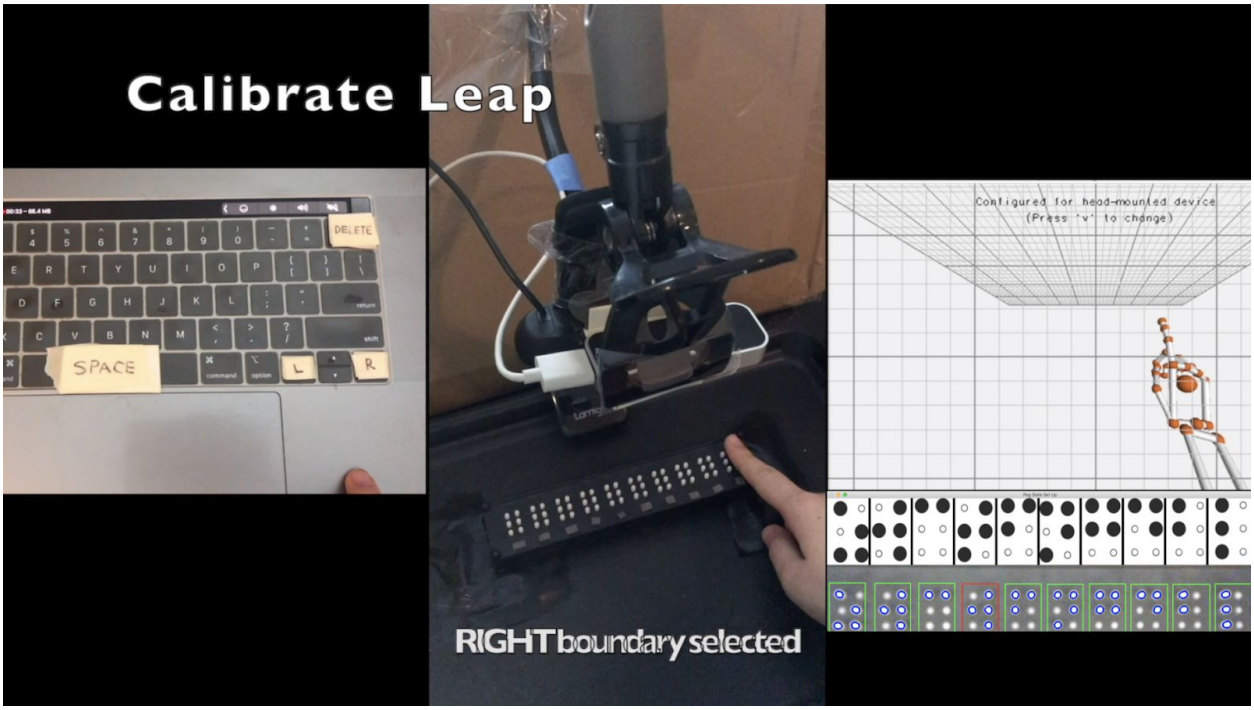

Braille E-Learner

Sabina Chen

The Braille E-Learner is a tactile, auditory, and visual multimodal learning system that teaches users Braille through an interactive feedback loop. The motivation for designing this system is to enable blind users to learn Braille by themselves, without the need for an instructor on the side.

I built a system that helps beginner users learn how to read and memorize new Braille symbols. The system first auto-generates new symbol configurations for the user to interact with, and then provides them opportunities to learn or review the new characters. Built-in configuration modules enable users to set up their environment independently.

I used a peg slate to enable tactile interactions with Braille symbols, a Leap Motion and the Leap Motion SDK to track finger positions, a Logitch webcam and OpenCV to detect symbols on the peg slate, the PyPI Speech Recognition library to recognize voice commands, Pygame for internal state control and keypress detection, and AppleScript to vocalize verbal commands.

Finger tracking is accurate but difficult to initially recognize. Speech recognition is unreliable and delayed. Symbol recognition is reliable and accurate. Verbal cues and system settings are consistent. And the general integrated system is fairly responsive.

Motivation

There are not many beginner-friendly tools for blind students to learn Braille by themselves. Traditionally, Braille is taught in-person by a Teacher of Students with Visual Impairments (TVI), or can be learned using textbooks for those with vision. However, students who are blind do not have many options for self-learning introductory Braille.

Enabling users who are blind to self-study Braille is not widely advertised or developed for. In-person learning is often preferred over self-studying due to the ability to interact with the instructor and get real-time feedback, something that is difficult for electronics to provide naturally. Therefore, an interesting question now arises: can an electronic learning device be developed to enable beginner students to self-learn Braille in an effective, interactive manner?

I propose the Braille E-Learner, a tactile, auditory, and visual multimodal learning system that teaches individuals Braille through a real-time, interactive feedback loop. This system can be used by both blind and sighted individuals alike, and enables users to learn Braille by themselves without the need for a physical instructor on the side or the use of textbooks.

System

Overview

The Braille E-Learner helps users learn and review new Braille symbols. As the user interacts with a peg slate, the system provides appropriate responses to aid the student in their learning. The system includes three primary modules: **Calibration**, **User Modes**, and **Settings**. Users use a resettable, plastic peg slate for tactile interactions, and keyboard buttons to navigate between different sub-modules.

Physical Setup

The peg slate of 10 braille cells is placed in the middle of a black non-reflective surface. A leap motion controller mounted above tracks the position of the user's index finger as it moves across the peg slate. A camera, also mounted above, detects the symbols created by the raised pegs. A laptop is used to process input and control state information. Four primary buttons on the keyboard enable navigation between different system modes. The physical setup for the electronics is depicted in Figure 1.

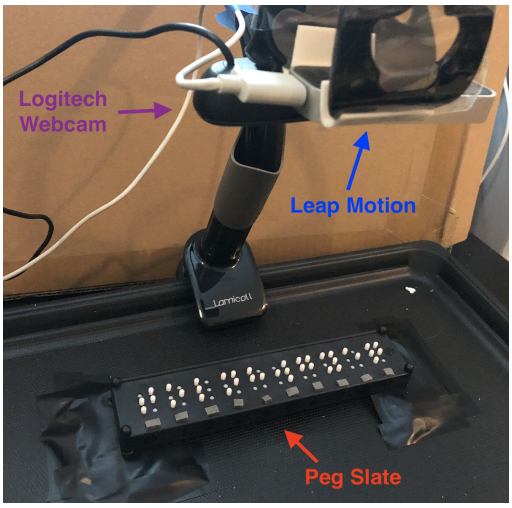

Figure 1. Physical hardware setup

System Usage

The system comes in three primary modules: Calibration, User Modes, and Settings. The internal state logic that connects these modules is depicted in Figure 2.

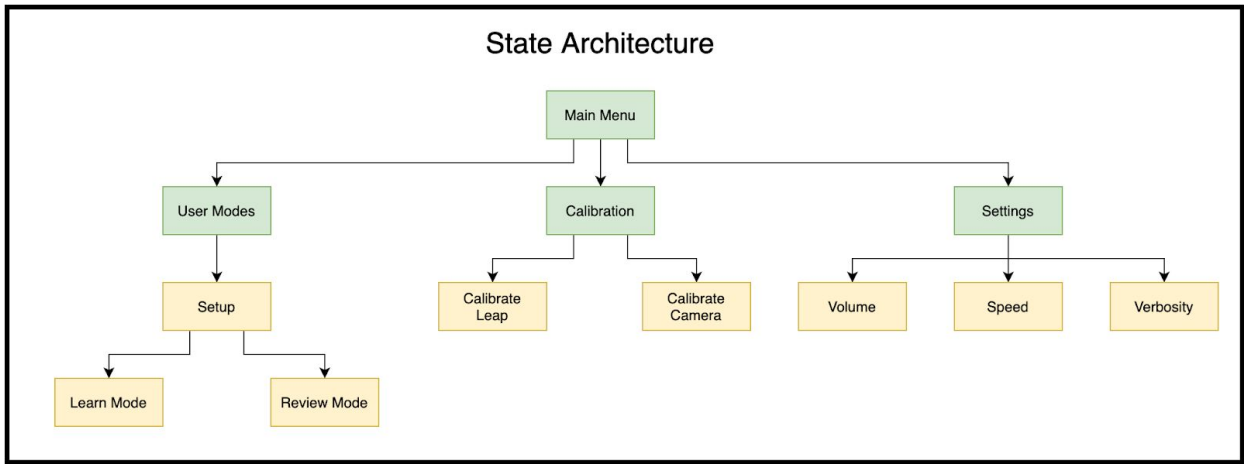

Figure 2. Internal state logic for individual control modules

The **Calibration** module includes two modes: Calibrate Leap and Calibrate Camera. These modes help ensure that the leap motion, camera, and peg slate are placed correctly at startup. Without proper calibration, the leap motion and camera won't be able to track the user's finger positions or peg symbols appropriately. Ideally, with a permanent, non-moving setup, the calibration module can be eliminated entirely. However, even though the majority of the physical setup is duct taped together, various physical parts still move, so calibration is necessary. Initially, this was a manual script used only for the developer (me) to calibrate system parameters. However, in the final implementation, it was included as a module for the user to access in order to eliminate the need for developer intervention. This ensures that the user can manage the system independently themselves, and will not need to ask another person for help, or need to interact with the code, mouse, or laptop monitor at any point.

The **User Modes** module includes three modes: Setup Mode, Learn Mode, and Review Mode. Setup Mode is an optional mode that auto generates a random Braille symbol order for the user to study. It then guides the user in setting it up on their own peg slate via step-by-step verbal instructions. After setup, Learning Mode then helps the user learn the names of new Braille characters. It does so by automatically voicing the symbol name whenever the user hovers their index finger over a new cell. Finally, once the user has confidently learned the symbol names, they can test their knowledge in Review Mode. During Review Mode, the user provides guesses for symbol characters, and the system responds with whether the guess was correct or not.

The **Settings** module includes three options: Volume, Speed, and Verbosity. This module was developed for user customization. The user can increase or decrease the volume and speed settings of the speech system using simple keypresses. There is also an option to toggle verbosity for more familiar users. When verbosity is toggled on, all instructions are verbalized, so that beginner users know how to use the system. When verbosity is toggled off, the system assumes user familiarity, and no longer verbalizes repeat instructions. By modifying these settings, advanced users will be able to navigate between system modules fairly quickly, just by increasing speech speed and turning off verbosity.

System Design

Overview

There are four inputs: **Finger Recognition**, **Peg Recognition**, **Voice**, and **Keypresses**. And two outputs: **Sound** and **Images**. Inputs are processed by a central Python control script, which keeps track of the internal module state of the user, and then outputs the appropriate responses. Sound is output from the laptop speaker system, and Images are shown on the laptop display. The system can communicate with the user solely through Sound output. Images are only available for ease-of-use during the Setup and Calibration modes for sighted users. Altogether, the primary software libraries used in the system include: Leap SDK, PyPI Speech Recognition (Google Cloud Speech API), OpenCV, PyGame, and AppleScript. The overall system architecture is depicted in Figure 3.

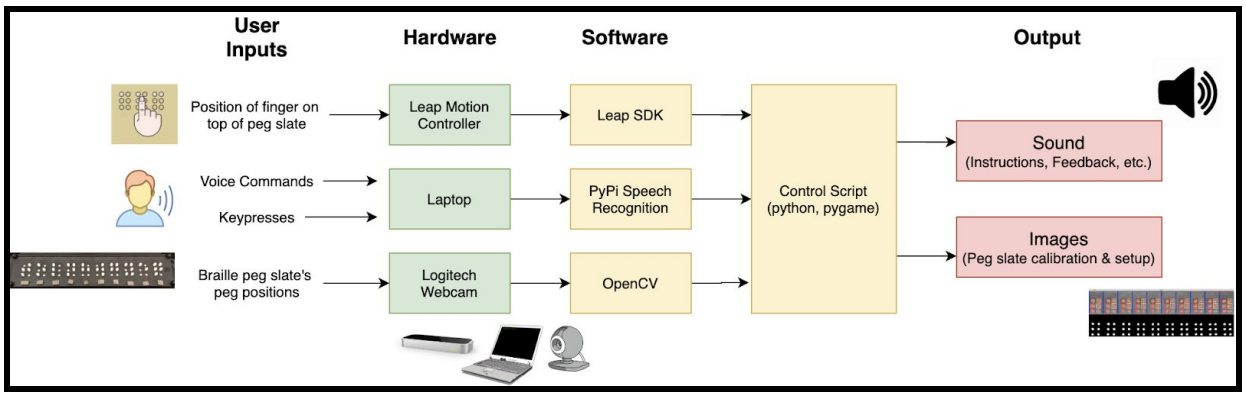

Figure 3. System architecture

Finger Recognition (input)

A Leap Motion Controller mounted above processes hand poses and motion. The Leap SDK can detect whether the hand is left or right, palm up or palm down, whether the fingers are extended, as well as where the locations of the bones are. I wrote a function that tracks the *tip* of the user's right index finger, only when it is *extended* and the palm is facing *downwards*. This specific setup ensures that the system only responds to conscious movements of the index finger by the user.

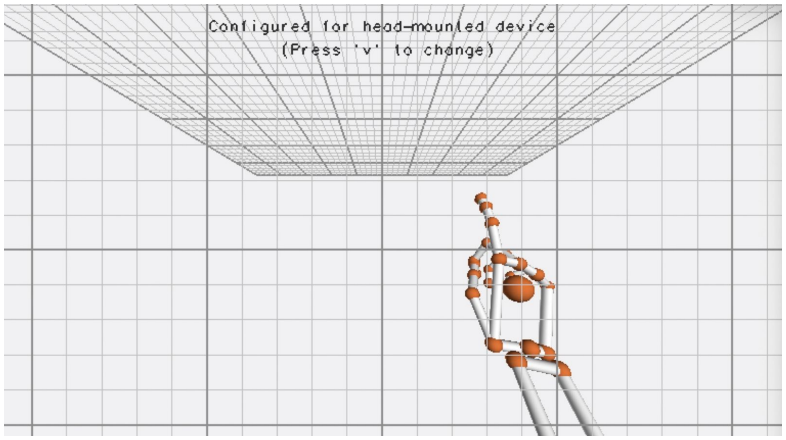

Figure 4. Leap Visualizer's head-mounted view of the right hand

Navigation (input)

Keypresses are detected using the PyGame library. Whenever a button on the keyboard is pressed or released, an event is triggered in the code, which lets us check if it is a desired key. The PyGame library was originally designed for writing video games. However I used it for overall state navigation and integration due to its effective threading capabilities and ability to debounce button presses. I integrated all the individual modules and software libraries using PyGame as the glue.

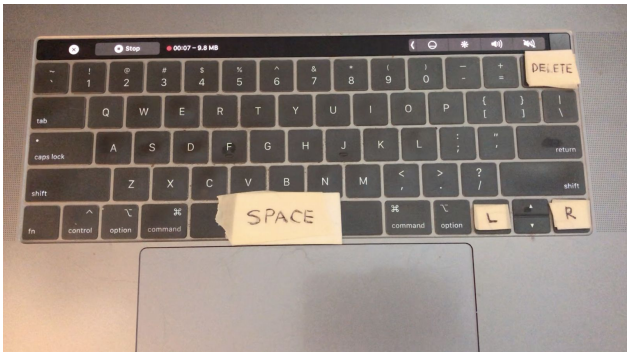

Figure 5. Relevant buttons on the keyboard

Peg/Symbol Recognition (input)

A Logitech webcam mounted above detects which symbols are currently set up on the peg slate. Using OpenCV, the camera image can be analyzed to determine whether a peg has been raised or not. The contour area of a peg is larger when it is raised, compared to when it is *not* raised. By calibrating, cropping, and thresholding the image input using OpenCV, I can determine the locations of the raised pegs with respect to each Braille cell, and then consequently figure out the symbol names.

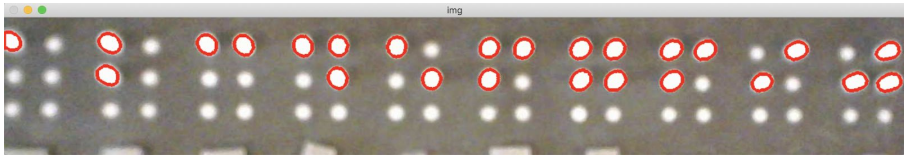

Figure 6. Contouring for raised pegs

New Symbol Order Generation (output)

Images of Braille symbols are pre-saved as *.png* files. To randomly generate and display images of new symbol orders, I use a combination of Python, Numpy, and OpenCV to concatenate a random order of 10 pre-saved images together for the user to view during Setup Mode.

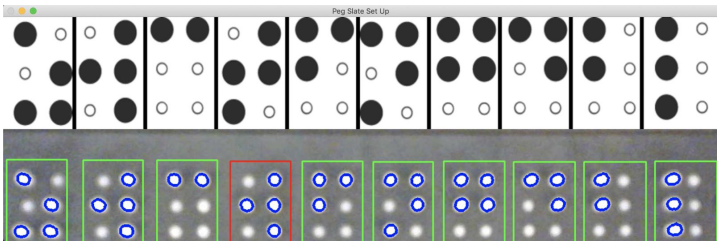

Figure 7. Concatenated *.png* of Braille symbols (above)
System checks if peg slate matches the generated symbols (below)

Voice (input)

During Review Mode, the user can communicate with the system by speaking into the computer microphone. I use the PyPI Speech Recognition library to convert speech to text. I tried using both the Google Cloud Speech API and Sphinx API. While the Google Cloud Speech API requires the internet to use, it is more accurate than the Sphinx API. However, the Sphinx API can be operated offline, which is beneficial when the internet is slow or unreliable. Both APIs have been implemented and can easily be toggled in the parameters file based on the needs of the current environment.

Sound (output)

The system primarily communicates to the user via verbal cues. Using MacOS's built-in AppleScript and *os* commands, I can convert any text string into speech, which is then outputted through the laptop speaker. The speed and volume settings are also manipulated using AppleScript.

System Evaluation

Overview

Transitioning between separate modules using keyboard commands is effective. The camera is also reliable under varying lighting conditions. The main limitations stem from gesture (leap) recognition and speech recognition.

Finger Recognition (input)

Finger recognition is used during Learn Mode and Review Mode. Once the Leap motion recognizes the hand, it has no issues keeping track of hand or finger positions. However, the difficulty lies in getting the Leap to *initially* recognize the hand. When I place my *right* hand under the leap, it often mistakes it for my *left* hand.

Usually two methods can resolve the issue: repeatedly removing/putting back my hand in frame until the leap recognizes it correctly, or wiggling/moving the fingers around. I have attempted multiple strategies to fix this issue on the physical side, including: removing reflective items, removing obstacles, blocking external light, increasing the field of view, not wearing sleeves, as well as showing more of my hand and arms when it is in frame. However, the inability to correctly identify my right hand *immediately* is still a consistent issue that I hypothesize stems from a combination of leap motion and environmental issues.

Once the correct hand and finger positions are determined, however, the probability of the Leap de-identifying it again is minimal (unless the user removes the hand from frame). To help users with initial hand detection, I added a timeout that verbally recommends that the user reset their hand positions if the system consecutively identifies the incorrect hand for a set number of seconds. This lets the user know that the leap motion is having trouble identifying their hand and gives them suggestions on fixing it.

Navigation (input)

Detecting keypresses using PyGame has been fairly smooth and reliable. I have had no issues with it so far! The system can immediately tell when a key is pressed and respond accordingly with no delays.

Peg/Symbol Recognition (input)

Using OpenCV to determine whether a peg has been raised, using color thresholding and contour size has been reliable. I was originally concerned that since this technique relies on color, variable lighting conditions would affect performance. However, unless the lighting conditions are either *extremely* bright or *extremely* dark, since the peg slate itself is already black-and-white, the color contrast has been consistent. One concern would be if the pegs were halfway pushed up, instead of all the way, thereby reducing the contour size, and then not getting recognized. That is something the user will need to fix by themselves. Further tuning the current threshold parameters might overfit to specific lighting conditions.

New Symbol Order Generation (output)

Using Numpy and OpenCV to generate, concatenate, and display images has been reliable. Since I was originally familiar with these libraries already, the tasks were fairly straightforward to implement.

Voice (input)

Voice input has been a consistent issue for me since the beginning of the project. This functionality is only used during Review Mode. Using the PyPI Speech Recognition library for speech-to-text conversions, sometimes the library hangs indefinitely, freezing the code while trying to parse my input (relevant to both Google and Sphinx API). Sometimes after a short (or long) delay, it will finally produce an output. Other times I am forced to reset the system. I suspect that this is either a microphone, library, or a personal bug in my code. I have attempted to switch microphones, but the same issue persists. Potential fixes include investigating the library more and tuning their default parameters, or just switching to another speech-to-text library altogether. When it *does* work, there is also always a delay in response.

Sound (output)

Using MacOS'es built-in AppleScript for sound generation and speech setting modifications is reliable. The primary drawback is that this means the script will not work on other operating systems, and will require modification to the source code. However because the primary function is written in a simple utility script, it shouldn't be too difficult to convert to.

Design and Implementation

Initial Impressions

My primary goal in designing this system was that I wanted to develop a product that can be used by both blind and sighted users alike. Meaning that all of the functionalities must be usable and logical when relying solely on auditory and tactile cues. Early on, I knew that not being able to rely on visual cues would be a challenge, since sight is something that I use, and take for granted every day. This means that I often had to stop, take a step back, and review my predisposed assumptions, in order to make sure that my designs were suited to the blind user's experience. Therefore when testing my implementations, I would often close my eyes in order to simulate what the user would be experiencing. This would often bring up many usability issues that I would then gradually improve upon after each iteration.

Design Considerations

On the software component side: First, verbal cues must be clear and short, yet descriptive. The system must be able to be used from start-to-finish without developer intervention or the need for sight. Furthermore, the product should not rely on the mouse or monitor, nor need to modify the code directly in any way. The user interface also needs to be usable by both beginners and advanced users alike.

On the hardware component side: I wanted to ensure that each item that the user is interacting with can be easily distinguishable by touch. The user should be able to easily reach and differentiate between different buttons and tactile objects. Furthermore, navigating between different modes of the system should be fast, easy, and intuitive to use.

These design considerations were what ultimately lead me to the minimal 4 button and peg slate system, with primarily verbal cues for feedback, and personalizable speech settings.

One big lesson learned through the design process is that balancing *both* design considerations and technical constraints is difficult. The final product I was envisioning consisted of many moving parts and multiple complex modules that must all come together to be used intuitively and smoothly. I eventually realized that the best way to tackle all of these design and technical considerations was to consider each one at a time, then put it together for the big picture. However, as with any user-focused product, there can always be new improvements to the user experience with each new iteration.

First Redesign (Hardware-focused)

The first major system redesign was hardware focused. The first time I assembled the hardware, I only used items I found at home. The leap motion was haphazardly taped to the side of a box, and the peg slate was free roaming. The result was that there were too many obstructions in the way that caused the leap motion to be rendered almost unusable. I even attempted to have a nearby friend test out my system, but we couldn't even get the system to recognize my friend's hand even once. After this first failed "user test", I decided to completely scrap the current hardware setup, and purchased the items I needed to make a more permanent and reliable setup.

From all of these failures, I learned a lot about the leap motion that I otherwise wouldn't have known or tried to understand before. For example, the leap motion uses an IR camera with LEDs, producing black and white images that can then be used to create an internal model of the hand. Therefore obstructions to the lense, even partially, can cause inaccurate readings and incorrect hand models. Furthermore, some surfaces are more reflective to IR light than others. For example, did you know that clear scotch tape shows up as white in the IR image, while black electrical tape shows up as black? Therefore, the result of the redesign was that the majority of the viewable surface below the mounted leap motion needed to be clear of any obstructions, and mostly made out of black non-reflective materials.

Another shortcoming I realized was when I attempted to test the system myself by obstructing my sight. Because the pegs were so far apart, sometimes it was difficult to know when a braille cell started or ended, especially when you can't see. Therefore I added soft pads at the bottom of each braille cell, to make it easier to differentiate between cells. I further added soft pads to the computer keys I would be using to check that the keys were easy to access and press, when used in conjunction with the peg slate.

Second Redesign (Software-focused)

The second major system redesign came from the software side. After the Implementation Studio, one of my classmates connected me with a tactile technology consultant, who gave me a lot of good advice involving designing for blind users. The conversation got me to think about aspects of the design that I never thought to consider initially.

For example, originally, I was unconsciously focused on designing an audio-based interaction system, which I thought would be “cool” to have along with finger recognition. However, after speaking with the tactile technology consultant, I realized that the majority of screen readers and devices designed for the blind were tactile and keyboard-based. Keyboard-based interactions was definitely not something that I had even thought to consider initially, since it didn’t seem “innovative” enough. However, it turns out that for the purposes of product design for my specific application, keyboard navigation was a genuine option, since (1) the range of the leap motion was so small that, without the use of sight, users will have difficulty gesturing in the correct area, and (2) audio-based input commands was very unreliable in my current system. Ultimately, I switched from audio-based navigation to keyboard-based navigation. The switch to keyboard-based navigation further proved to be very beneficial for general ease-of-use and efficiency.

During our conversation, the idea of speech customization for advanced users (ie. volume, speed, verbosity) was also highlighted. Because blind users rely on verbal cues from electronics, they get very fast at listening to instructions and verbal cues. Therefore many users would appreciate having the ability to customize their listening speed and verbosity in order to more comfortably use the system.

Therefore the major redesigns for my project after the Implementation Studio involved the switch to keyboard-based navigation, as well as increased reliance on and personalization of the speech output.

Technical and Design Limitations

To be frank, people who read Braille do not read characters one cell at a time using only their index finger. Instead, they use both hands, and multiple fingers. The right hand has two to three leading fingers “reading”, while the rest of the right fingers and left fingers keep track of page position. The leftover fingers also may start looking ahead to begin reading the next line of text when the user gets near the end of a line of text. This mode of reading is efficient. However, for ease of implementation and proof-of-concept, I chose to only keep track of the index finger instead of multiple fingers at once.

Furthermore, for advanced users, it would be difficult to know exactly which finger is currently the “reading” finger in a multi-line text, since the “reading” finger switches around depending on where the user is on the line or on the page. So if I *were* to develop a system that can account for multi-line reading, this would be an interesting issue to look into, and would require a lot of user tests to configure. Perhaps investing in a multi-lined peg slate, or even just designing a system that can read symbols on actual Braille pages (instead of just the peg slate), would be an interesting challenge.

As of now, the major design limitation to the current system is how realistic and applicable the product is for learning how to read Braille. The current peg slate-only setup can definitely achieve its goal of tactilely training users to memorize individual words and symbols, but it isn’t too useful at teaching them *effective* reading strategies for multi-line text.

Future Work

Currently, the Braille slate includes 1 line of 10 cells, allowing for 10 unique Braille symbols to be made and parsed at a time. Symbols that can be learned include the 26 lowercase letters of the alphabet. Braille E-Learner comes in two primary modes: Learn Mode and Review mode, and also includes settings for device calibration and personalization. Symbols are randomly generated and learned individually. The system provides real-time, interactive feedback depending on user input.

Future work may include:

- Ability to learn phrases and words, as opposed to being limited to just letters
- Accommodation for different levels of reading (ie. contracted vs. uncontracted Braille)
- Multi-line reading
- Ability to detect Braille symbols on actual Braille paper
- Automatically refreshing the peg slate so that the user doesn't have to do it themselves (this may require building one myself, or purchasing a very expensive electronic device)
- Adding more modules (ie. interactive game modes)
- Bug-hunting (ie. fixing audio input, hand recognition issues, etc.)
- Further improving the usability via user testing

Lessons Learned

Prioritize core functionalities, and separate tasks into small achievable tasks. My initial vision for this project involved both hardware and software solutions. I wanted to develop not only an interactive software pipeline, but also an electronic peg slate that can automatically reset the peg positions without the user having to do so themselves. This would have involved having to build the peg slate completely myself. However, after receiving feedback regarding the feasibility and scope of the project, it was scaled down to focus on the main software pipeline using only existing electronics. I am thankful to have received this feedback early on. It was definitely difficult to scale back from the big plans I had, however, by doing so, and separating tasks into bite-sized chunks, I was able to implement more functionalities for the project than I originally thought I could. Every time a task or goal was achieved, it would encourage me to work towards the next goal, and spark even more ideas for improvement. Though this is not a new lesson, it is a good reminder to constantly focus on one achievable task at a time. Once the minimum product is built, *then* see if there could be improvements. Core functionalities should be a priority, any extra features can be added later. If a system doesn't work as-is, then the extra features are worthless.

Start early, fail fast, and then iterate. I am thankful that we had the studio assignments to keep us on schedule. Having consistent checkpoints and feedback helped me realize failure points early on. Also, because this project requires a very specific hardware setup, in order to make the project work, new hardware needed to be ordered that wouldn't arrive until weeks later. Luckily most of the physical hardware issues were discovered early on, but had I started working on the project later, I would have not been able to make as much progress. Resolving the majority of hardware issues early on allowed me to focus on software and usability in the later weeks. It's better to start early and discover issues sooner, than to have it pop up when it is too late.

Don't be afraid to ask for help. For the first 3/4ths of the project, I was designing a product for a community of individuals that I had never spoken to or interacted with before. I was designing a product that might not have even been useful to the community of users I was targeting. I did as much research

as possible on the needs of visually impaired individuals, reading Braille, and existing related assistive technologies, but I still lacked a fundamental understanding of what users really needed. Why didn't I just reach out then? Because I was scared. I was scared of being ignorant, and for potentially saying something that might be taken wrongly, or for being "politically wrong". It wasn't until after the Implementation Studio that I mentioned my worries, that a fellow classmate connected me to a Perkins contact who turned out to be incredibly patient, understanding, and willing to talk. After that talk, a large part of my project design was pivoted since I had a better understanding of what the users needed and wanted. A large part of the fear stems from being afraid of being ignorantly incorrect, and the best way to conquer that fear is to proudly face that ignorance and being willing to confront it and learn.

Code

Github: https://github.com/sabinach/braille-elearner (includes README)

Hardware

Peg Slate

- Paperless device designed to help teach beginning users of the braille slate.

Logitech C270 Webcam

- Camera used for peg and symbol recognition

Leap Motion Controller (2013)

- Optical hand-tracking module used for tracking finger positions

MacBook Pro 15-inch, 2018 (MacOSX Mojave 10.14.6)

- Computer used for running software and processing electronic inputs

Software Packages (Installation Instructions Available on Github README)

Leap Motion Python SDK (Desktop Version 2.3.1 for Mac)

- Interface with Leap Motion via Python

Pygame

- Designed for making video games
- Super useful for detecting keyboard inputs

PyPI Speech Recognition

- A cross-platform Python speech recognition library

- Accesses microphone

Google Cloud Speech API

- Speech-to-text converter
- Requires PyPI Speech Recognition library
- Requires internet
- Limited free to use (includes license)
- Pretty accurate

Pocket Sphinx API

- Speech-to-text converter
- Requires PyPI Speech Recognition library
- Works offline
- Not very accurate

OpenCV (Version 4.2.0)

- Computer Vision/Image Manipulation library
- Make sure to check the version number, earlier versions may have different function outputs

Numpy

- Useful for fast mult-dimensional array and matrix calculations in Python
- Useful to be used in conjunction with OpenCV

Pathlib

- Get file paths

osascript

- Wrapper for AppleScripts
- Enables automated control over computer system functionalities