Smart-Cane
Final Report

Baltazar Guerra L.
Jonathan Williams
Matthew Giuffrida
Arthur Helmen
Shawn Popal

5/1/2020

Contents

1 Executive Summary **4**

2 Introduction **6**
- 2.1 Problem Background . . . 6
- 2.2 Needs Statement . . . 6
- 2.3 Goal and objectives . . . 6
- 2.4 Design constraint and feasibility . . . 7
- 2.5 Literature and Technical Survey . . . 8
  - 2.5.1 SmartCane . . . 8
  - 2.5.2 UltraCane . . . 9
  - 2.5.3 Smart Stick . . . 9
  - 2.5.4 Sound and Touch Based Smart Cane . . . 9
  - 2.5.5 Robot-Assisted Indoor Navigation . . . 10
  - 2.5.6 Haptic Assistive Bracelets for Blind Skiers . . . 10
  - 2.5.7 Our Development Direction . . . 10
- 2.6 Evaluation of alternative solutions . . . 11

3 Final Design **12**
- 3.1 System description . . . 12
- 3.2 Design Specifications . . . 13
- 3.3 Approach for design validation and testing procedures . . . 16

4 Implementation Notes **17**
- 4.1 Obstacle Detection (Ultrasonic Sensors) . . . 17
- 4.2 Object Recognition . . . 21
- 4.3 Indoor Navigation Hardware . . . 23
- 4.4 Indoor Navigation Software . . . 25
- 4.5 iOS Application Software . . . 29

5 Experimental Results **34**
- 5.1 Camera Lenses . . . 34
- 5.2 Battery . . . 38
- 5.3 Ultrasonic Sensors . . . 39
- 5.4 IOS Application . . . 41
- 5.5 Indoor Navigation Hardware . . . 47
- 5.6 Indoor Navigation Software . . . 48

6 User Manual **49**
6.1 Raspberry pi3 setup 49
6.2 Indoor Navigation Hardware 50
6.3 Indoor Navigation Software 51

7 Course debriefing **52**
7.1 Team Management 52
7.2 Safety and Ethical Concerns 52
7.3 Testing Success and Real World Applicability 52

8 Budgets **54**
8.1 Purchased Items 54
8.2 Costs Summary 54
8.3 Manufacturing Costs 55

9 Future Plan **56**

10 References **57**

1 Executive Summary

Travel can be significantly more of a challenge for those with visual impairments or blindness than it is for the sighted. Specifically, obstacle detection and navigation are more difficult when the sense of sight cannot be relied on in environments where that capability is assumed. A low tech method of easing this difficulty is to use a traditional white cane. However, this type of cane, which is purely mechanical, cannot fully address the needs presented by travel and navigation. The goal of this project is to create a "smart cane" - a device that attaches to a white cane and provides the additional functionality of indoor and outdoor navigation, orientation, smartphone connectivity, and obstacle detection. The hope is to make travel safer, more efficient, and more transparent for people with visual impairments or blindness, and allow them to have a better understanding, and stronger sense of confidence in their understanding, of the area around them as they travel, both in indoor and in outdoor environments.

The product is a cylinder, roughly 8 inches in length and 3 inches in diameter. It utilizes a combination of a camera and an ultrasonic sensor to detect obstacles that may be approaching and warn the user. The device utilizes a gyroscope to determine the user's orientation with respect to an 'anchor point' that they set, and provides feedback back to them about their orientation on request. The user can chain those anchor points to create a 'trail of breadcrumbs' to follow, allowing for more precise indoor navigation. A Bluetooth connection is made to the smartphone of the user, and allows the user to provide additional input to the device via an app on their phone. The phone's GPS sensor will be utilized to allow for outdoor navigation that synchronizes with the output methods provided by the device. Specifically, feedback from the device is provided in either haptic or audio form, depending on the user's preferences.

Unfortunately, due to limitations resulting from the pandemic, a connected, integrated final product was not able to be constructed. Instead, the device was broken down into a number of subsystems, specifically, the indoor navigation software, the sensors (ultrasonic, gyroscope/accelerometer), haptic motors, the iOS application, and the camera. Each of those subsystems functions independently within the parameters set by our requirements, and are designed to work with the outputs of the other subsystems to simulate an integrated product. Overall, our results were fairly successful. Each subsystem performs up to the standards we originally set. We weren't able to test an integrated, complete system, but to the extent we were able to evaluate the final product, it succeeded in our original goals.

The experience of working on this project was overall a good one. There was

no issue with the division of the work, and communication was easy and consistent. The strengths of the members complemented each other, and so we were able to accomplish what we set out to do. There was space for discussion without leading to conflict, and so we were able to evaluate a number of possible avenues of pursuit for each issue we ran into. In general, our team was on the same page and functioned well.

2 Introduction

2.1 Problem Background

Blindness and visual impairment present natural complications to travel for those who have those conditions. Obstacle detection in particular is an area of concern, as could be expected. One of the most common tools used by those individuals who have visual impairments to assist in travel is a specialized cane, usually referred to as a "white cane" in the United States. These are long canes, usually capable of folding up for ease of storage. They are generally white, as the name indicates, for both ease of visibility for others, and to identify the user as blind or visually impaired. The canes are used by the individuals to identify potential obstacles through sweeping the cane in front of themselves, as well as a form of surveying the environment around them – for example, feeling the bumps on a road that indicate a crossing is there. These canes generally are purely mechanical, with no electronic components.

This led to our proposal to create a "smart cane" – a device that acts as a white cane, while also leveraging technology to add to or enhance the basic functionality of the traditional, mechanical cane. The primary goal is to create a device that keeps all of the functionality of a traditional cane, while adding additional capabilities to the core functionality of the cane – that is, to make safe, efficient travel easier for those with blindness or visual impairments.

2.2 Needs Statement

In order to have a better understanding of the problems we need to tackle we spoke to Justin Romack who works at Texas A&M Disability Services (and is blind) so he could give us some insight of his everyday struggles and those of his peers. Through this conversation and separate research, a list of areas of need was identified. Those primary issues are: not being able to detect obstacles that are farther away from their cane or overhead, losing track of one's spatial location inside of a room or building, indoor and outdoor navigation, and heavy equipment hurting one's wrist and arm after some time.

2.3 Goal and objectives

The overall goal of the project is to make it easier and safer for people with visual impairments to travel through everyday environments. To that end, a list of goals was compiled below.

- Current market solutions are very expensive and lack certain quality of life functionality. The product should be relatively low cost.
- Users often become very accustomed to the canes that they use. The product should be able to attach to a variety of existing canes.
- White canes cannot generally detect obstacles at or above chest level. The product should be able to do so.
- The product should be able to connect to, and communicate with, a user's smartphone.
- The product should be able to quickly and easily allow the user to contact emergency services.
- The product should be able to provide feedback through audio and haptic channels.
- The product should be able to provide directions in an outdoor or indoor environment

2.4 Design constraint and feasibility

There are four primary constraint areas. First, power. The device must be portable in order to perform its function, and therefore must have its own power source with enough charge to be useful to the user. Second, the weight. White canes are designed to be constantly moved around, in order to "feel" the environment. If the product is too heavy, it will quickly tire the user to be moving it around constantly. Since preserving the original functionality of the cane is a key requirement, the device must be lightweight enough to not be a burden on the arms or wrists of a user. Third, the size. If the device is too bulky, it can get in the way of the user or make the cane difficult to store. Also, a bulky device would likely have a poor aesthetic, and therefore make the user less willing to use the device. Fourth, the function. The device must speak to the key needs identified previously – namely, obstacle detection, navigation, and orientation, among others.

Using these constraints, a list of requirements was drawn up, and can be seen in the table below.

The product at its core is feasible. The hardware requirements are within budget, and components that are capable enough for our needs have been identified. The largest possible issue is in fitting the performance of the product into the maximum

Figure 1: Product Requirements

Marketing	Engineering	Rationale
Must run for at least 4 hours without charging	A portable lithium battery should provide sufficient battery life	This is a device that must be portable, and therefore must have a decent battery life.
Must be 2.5 pounds or lighter	Total sum of device weight should fall within those parameters.	The user will need to swing the cane back and forth, and too heavy a device could make that tiring.
Must not exceed the volume of a cylinder with a diameter of 4 inches, and a height of 8 inches	The RasPi 3 is the largest planned component and it falls within those parameters.	Must be able to be held in an average hand
Must be able to attach to a cane	An adjustable clamp will allow a range of canes to be attached.	Users do not want to have to give up the canes they are familiar with.
Must connect to a smartphone	A Bluetooth sensor will allow communication with a smartphone with Bluetooth capabilities.	This allows us to better integrate with existing systems the users are familiar with.
Must be able to communicate through audio and haptic feedback	Small motors run by the Pi will provide haptic feedback, while a speaker will be attached to provide audio feedback.	Different users have different preferences that must be accounted for
Must find directions to a desired location	The GPS sensor should allow integration with existing maps.	This is a core area of need for the userbase.
Must detect obstacles in a 3-foot radius of the device at a minimum of 10 feet in front of the user	Input from a camera will be fed into image recognition software to detect obstacles in the desired range.	Obstacle detection is the main function of a cane, the goal of the project being to augment that device.
Must track orientation and location within a margin of 3 feet indoors (with respect to an 'anchor' point)	A gyroscope and accelerometer will be attached to track those parameters	This is critical in particular for indoor navigation, a key area of need.

tolerances we have identified. In particular, the area that is both the most difficult and the largest possible differentiating factor for the product is the indoor navigation. While we have done enough research to believe that reaching our goal is feasible, it is the most likely of the functionalities present to present the most difficulties in performing consistently within our margin of error.

Another area of potential difficulty is the image recognition system used to detect overhead obstacles. This system has a great deal of complexity in order to address the complex problem it is aimed at solving, and thus has many possible places for the tolerances to start becoming strained.

2.5 Literature and Technical Survey

2.5.1 SmartCane

This is a device that is able to be fitted onto the end of a traditional collapsible cane. It acts as a handle, and can detect obstacles from knee to head height. It has different detection range settings, one for close range (1.8m), and one for long

range (3m), which is intended to help the device work in different environments (indoors, outdoors, crowds). It has a high sensitivity, their website says it can detect an 3cm wide object from 3m away. It uses non-localized vibration feedback in the handle to inform the user of obstacles. This device is available for $90. This is the most similar product we found to what we intend to complete. Although we hope to include additional functionalities, like indoor navigation, and localized vibration feedback. [1]

2.5.2 UltraCane

This device is a commercial product that is currently available for purchase. This cane is meant to replace the traditional collapsible cane often used by people who have visual impairments. It uses two ultrasonic sensors for obstacle detection. One of these detects obstacles waist-down, and they claim it can detect obstacles 2-4m ahead of the user. The second sensor detects waist-up obstacles, with a range of up to 1.5m ahead/above the user. There are also two vibrating thumb buttons that relay obstacle detection to the user, the buttons vibrate to indicate direction, and the frequency they vibrate at indicate proximity. The cane is fairly portable, it can telescopically retract up towards the handle, but it ends up being less compact than a normal collapsible cane. It is sold for $762.20, which is more expensive than we would like our product to be (should it go into commercial production). [2]

2.5.3 Smart Stick

This is a research product that combines a smart cane with an android application. The cane uses three ultrasonic sensors to detect obstacles, as well as potholes, with a 400cm range. The cane uses an arduino to do the distance calculations, then uses the android app to give text-to-speech feedback to the user. The user can also use the application to call emergency contacts, by drawing and assigned gesture onto the screen the app can call the appropriate contact. A destination can also be input, it does not provide navigational instructions, it only alerts the user once they have arrived. [3]

2.5.4 Sound and Touch Based Smart Cane

This was a research project that didn't actually assemble the cane, but they did test all the sensors they intended to use. This project used ultrasonic sensors to detect both stationary, and moving objects. If an obstacle was detected close (d $<$ 50cm) the user a "stop" command would be verbally communicated through an

android app. If an object was far (50cm<d<100cm), then an "object" command would be communicated. If a moving object was detected moving towards the user, then the handle would vibrate- intensity of the vibration would indicate the speed of the object. A heat sensor was also added to detect hot objects if the user placed the tip of the cane onto the suspected hot object. The idea of detecting moving objects is a good function to add to a smart cane, I don't believe an ultrasonic sensor is appropriate for this, it's unable to detect an object and make sure it is tracking the same object as it moves towards the user. [4]

2.5.5 Robot-Assisted Indoor Navigation

A research project that used a robot that interacted with RFID tags, set up in an indoor space, that a visually impaired person could use to be guided through a building. The robot was successful in its task, but had trouble re-routing a user if there was an obstacle blocking its planned path to an RFID tag. The use of RFID is interesting, and this was one of the only projects we found that attempted to guide a visually impaired person through an indoor area. The problem is that an RFID network must be in place for the robot to interact with, a user wouldn't be able to take a device with such capabilities anywhere they wanted. The robot also used sensors to detect blockages, and attempted to move around the obstacle, but it wasn't able to find a new path unless there was another RFID tag it could detect. [5]

2.5.6 Haptic Assistive Bracelets for Blind Skiers

These researchers explored the use of vibration motors in a visually impaired person's ski poles, so an instructor could relay turning instructions from buttons on the instructor's poles. Normally, a visually impaired person has to ski behind an instructor and listen to voice commands, but these researchers were able to successfully guide their test subjects using haptic feedback. This is promising since we hope to be able to have haptic feedback present - not only for obstacle warning- but also for navigational cues. [6]

2.5.7 Our Development Direction

We hope to integrate a lot of the functionalities mentioned above into our final product. We will be using ultrasonic sensors for detecting objects, like many of the examples discussed above, but we will also integrate cameras for added obstacle detection (and to be able to recognize things like crosswalks). We also hope to keep the

cost down, to ensure affordability, without compromising our device's effectiveness. Unlike many of the solutions we found (except the SmartCane) we hope to create device that can fit onto the end of a collapsible cane, so the user can still experience the portability of a traditional cane

2.6 Evaluation of alternative solutions

Our initial idea for the overall project was to make an actual cane that would have all this features. However, we decided that it would be better to instead make an attachment that could be put into existing canes. This way we can tackle a larger crowd, reduce the cost on our end, and not change their routines too much by having a whole new cane.

Initially we were thinking of using Wi-Fi to send the feedback from the cane to the smartphone or smart-watch. However, due to the fact that at times the Wi-Fi on campus will fail and also due to having to hop in between routers, we decided that a more consistent option would be to use Bluetooth. Justin had told us that most visually impaired people (that he knows) will always carry their smartphones and smart-watches on them, so this helps to ensure that there will always be a stable connection in between their devices and the cane. Similarly, it's better for whenever the user is moving inside of a building.

While it is good that we're using the sensors to detect the incoming obstacles, it's not a good idea to have that as the only source of input. Hence, we decided to use a camera to aid in detecting the obstacles. This will help in case the sensors malfunction, and will also be able to measure the depth so it can have a more accurate measurement. Furthermore, it will detect what type of object it is so it can give a warning if it's a dangerous object like a streetlight or a car, and can also recognize sidewalks.

Our cane will be able to direct people inside of a building form point A to B. While this will be very useful, we're also implementing a "breadcrumb" system that the user will be able to use in order to leave a "trail". By doing so, the cane can retrace their path whenever they have to leave, and will be "picking up the breadcrumbs" along the way. It's an addition to the mapping inside of buildings so it's more efficient.

Finally, after speaking with Justin, we came to the conclusion that different users will prefer different types of feedback. Some users rely too much on their hearing, so audio feedback can result disturbing to them. Similarly, some other users rely more on the sensation from their cane, so them using haptic feedback could result problematic. This is why we decided to go for a robust system rather than a fixed system. With this, users can change the settings and select which type of feedback

they will receive.

3 Final Design

3.1 System description

Our overall system is comprised of a cluster of smaller but essential subsystems that work together to make our product work. Starting off, we have our iOS App. A major use of this app is that the user can select (by using swiping gestures and clicking) the building that he wants to go to from within the campus and be guided to it with the help of Google Maps (also known as our "Outdoor setting"). This app then connects to the Raspberry Pi via Bluetooth for a variety of reasons. With the app the user can select what type of feedback he wants to receive (haptic or auditory) and when to drop or pick up a breadcrumb while on the "indoor" environment. These actions are forwarded from the app to the Raspberry Pi.

For object detection, we used a combination of Ultrasonic sensors and a camera. They were both connected in the end to the Raspberry Pi and were responsible for making the feedback for the user. Depending on the setting made by the user on the app, this system would then output the corresponding feedback if an object was detected nearby. The camera would be used as an aid (or a double-check) to the sensors and would also help detect important objects to warn the user if he was approaching them (like a cross-walk, a car, or a streetlight). The sensors would send an activation signal (for the feedback) if the obstacle was detected to be close to the user as he walked.

Finally, the Raspberry Pi would use one more component for the indoor navigation: an accelerometer/gyroscope module. This would be used to track the user's position relative to the breadcrumb(s) he has placed, and would also be used to guide it back. This system, as mentioned previously, would communicate with the app by having the user select when to drop or pick up the breadcrumbs on the app, then receiving said information. Altogether, all of these components (besides the iOS app or the phone) would be powered by our rechargeable battery and would all be an attachment that could be added to any other cane. Of course, because of the situation with COVID-19 we weren't able to get a complete prototype with the systems being together, but that was the end goal of having all these systems.

3.2 Design Specifications

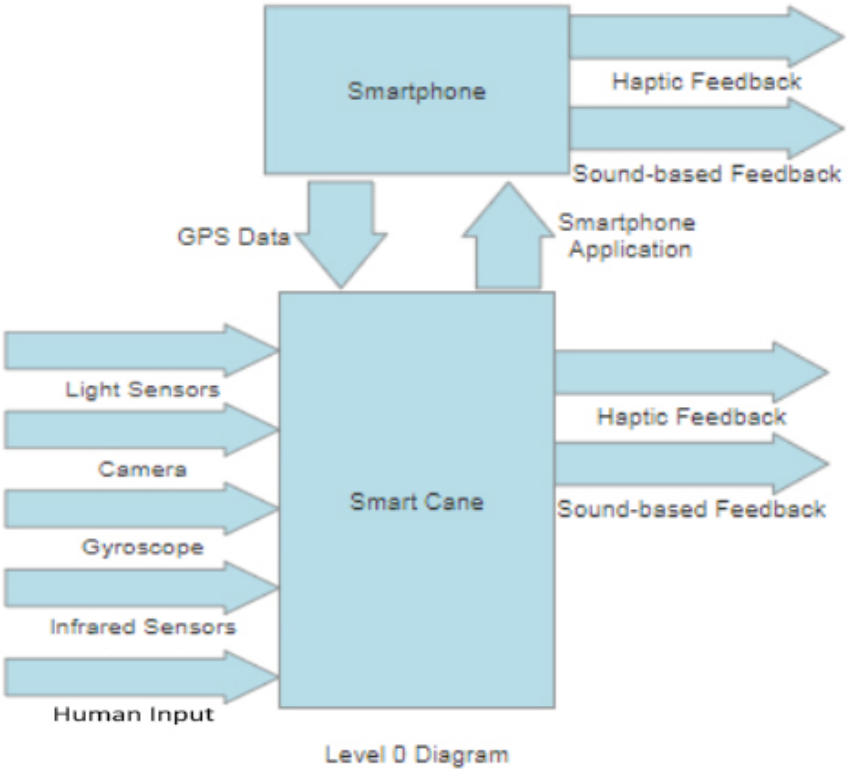


Figure 2: Level 0 Diagram

The main module for this will be the cane-mountable device itself. All the sensors will send their inputs into the cane, from which the internal computer will process the incoming data and react appropriately based a multitude of factors. Human input will also be a factor, as such features like the waypoint drop system will require input on when the waypoints should be dropped along with determining when to retread the dropped waypoints. The can will be using a combination of vibration motors and speakers to deliver the haptic and sound-based feedback respectively.

The other module is the optional smartphone connection available via Bluetooth. The smartphone is responsible for providing GPS data to the cane so that it can operate most of its features including the waypoint system. The smartphone is also where most of the user settings will be changed, depending on how the user would like to receive their feedback from the cane. Finally, the smartphone can mimic the haptic and sound feedback from the cane should the user desire so.

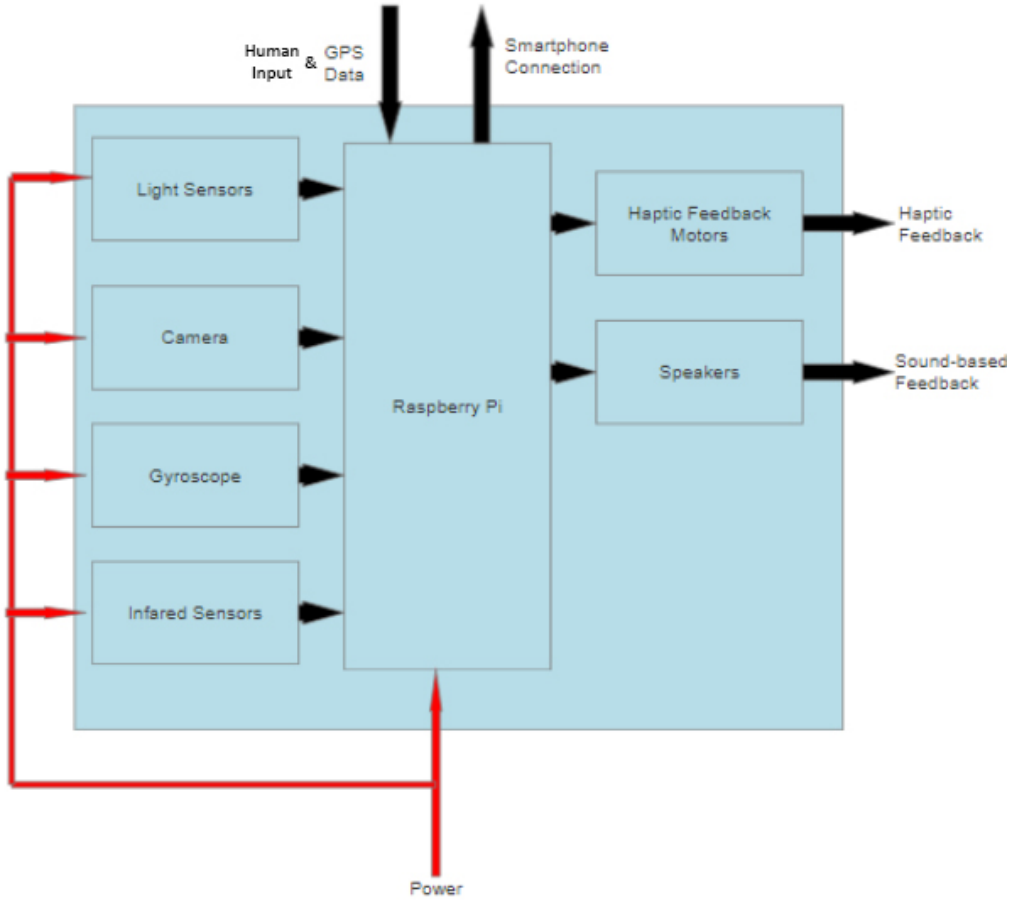


Figure 3: Level 1 Diagram

This diagram takes a closer look at the internal components of our cane module. All of the sensors and the Raspberry Pi itself will receive power from an external power source, a rechargeable battery pack. Those sensors will then feed info constantly into the Raspberry Pi. From outside the system, the Pi is receiving various forms of user input along with the GPS data from the smartphone. The Pi then processes the data, determines the appropriate action to take, then broadcasts that action to the user through both the haptic feedback motors and the speakers. The data from the smartphone connection will control how this data is processed as that is where users can change and personalize the settings of their device to best suit their daily needs.

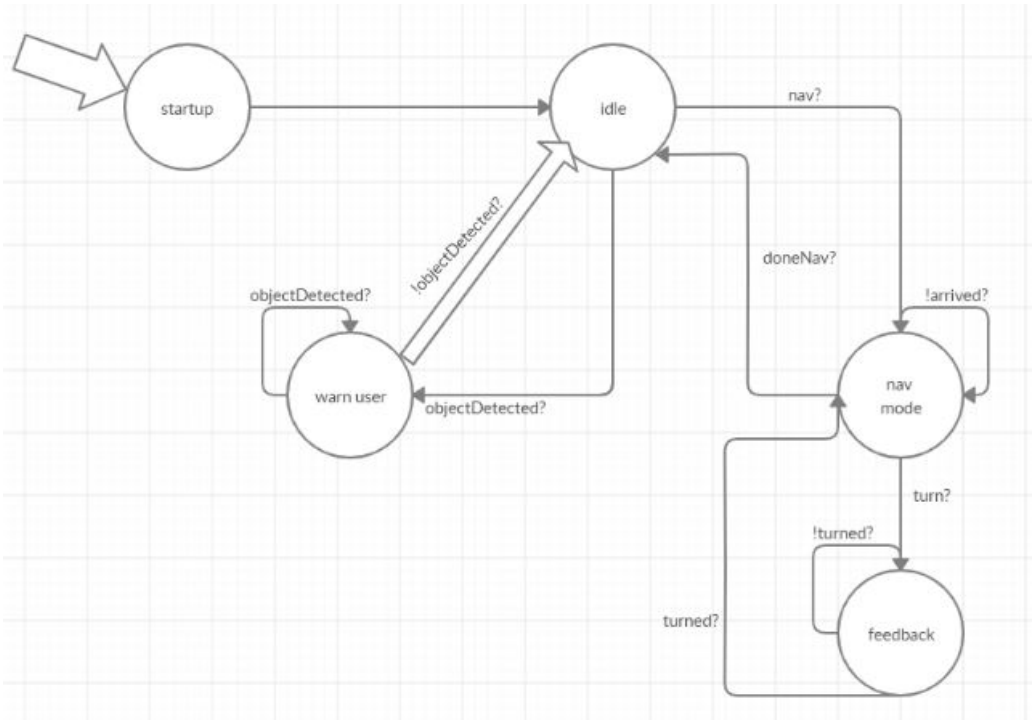


Figure 4: Finite State Diagram

This diagram describes some of the possible actions the cane will be able to take. On startup, the cane will attempt to make a Bluetooth connection with a smartphone if there is one already paired along with anything else needed to startup. A small vibration will alert the user when startup is finished and then the cane will enter the "idle" state.

In the idle state, many of the unneeded sensors like the gyroscope and GPS data will be turned off to save on power and extend battery life. The only real smart features active in this state are the ones necessary for object detection. Whenever an object is detected, the user will be warned by some form of feedback. Once the obstruction is cleared, the cane returns to the idle state.

Should the user want some sort of navigational assistance, then the cane will enter "nav mode" where the cane is able to take full advantage of its sensor suite. This mode is where some of the cane's unique features like the waypoint system are housed. The gyroscope will be responsible for determining the orientation of the user and guiding them along the path to their destination. At turns, the cane will provide feedback to indicate to the user which direction to head in. The cane will also automatically leave waypoints at certain intervals, along with allowing the user to set additional ones as needed. Once navigation is finished, the user can choose to

follow their waypoints set earlier to go back where they came from, or to discard the waypoints and head down a new path.

3.3 Approach for design validation and testing procedures

This project will bring multiple functional systems together to create our final device. Each of these systems can be tested individually to ensure they are working, and we have chosen tests and procedures that will be able to validate their functionalities. These system level tests will be conducted on: the obstacle detection, the outdoor navigation, the indoor navigation, and the emergency contact feature.

Our obstacle detection system will lie within the device housing that will be attached to the cane. In order for it to work correctly, the RaspberryPi, ultrasonic sensors, and vibration motors/speaker must all work together. This function will be tested in two part, first by placing obstacles within the radius we had specified in our requirements (10ft in front).We will select different size objects (depending on what we have in our houses) and place the sensors at half foot increments from 0.5-10 ft to ensure the detection works. For overhead, a towel can be taped above a door way to simulate an overhead obstacle. If the sensors accurately detect the obstacles in front and above the system, and consistently relay this information through the users choice of output (audio or haptic). Then we can say this part of the project is working.

For our outdoor navigation, we have to make sure that the developed application is able to communicate with the device the user will attach to their cane (specifically the vibration motors). We can test this feature by inputting multiple on-campus locations, using voice to text, and making sure the app is retrieving the appropriate navigation directions and sending them to the pi. If the instructions are being accurately sent, we can validate this function by choosing on-campus locations, then following the feedback given to the user from the device. If we are able to accurately go to our intended destination, then we can conclude that the system is working correctly.

Our indoor navigation system will rely on the pi's gyroscopic sensor to track a user's location relative to a series of user chosen anchor points set through the app, as well allowing the user to navigate back to those points through cues communicated from the vibration motors. First we will test to make sure the user is able to set one anchor point, and then able to be led back to it. Then we will mark a series of points on the floor, with tape, while we also set these anchors with the app. If we can be guided back to the markings on the floor in the correct order, then we can confirm that the system has the ability to consistently mark, and recognize, these points-as

well as accurately guide the user to each point.

The emergency contact feature involves a button on the device, and the app, to allow for a phone call to be made. To test our device's ability to contact an emergency contact, we will set a non-emergency number as the contact (using voice to speech), and activate the button on the device (which will require 2 long presses). If we can consistently use the button to make the phone call, we can be confident the feature will work in an emergency situation.

Our demonstration of our project will closely resemble the methods we used to test its system-level functionalities. To demonstrate the ability of our obstacle detection, we will set up an indoor space with cardboard box, or chair, obstacles and record a team member navigating these obstacles blind-folded. The audio feedback option will be turned on so the cues can be heard on camera. To demonstrate our outdoor navigation, a video of a team member setting a location using the app, and then navigating to the chosen location using the device (again with audio feedback on) will show the functions capability. To demonstrate our projects indoor navigation ability, a room will be set up with pre-marked x's on the floor, a team member will be filmed going to each x and setting an anchor point. Then they will be blindfolded, and will use the device to navigate back to each x until they reach their starting point. The emergency contact feature will be demonstrated using a video of an emergency contact being set, then the button on the device will be activated, and the phone will be shown making an outgoing call. We believe these demonstrations will be sufficient proof of the features that will make up our project.

4 Implementation Notes

4.1 Obstacle Detection (Ultrasonic Sensors)

We received most of our materials right when the quarantine due to COVID-19 started. Given that now we couldn't meet to work for the project, we had to split up the testing and work so we could progress on this project as much as possible. Therefore, the Ultrasonic sensors were at first tested and calibrated using an Arduino rather than a Raspberry Pi (at the moment only 1 of us had a Raspberry Pi).

```
#include "SR04.h"

//Initialize pin variables
#define TRIG1_PIN 12
#define ECHO1_PIN 11
#define ECHO2_PIN 9
#define TRIG2_PIN 10
#define v_motor_1 2
#define v_motor_2 3

//Initialize sensor objects and distance variables
SR04 sr04 = SR04(ECHO1_PIN,TRIG1_PIN);
SR04 sr04_2 = SR04(ECHO2_PIN, TRIG2_PIN);
long a,b;
```

Figure 5: Use of library, declaration of pins, and creation of sensor objects in Arduino

We used the Ultrasonic sensors HC-SR04. It so happened that there is a library on Arduino that does all the computation for finding the distance that the sensors detect. Since Arduino is very similar to C++, you can see that the first thing that happens on the code above is that we include the library for the "SR04" sensor. Then we define the pins that will be used by the sensor(s) and store those in variables that we use later on. Now, this implementation was easier to do on Arduino than on the Raspberry Pi since we can just create the SR04 objects and call some functions. For now we will demonstrate the logic behind the Arduino code, and later on this section we'll display the logic behind the Raspberry code. At the end of the previous picture, we created the objects for 2 sensors where the parameters are the "echo" and "trig" pins.

```
void setup() {
  //Declare output pins and set them as low for the vibrating motors
   pinMode(v_motor_1, OUTPUT);
   digitalWrite(v_motor_1, LOW);
   pinMode(v_motor_2, OUTPUT);
   digitalWrite(v_motor_2, LOW);
   Serial.begin(9600);
   delay(1000);
}
```

Figure 6: Setting the digital output pins for the vibrating motors

Here we set the digital output pins for the vibrating motors. The "setup()" function on Arduino runs only once before the rest of the code. At the very start we set the digital outputs to low (0) which would then be turned to high (1) if the distance detected by the sensors were less than 1 ft. Finally, in this portion we

start the "Serial" which is where you can see the feedback from the Arduino to the computer. We used this serial to see what distance it detected and see on what range it worked best.

```
void loop() {
  //Get the distance from the sensors
   a=sr04.Distance();
   b=sr04_2.Distance();
   delay(100);

   //If the distance is below a foot, go ahead an activate the motors
   if(a < 30){
    digitalWrite(v_motor_1, HIGH);
   }
   if(b < 30){
    digitalWrite(v_motor_2, HIGH);
   }

   //Bring the motors back to low.
   delay(100);
   digitalWrite(v_motor_1, LOW);
   digitalWrite(v_motor_2, LOW);
}
```

Figure 7: Infinte Loop on Arduino

This is the last portion of the Arduino code. Here we get the distance readings from the sensors and make a simple condition where the respective motor would vibrate if it's sensor detected a distance of less than 30 cm. The way that we turn on the motor is that we send a High signal to the output pin connected to the motor. We then add a delay so it can remain on for a while before turning it back off.

```
#function takes in sensor number (1 or 2) and calculates distance object is from sensor
def distance(sensNum):
    GPIO.output(led, False)
    GPIO.output(led2,False)
    #for sensor 1
    if sensNum =='1':
        GPIO.output(trig, GPIO.HIGH)
        time.sleep(.00001)
        GPIO.output(trig, GPIO.LOW)
        startTime=time.time()
        #timeout is necessary to prevent from loop stalling
        #in case the pi is busy and misses the trigger pulse coming back
        timeout=maxtime+startTime
        while GPIO.input(echo)==0 and startTime<timeout:
            startTime = time.time()
        endTime = time.time()
        timeout = endTime+maxtime
        while GPIO.input(echo)==1 and endTime<timeout:
            endTime = time.time()
    #sensor 2
    if sensNum=='2':
        GPIO.output(trig2, GPIO.HIGH)
        time.sleep(.00001)
        GPIO.output(trig2, GPIO.LOW)
        startTime=time.time()
        timeout=maxtime+startTime
        while GPIO.input(echo2)==0 and startTime<timeout:
            startTime = time.time()
        endTime = time.time()
        timeout = endTime+maxtime
        while GPIO.input(echo2)==1 and endTime<timeout:
            endTime = time.time()
    elapsedTime= endTime - startTime
    #distance returned in cm
    distance = round(elapsedTime*17150,2)
    return distance
```

Figure 8: Raspberry code for Ultrasonic Sensor

As stated previously, when moving the Ultrasonic sensor code onto the Raspberry Pi, we had to perform more calculations since there were no premade libraries there.

As portrayed in the picture above, we had to measure the time that the echo took to travel. We had to look into a forum on Adafruit's website to see how we could calculate this. Everything else besides this worked the same as with the Arduino. If the sensor detected a close distance, it would activate its respective motor.

4.2 Object Recognition

Another important aspect of our project was the ability to identify objects. We decided to utilize the easy to use Raspberry Pi to host our object recognition system, and took use of the many available camera libraries available for the Pi. We started with four different cameras, two of which were standard Pi cameras. The other two cameras were stronger and had adjustable lenses therefore offering a much higher quality image output.

When it came time to test the cameras and record their outputs, we quickly realized that one of the big cameras actually had no way to physically connect to our Pi, so immediately we were down one camera. Thankfully the remaining three cameras could connect with via ribbon cables and we were able to gather useful test images out of them as seen in the experimental results section. From now on, we will call the three cameras Lens, V1.3, and V2.1.

To keep matters simple, we initially chose the lens camera as it had both the best quality and highest FOV of the cameras. Once we confirmed the functionality of the cameras, we then had to find a library that would allow us to perform object recognition on live video. If we couldn't find one for live video, our plan was to quickly take pictures and analyze them. This would of course require a library that could analyze multiple images per second. Our initial findings pointed towards YOLO V3 being a popular and powerful object recognition library. From YOLO we had a few options. Both V3 and V2 were still kept updated due to some issues with some embedded systems not supporting V3. Among each version there was both normal and "Lite" versions of YOLO. We decided to test on the lite version as we figured there was no way normal could run in a reasonable time on hardware as weak as the Pi. Sadly the Pi proved far weaker than even we thought. V3 wasn't even able to run on the Pi, while V2 lite took on average 45 seconds to analyze a single image. There was no way this was suitable for an environment requiring multiple images analyzed per second. Back to the drawing board we went where eventually we discovered upon Tensorflow Lite and their object recognition suite. Boasting an average of 2 fps on a stock Raspberry Pi 3 (which isn't much, but is around 90 times faster than YOLO V2 lite).

https://www.overleaf.com/project/5eac3a42ef1df20001eb63c4

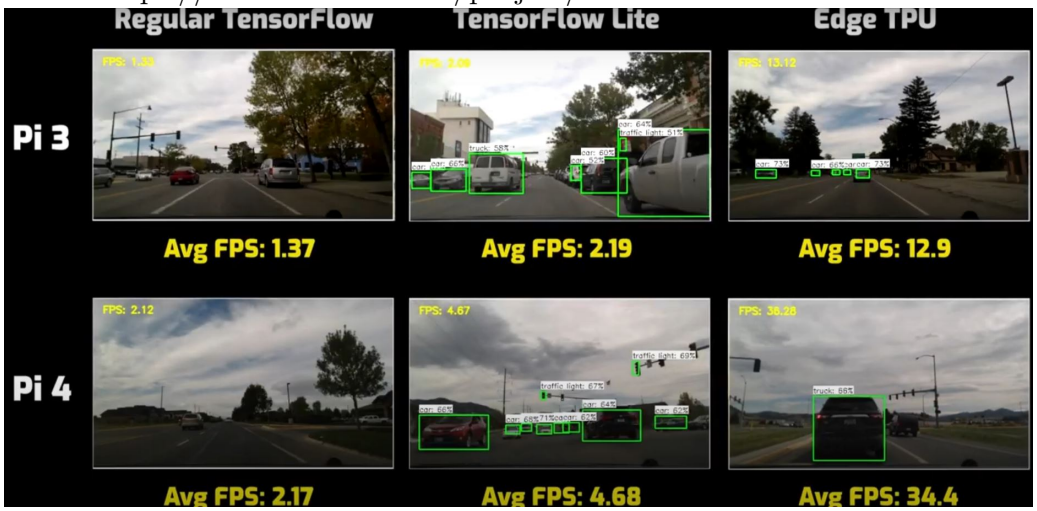


Figure 9: Comparison of TensorFlow performance on various systems. Image courtesy of Edje Electronics on YouTube

While in most cases, 2 FPS isn't suitable for real time applications, at the speeds users of this cane would be travelling at (average walking speeds, around 2-3 MPH) this provided more than enough time to recognize and analyze images. When hooking up the cameras, another tragedy occurred. For some reason, the lens camera was suddenly unable to be recognized by the Pi. This resulted in using V2.1 for the remaining testing and demonstrations as it was a standard Pi camera and the newest one we had remaining.

The model used was a specially trained model made by Google. It is a quantized COCO SSD MobileNet v1 model that identify up to 10 objects in a single frame and can recognize up to 80 objects in total. While most of these labels weren't of any use to our system (for example: apple, pear, bear, etc), there were some useful ones like car, traffic light, and truck to name a few. Despite being a special model, the TensorFlow lite website has special instructions on setting up both the original model, along with how to retrain the model without losing its efficiency.

The code itself is fairly straightforward as well. Since TensorFlow lite has its own library, the code was simply a bunch of library calls that passed the video feed into TensorFlow lite. The end result was a piece of software that displayed on the Pi screen (for demonstration purposes only, in the real prototype this would not be needed as there isn't a screen on the cane) showing both the camera feed and boxes around recognized objects. Above the objects would be the model's confidence in identifying the object. On the top left was the FPS in real time on the Pi.

https://www.overleaf.com/project/5eac3a42ef1df20001eb63c4

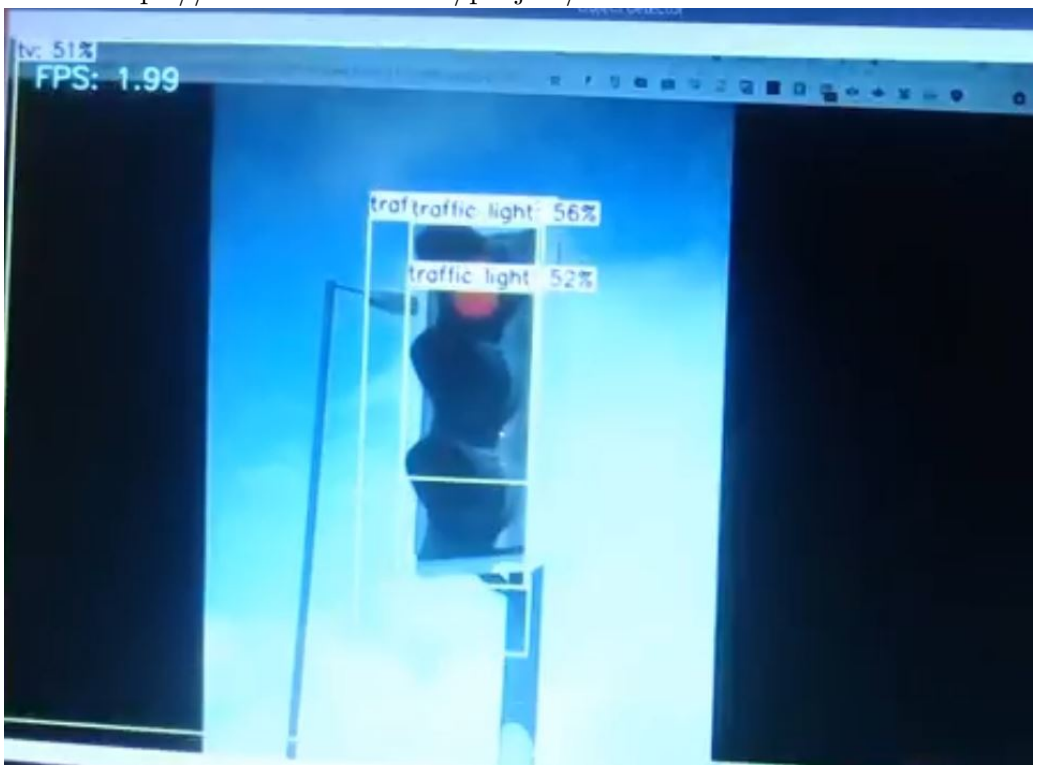


Figure 10: Comparison of TensorFlow performance on various systems. Image courtesy of Edje Electronics on YouTube

4.3 Indoor Navigation Hardware

If you have set up the RaspberryPi3, and the indoor navigation software, according to our user manual, you should be able to obtain the accelerations, and change in angle can be directly from the 'sensor' like so:

```
while (True):
    accelArr = sensor.acceleration
    gyroArr = sensor.gyro
```

Where accelArr, and gyroArr, store the changes in acceleration, and direction, respectively. The arrays store values in an (x-axis,y-axis,-z-axis) format. The loop allowed us to constantly read values from the sensor. Now, since the pi is able to receive input from the sensor, we can talk about how we used this data to determine the distance the sensor travelled, and the angle change from the initial angle the

sensor was facing. The following function was used to determine the distance, and angle change from the initial point:

```
i2c = busio.I2C(board.SCL, board.SDA)
sensor = LSM6DSOX(i2c)
def sensorDataProcessing():
    print("getting data...waiting")

    accelBeginX = sensor.acceleration[0]#initial accel in x-axis
    accelBeginY = sensor.acceleration[1]#initial accel in y-axis
    accelsX = np.array([accelBeginX])#created arrays of accel values
    accelsY = np.array([accelBeginY])
    finalAngle =0# initial orientation is set to 0
    startTime = time.time()
    try:
        #the while loop has the accel arrays constantly updating  with new values
        while True:
            accelsX= np.append(accelsX,sensor.acceleration[0])
            accelsY=np.append(accelsY,sensor.acceleration[1])
            finalAngle = finalAngle + sensor.gyro[2]
    #a keyboard interrupt was used because we didn't have a button available
    except KeyboardInterrupt:
        #once the measurement was stopped, the distance was
        # calculated using average accelerations and elapsed time
        elapsedTime = time.time()-startTime
        avgX = np.mean(accelsX)
        avgY = np.mean(accelsY)
        xDist = avgX*(elapsedTime**2)
        yDist = avgY*(elapsedTime**2)
        radius = math.sqrt(xDist**2 + yDist**2)/100
        #final angle calculated
        finalAngle=finalAngle/elapsedTime
        finalAngle=finalAngle*(180/math.pi)/1000
        #print statements sued to verify readings
        print("----------------------------------------")
        print("*** RADIUS and THETA data to be sent***")
        print("radius length indicating distances travelled: %.4f m"% (radius))
        print("theta in degrees: %.4f degrees" %(finalAngle))
        print("----------------------------------------")
```

Figure 11: Sensor Data Processor

This function isn't too complicated, to calculate the sensors distance from an initial point it constantly pushes the measured values of acceleration in the x axis and y axis into separate arrays. Once the user wants the distance back to the array, the average acceleration is calculated for both arrays. Then distance is calculated using those average, and the elapsed time. This method of distance calculation isn't precise to the millimeter, but the simplicity allows it to be calculated very quickly. To calculate the change in orientation, the sensors initial angle is set to zero, and it is updated by adding the the change in the angle to that initial value, once the data

is requested, the final angle is produced. The final angle, and distance are then sent to the software.

4.4 Indoor Navigation Software

The basic idea of the indoor navigation software is fairly straightforward. The system has to take input from the accelerometer and gyroscope about the movements of the user, and track where the user is in relation to a specific point (or many). It has to allow the user to define that point, meaning it also has to take input from the iOS app, and finally it has to relay back to the user where they are in relation to the defined point on command, meaning it has to send output back to the haptic motors. Conceptually, we can consider the system as consisting of a coordinate system. To best fit the format of the input coming from the accelerometer and gyroscope, we'll think about this system as using polar coordinates (radius, angle). A class was defined with this in mind.

```
class Indoor:
    def __init__(self):
        self.anchor = False
        self.loc = (0, 0)
        self._current_angle = 0
        self.trail = []
```

Figure 12: Class Definition

The class has four member variables. The first, 'anchor', is simply a Boolean that tracks whether an anchor has been dropped by the user - i.e. whether or not the user's location needs to be tracked. The second and third, 'loc' and 'current_angle' are similarly straightforward, just being the user's current location and viewing angle relative to the most recently placed anchor. The fourth and final member variable is 'trail', which is a list of all the anchors that have been placed. More detail about that in a moment. The class also has a number of member functions.

```
@property
def current_angle(self):
    return self._current_angle

@current_angle.setter
def current_angle(self, x):
    if int(x) < 2*np.pi:
        self._current_angle = x
    else:
        self.current_angle = x - 2*np.pi
```

Figure 13: Angle Setter

For the the sake of ease of use, the angle of the user is restricted to a range between 0 and 2 pi radians. This was accomplished through the use of a special setter function that takes advantage of some of Python 3's capabilities, allowing for those restrictions without setting the variable to be private.

```
def drop_anchor(self):
    if self.anchor:
        self.trail.append(self.loc)
    else:
        self.anchor = True
    self.loc = (0, 0)
```

Figure 14: Drop Anchor Function

This function drops an anchor point at the user's current location, and will be triggered in response to input from the iOS app. It works fairly simply, saving the user's location relative to the previous anchor point within the 'trail' variable, and then setting the current location of the user as (0,0), representing the fact that they are now directly on the new anchor point. As each anchor point is removed in order from the most recently placed to the oldest, similar to how a stack functions, we don't have to update all entries in the list every time a new anchor is placed - we can simply save each anchor's position relative to the next.

```
    def lift_anchor(self):
        if len(self.trail) > 0:
            temp = polar_to_cartesian(self.trail[-1][0], self.trail[-1][1])
            temp2 = polar_to_cartesian(self.loc[0], self.loc[1])
            self.loc = cartesian_to_polar(temp[0] + temp2[0], temp[1] + temp2[1])
            del self.trail[-1]
        else:
            self.anchor = False
```

Figure 15: Lift Anchor Function

```
def polar_to_cartesian(r, t):
    x = r * np.cos(t)
    y = r * np.sin(t)
    return x, y
```

```
def cartesian_to_polar(x, y):
    r = np.sqrt(x**2 + y**2)
    t = np.arctan2(y, x)
    return r, t
```

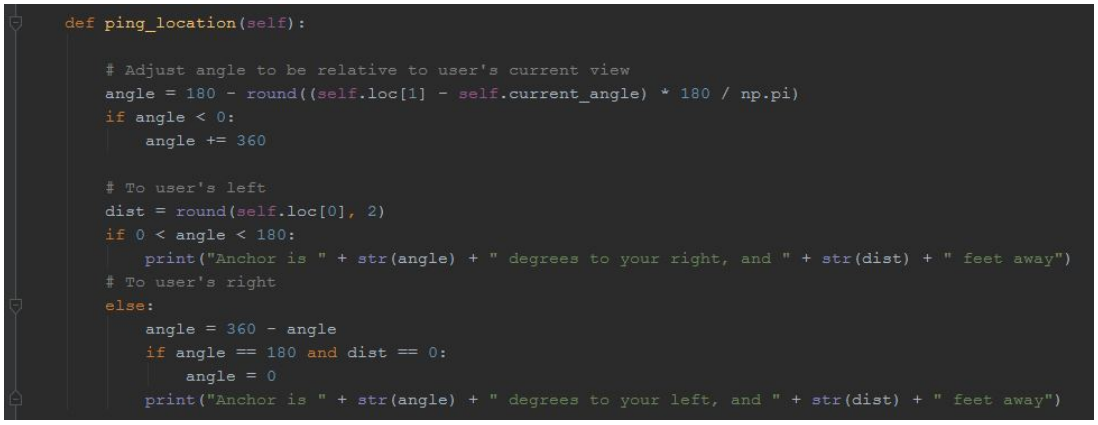
Figure 16: Helper Functions

To lift the anchor, this function is used. In order to adjust the current location of the user to be relative to the new anchor point, the coordinates are converted to Cartesian, summed, then converted back to polar format (this is done using the helper functions shown above.) If there is only one anchor down when the function is called however, this step is unnecessary and therefore won't be taken.

```
    def update_location(self, dist, angle):
        if self.anchor:
            temp = polar_to_cartesian(dist, angle)
            temp2 = polar_to_cartesian(self.loc[0], self.loc[1])
            self.loc = cartesian_to_polar(temp[0] + temp2[0], temp[1] + temp2[1])
```

Figure 17: Update Location Function

This function simply updates the user's location with new information from the accelerometer. It utilizes the same helper functions that are used in the 'lift_anchor' to do so.



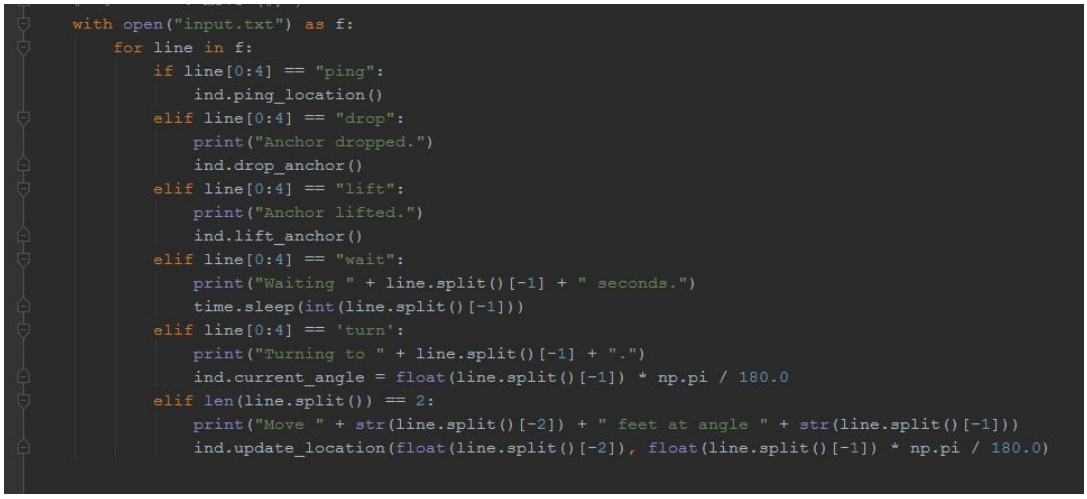
```
def ping_location(self):

    # Adjust angle to be relative to user's current view
    angle = 180 - round((self.loc[1] - self.current_angle) * 180 / np.pi)
    if angle < 0:
        angle += 360

    # To user's left
    dist = round(self.loc[0], 2)
    if 0 < angle < 180:
        print("Anchor is " + str(angle) + " degrees to your right, and " + str(dist) + " feet away")
    # To user's right
    else:
        angle = 360 - angle
        if angle == 180 and dist == 0:
            angle = 0
        print("Anchor is " + str(angle) + " degrees to your left, and " + str(dist) + " feet away")
```

Figure 18: Output Current Location Function

The purpose of this function is to provide output to the user about where they are relative to the most recently placed anchor point. In this code, the output is printed to the terminal, but in an integrated system this would be sent to the haptic sensors to produce meaningful physical results. The information is provided as an angle and a distance. The angle is first converted to be relative to the where the user is looking, then it is determined if that is to the right or left of the user, and then the output is sent.



```
with open("input.txt") as f:
    for line in f:
        if line[0:4] == "ping":
            ind.ping_location()
        elif line[0:4] == "drop":
            print("Anchor dropped.")
            ind.drop_anchor()
        elif line[0:4] == "lift":
            print("Anchor lifted.")
            ind.lift_anchor()
        elif line[0:4] == "wait":
            print("Waiting " + line.split()[-1] + " seconds.")
            time.sleep(int(line.split()[-1]))
        elif line[0:4] == 'turn':
            print("Turning to " + line.split()[-1] + ".")
            ind.current_angle = float(line.split()[-1]) * np.pi / 180.0
        elif len(line.split()) == 2:
            print("Move " + str(line.split()[-2]) + " feet at angle " + str(line.split()[-1]))
            ind.update_location(float(line.split()[-2]), float(line.split()[-1]) * np.pi / 180.0)
```

Figure 19: Input Parsing

Finally, we need a method to manage input. Input was simulated using a test file. Each new command is parsed, and the corresponding command is executed using the functions described previously.

4.5 iOS Application Software

In designing the app we first must illustrate the actual layout of the various views and how we will perform segues between each view controller. View controllers are the basic components that houses all the various widgets that you would place on a singular screen, so for every different screen that we want, we need a view controller and a corresponding custom View Controller class to be able to utilize those widgets in functions.

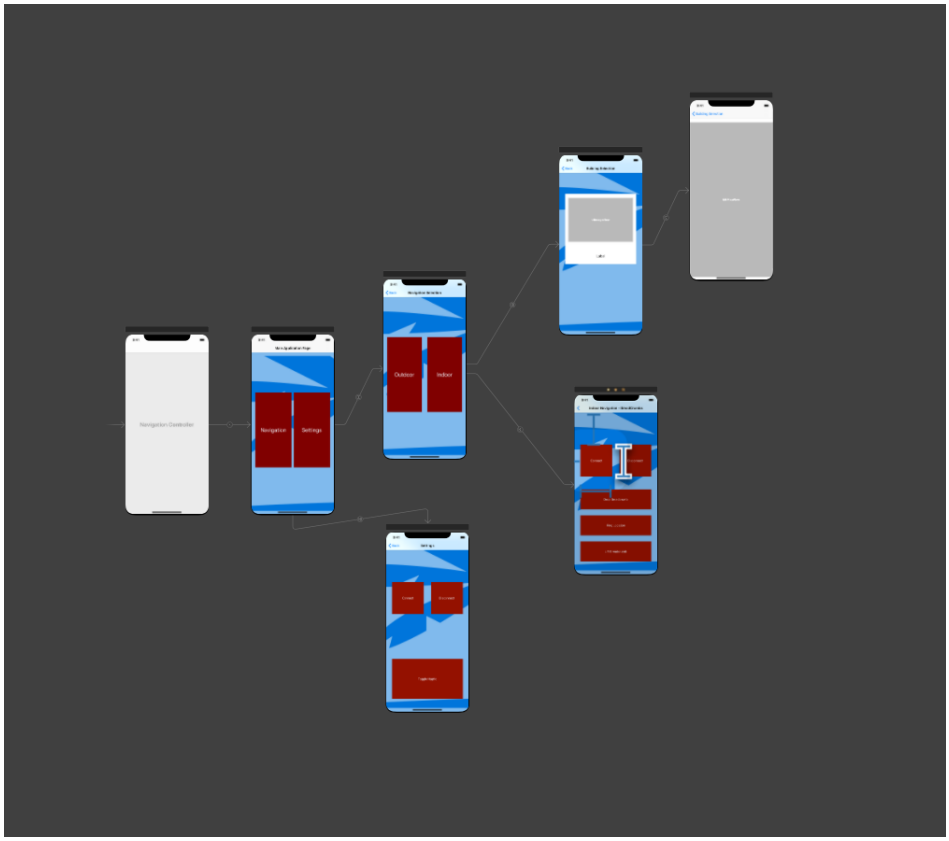
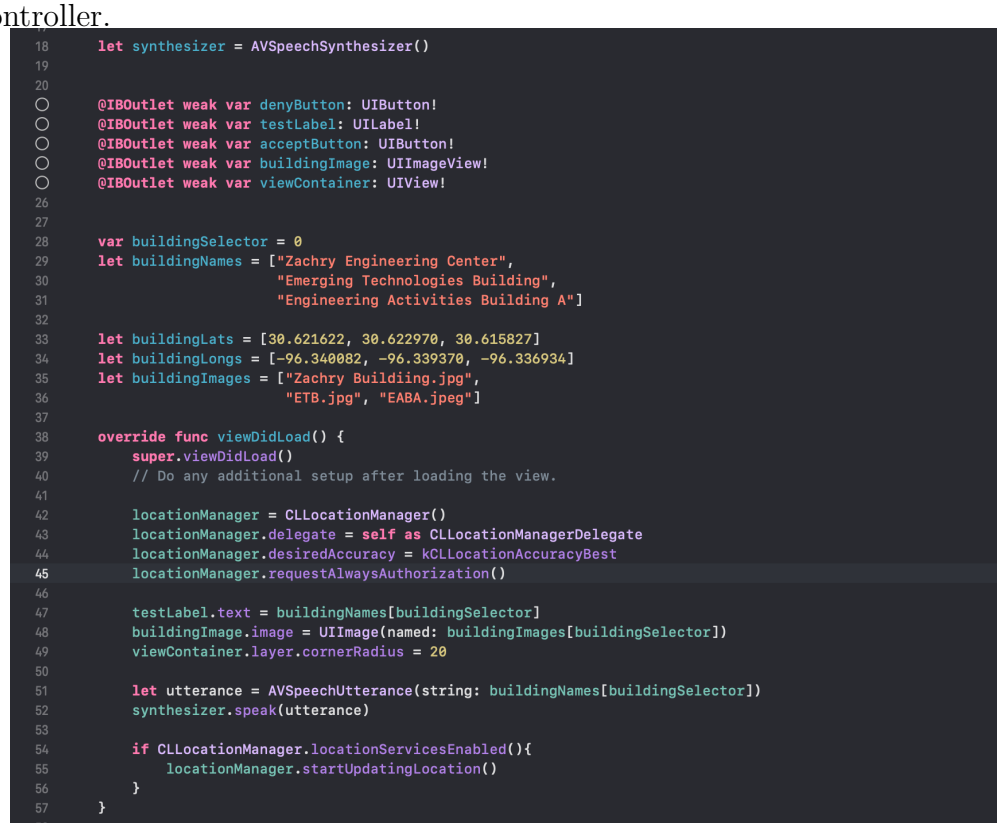


Figure 20: Storyboard

The collection of all the View Controllers is referred to as the Storyboard, as seen above. Here we can see the basic layout of our application as well as the various segues between each View Controller which is facilitated by the buttons on each

View Controller.



```
18      let synthesizer = AVSpeechSynthesizer()
19
20
21      @IBOutlet weak var denyButton: UIButton!
22      @IBOutlet weak var testLabel: UILabel!
23      @IBOutlet weak var acceptButton: UIButton!
24      @IBOutlet weak var buildingImage: UIImageView!
25      @IBOutlet weak var viewContainer: UIView!
26
27
28      var buildingSelector = 0
29      let buildingNames = ["Zachry Engineering Center",
30                           "Emerging Technologies Building",
31                           "Engineering Activities Building A"]
32
33      let buildingLats = [30.621622, 30.622970, 30.615827]
34      let buildingLongs = [-96.340082, -96.339370, -96.336934]
35      let buildingImages = ["Zachry Buildiing.jpg",
36                            "ETB.jpg", "EABA.jpeg"]
37
38      override func viewDidLoad() {
39          super.viewDidLoad()
40          // Do any additional setup after loading the view.
41
42          locationManager = CLLocationManager()
43          locationManager.delegate = self as CLLocationManagerDelegate
44          locationManager.desiredAccuracy = kCLLocationAccuracyBest
45          locationManager.requestAlwaysAuthorization()
46
47          testLabel.text = buildingNames[buildingSelector]
48          buildingImage.image = UIImage(named: buildingImages[buildingSelector])
49          viewContainer.layer.cornerRadius = 20
50
51          let utterance = AVSpeechUtterance(string: buildingNames[buildingSelector])
52          synthesizer.speak(utterance)
53
54          if CLLocationManager.locationServicesEnabled(){
55              locationManager.startUpdatingLocation()
56          }
57      }
```

Figure 21: View Controller Class

Here we can see the beginning of a particular View Controller class, this one is describing the View Controller in which we make our selection for which building we would like in our outdoor navigation. Xcode allows you to program certain features into the application without actually having to write code for it. So for the swiping mechanic, it is actually a feature of the button itself that can be changed on how it is activated which is shown below where we can see for our button that allows us to swipe through, it is activated whenever the user performs a Touch Drag Exit which translates to when the uses touches the button then drags their finger outside the buttons borders.

In the code above we can see various @IBOutlet variables which are our direct connections to the widgets on our Storyboard along with the variables for the specific buildings that we are supporting at the moment. When the View is entered through a segue, the function viewDidLoad() will always trigger, it is a sort of initialization function for the View where we can initialize things we need immediately on this

view. You can see the two main things we do here for this page is initializing the LocationManager which allows us to ping our current locations, the values for elements on our view such as our textLabel and buildingImage, and lastly our utterance which is the object that is needed to verbalize text through the Synthesizer object shown at the top of the code.

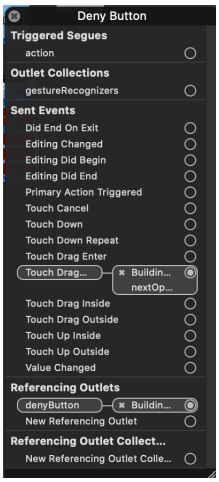


Figure 22: Button Events

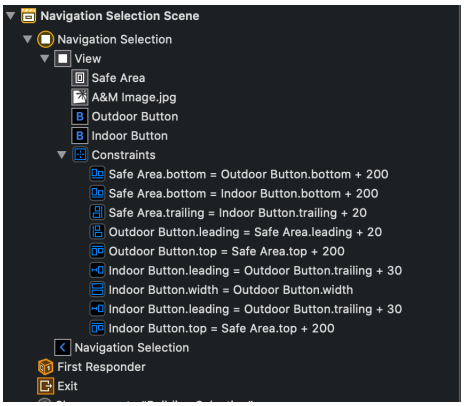


Figure 23: Widget Contraints

When laying out our widgets on our View Controllers we must design them in regards to the safe area on our screen. The way Xcode allows us to do this is through the use of constraints as seen above. These constraints tells our application the specifications of our widgets such as their size relative to the borders of the screen, distance between another widget, or even matching dimensions between multiple widgets.

Figure 24: Building Selection Functions

Above we can see some functions that are triggered when particular events occur, such as a button trigger, denoted by the @IBAction. When our acceptButton is triggered, it will call the function selectOption that will take the users selected option and attempt to open Google Maps and submit the specified parameters, being the building's location as well specifying that we would like to walk to our destination.

```
43
44
    @IBAction func dropCrumb(_ sender: Any) {
46      let utterance = AVSpeechUtterance(string: "Dropping Breadcrumb")
47      synthesizer.speak(utterance)
48
49      mqttClient.publish("rpi/gpio", withString: "drop")
50  }
51
52
    @IBAction func connectButton(_ sender: Any) {
54      let utterance = AVSpeechUtterance(string: "Connecting to Cane")
55      synthesizer.speak(utterance)
56      mqttClient.connect()                                    Result o
57  }
58
    @IBAction func disconnectButton(_ sender: Any) {
60      let utterance = AVSpeechUtterance(string: "Disconnecting from Cane")
61      synthesizer.speak(utterance)
62      mqttClient.disconnect()
63  }
64
    @IBAction func pingLocation(_ sender: Any) {
66      let utterance = AVSpeechUtterance(string: "Pinging Location")
67      synthesizer.speak(utterance)
68
69      mqttClient.publish("rpi/gpio", withString: "ping")
70  }
71
    @IBAction func liftCrumb(_ sender: Any) {
73      let utterance = AVSpeechUtterance(string: "Lifting Breadcrumb")
74      synthesizer.speak(utterance)
75
76      mqttClient.publish("rpi/gpio", withString: "lift")
77  }
78 }
79
```

Figure 25: Indoor Navigation Functions

Here we can see the code that allows us to communicate with the Raspberry Pi through the use of the Mosquitto Server hosted by it which we can interact with through the mqttClient object provided by the CocoaMQTT library. Each button on our page is tied to a specific action, the first and most important of which is the connectButton function will have our mqttClient attempt to connect to the server as well as vocalize doing so. After the connection is established, all that we need to do now is to publish a specific message to our custom Topic, in this case our topic is "rpi/gpio" but they can be named anything. Topics are just a specific mailbox that our Raspberry Pi will subscribe to and read messages only from that topic that it doesn't receive messages from another client. Once we publish our message denoted by the parameter withString, our Raspberry Pi need only to read the buffer for our topic on our specified port to decode the message and from there perform the appropriate action based on the message which can simply be determined from a series of if else statements.

5 Experimental Results

5.1 Camera Lenses

Figure 26: From left to right: Long Lens, V1.3, V2.1

These are the lenses we tested for use in our object detection system. We setup a simple testing environment to determine the quality of each lenses and the quality of the picture they can each provide. The goal was to take a picture of the back of a shampoo bottle from approximately 6-8 inches away. We would then look at how legible the small text on the bottle was to determine the quality of image provided by the camera.

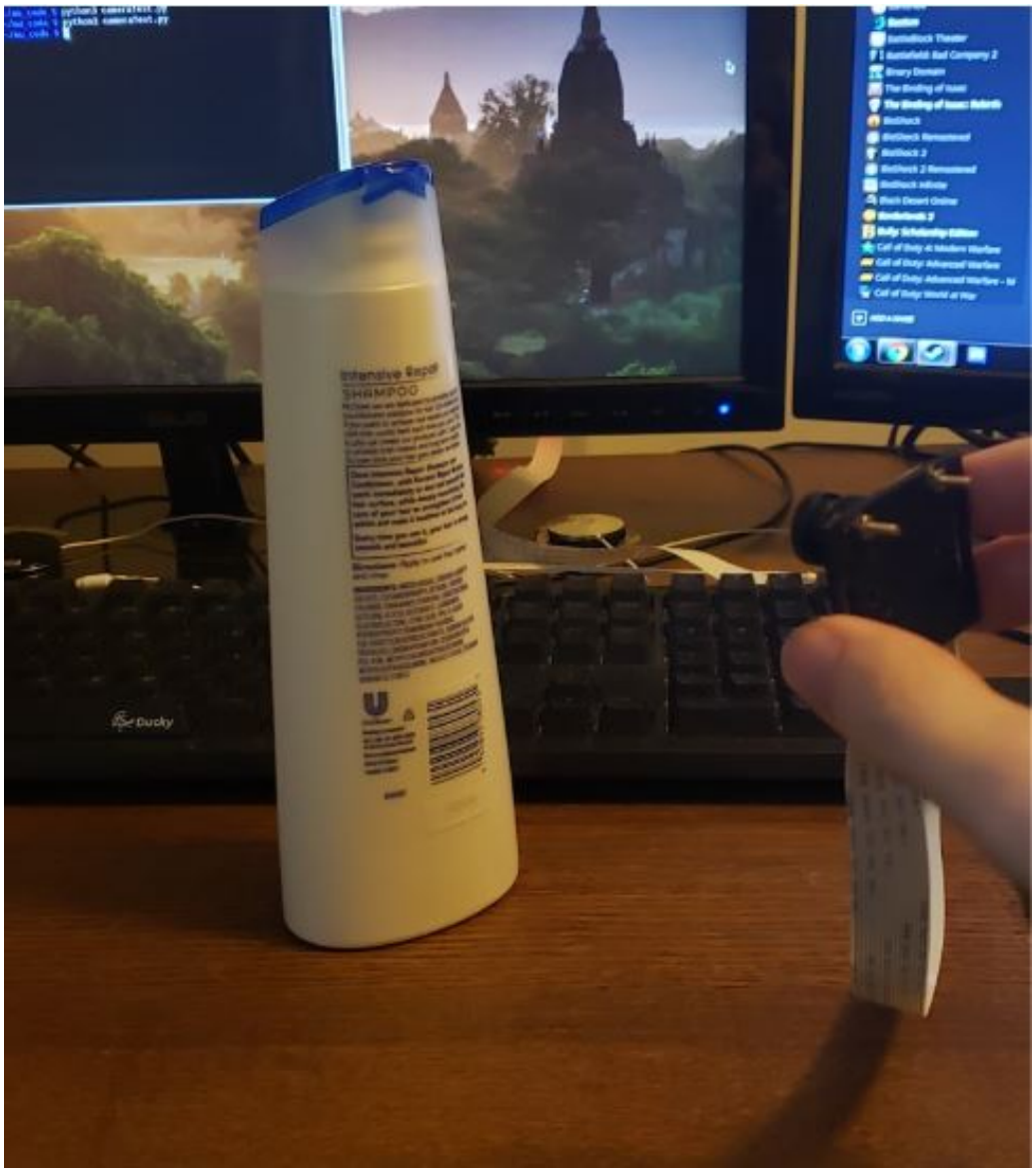


Figure 27: The testing environment

Using this setup, we took a picture with each lens and have collected them below.

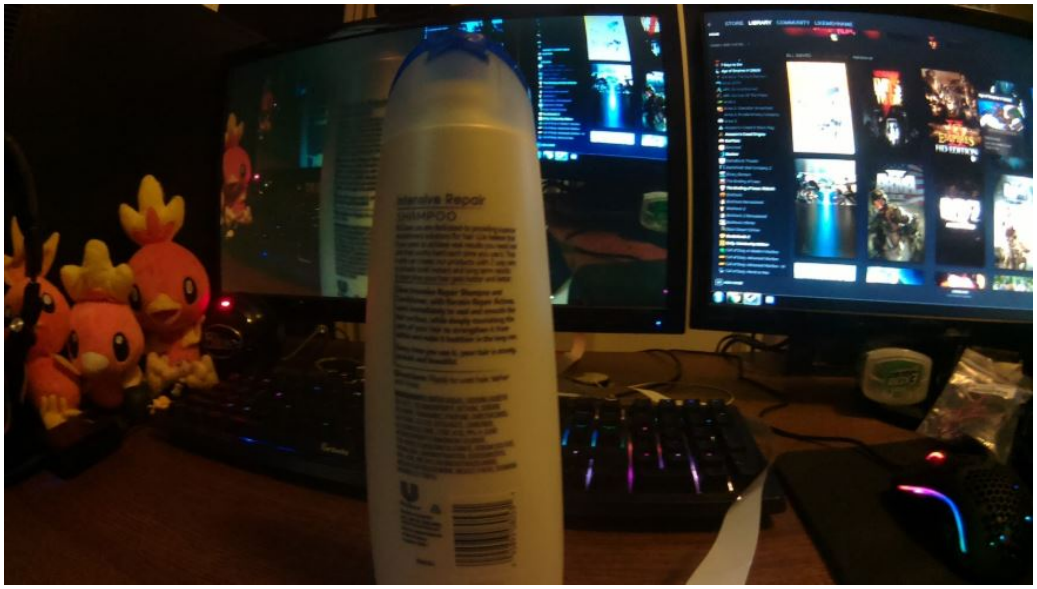


Figure 28: Picture using Long Lens

Figure 29: Picture using V1.3

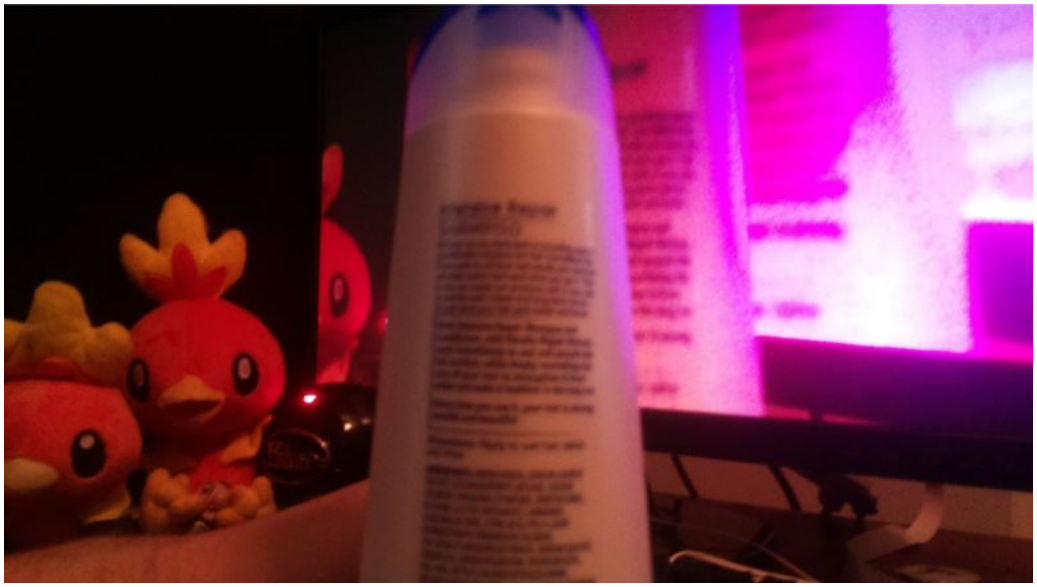


Figure 30: Picture using V2.1

As you can see, the Long Lens has not only the highest image quality, the lens has a much wider field of view compared to the other two. V1.3 and V2.1 are the same style, so they have the same quality and resolution. They both appear to struggle when focusing on items within a distance of 8 inches, but for the scope of our project they will be looking at images much further away so that is acceptable. V2.1 seems to have an even rougher time focusing on closer objects but in both images, you can see that objects further away are still clear and easy to see.

5.2 Battery

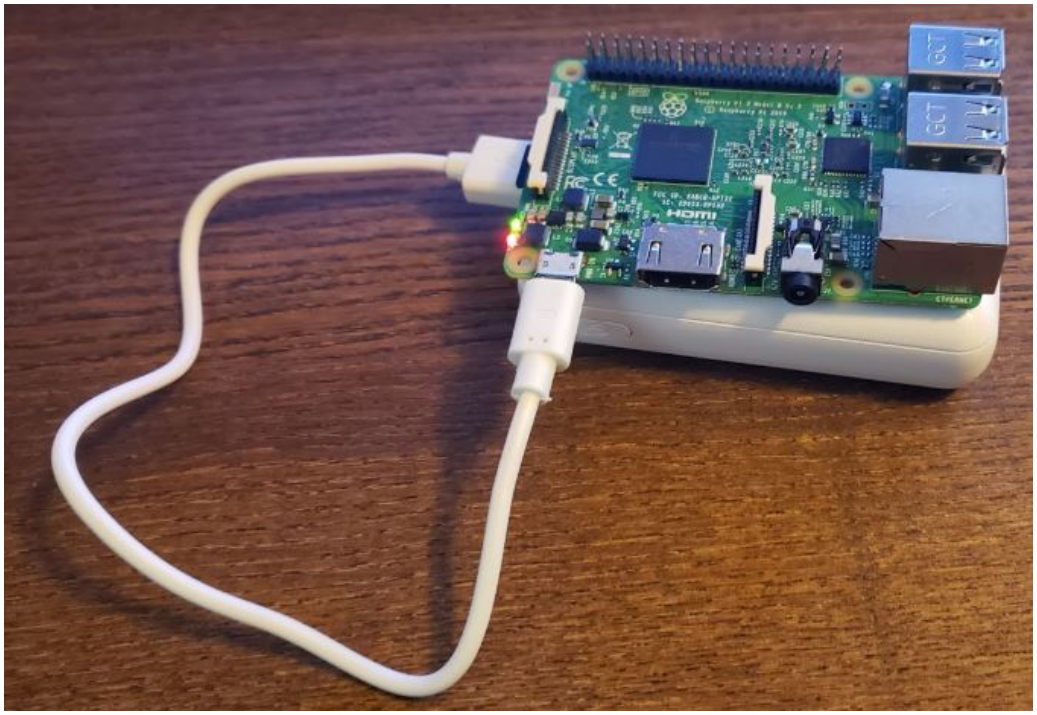


Figure 31: Battery testing environment

For testing the battery, we simply hooked up the Raspberry Pi to the battery and ran it dry on idle. We initially expected the Pi to last around 5 hours but to our surprise it ran for almost 13 hours. While this doesn't simulate the Pi operating under load, which would undoubtedly drain the battery even faster, it is easy to extrapolate an average run time of about 9 hours per charge. This is more than enough to last the whole day without needing to constantly recharge.

5.3 Ultrasonic Sensors

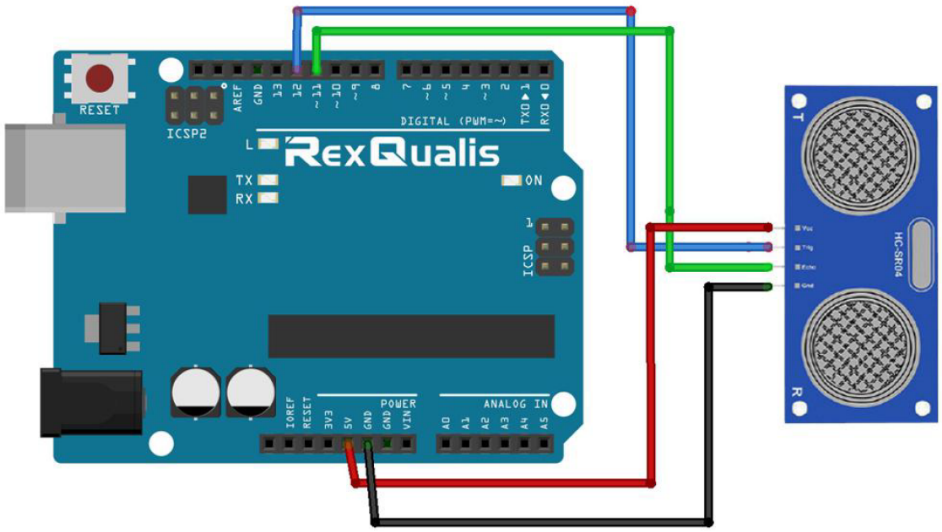


Figure 32: Schematic of the Sensor connected to the Arduino UNO for testing. [8]

In the midst of splitting the work, we ended up splitting up the components to test amongst us. Since the readings will be the same on the UNO and the Pi (and since the tester did not have a Pi), we used an Arduino UNO to test how well the sensor detected objects at certain distances. It's very important for our project to have these sensors since those are the ones that will notify the user if he is approaching and obstacle before he bumps into it. These sensors will be used heavily both in outdoor and indoor navigation.

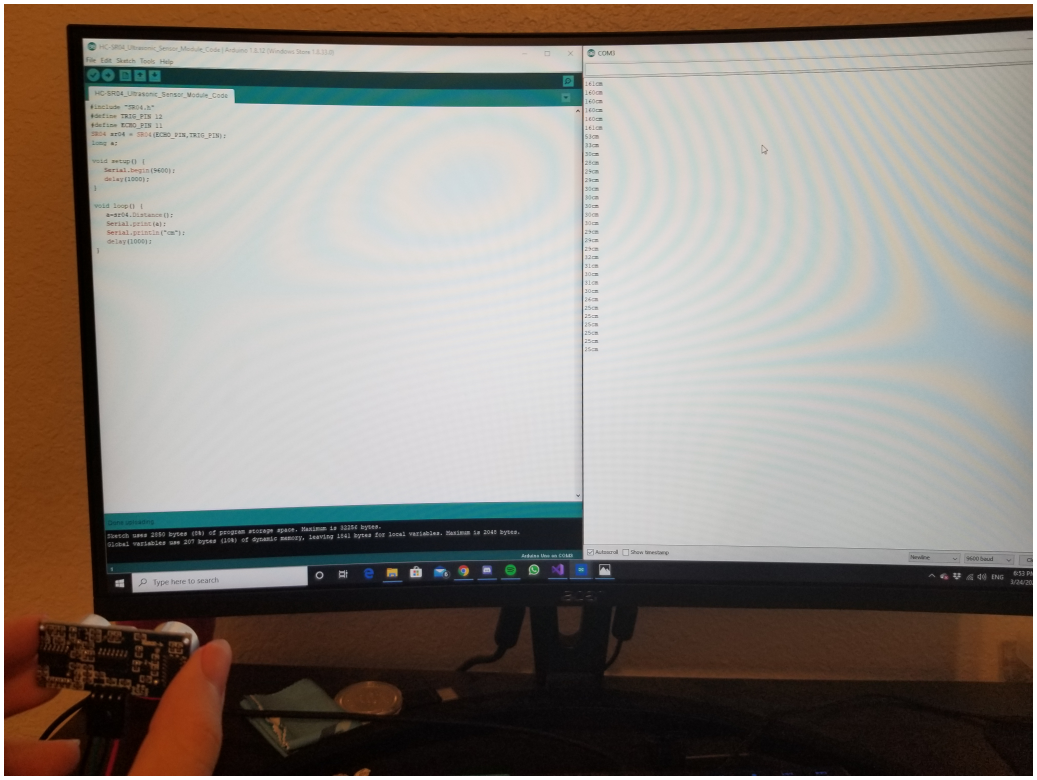


Figure 33: Holding a sensor at a distance from the monitor, where it shows the readings.

To test the sensors, we would hold it in front of an object and check if the distance was accurate enough. After many tests, we saw that the distance was near exact if it was under 5 ft. Anything above that could sometimes give a variation of up to 10%, but since it's over 5 ft it doesn't affect the scope of the project. We then did more tests in which we constantly changed the distance in between the object and the sensor to see how well the sensor gave the updated information. The sensors consistently updated the new distance without much struggle and remained pretty accurate when the distance was under 5 ft. We will continue doing more testing were now we will mimic the motion of a cane to see if it can still detect the distance accurately and update our code accordingly. Due to the COVID-19 pandemic, it has become really difficult to work on the hardware aspect of this project. Since we can't really be working on something at the same place, it slows things down. However, as we finish testing the individual components, we will start merging them together and slowly build the prototype. The thing to follow after testing these sensors will be to incorporate the vibrating motor so they vibrate when an obstacle is close.

5.4 IOS Application

We have made good progress on developing not only the User Interface for the application but the Outdoor Navigation as well.

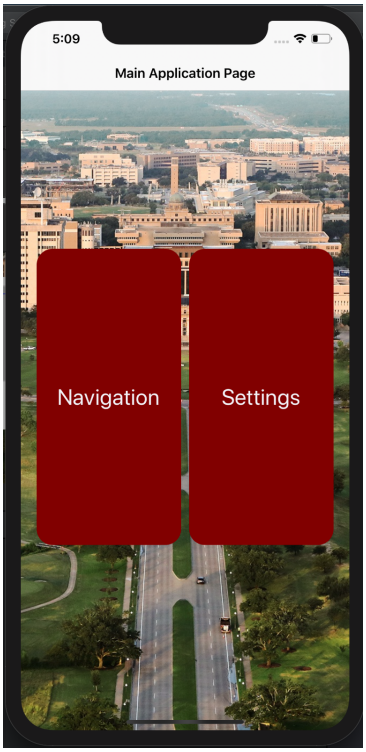


Figure 34: Main Application Page

Here we can see the Main Application page where users can either configure their settings or more commonly, navigate to a certain location. The layout of the two buttons is designed in a way that makes it easy for users with visual impairments to remember and select what options they would like. Additionally, when clicked on, the buttons will announce what has been clicked. Such as when users press the Navigation button, "Navigation", will be heard.

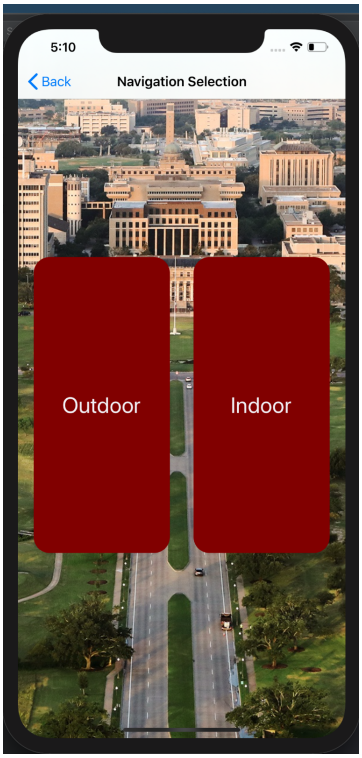


Figure 35: Navigation Type Selection Page

Here users have the choice of selecting between Outdoor and Indoor navigation. Currently our focus is on Outdoor navigation as we feel like that would be the more general use case and we are still determining the exact integrations we will be making with the Cane and the App to perform the Indoor Navigation. This page contains the same button layout structure as the Main Page for the same reasons mentioned.

Figure 36: Building Selection Page

When Outdoor navigation is selected, the user will be presented with this screen to allow them to select which building they would like to travel to. We have implemented swiping gestures to ease the process of going through the list of implemented buildings as well as selecting one. In order to scroll through the list, users only need to place their thumb on the right side of the screen and swipe left. This will trigger the hidden button on the right side of the screen which will show and vocalize the next building. Once a desired building has been reached, the user will now place their thumb on the left side of the screen and swipe right which will save and transfer the building location to the next page to generate the route to the selected building. This intuitive gesture makes it easy for users to scroll through and select the desired building, particularly those who have visual impairments.

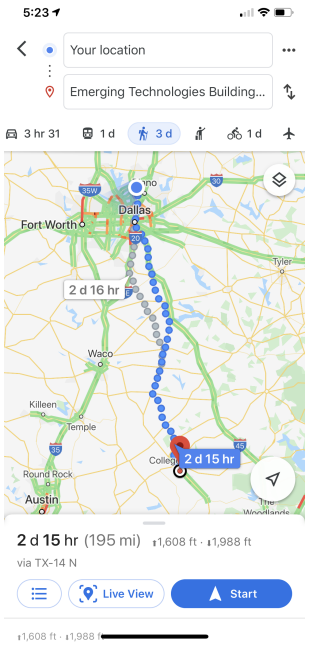


Figure 37: Route Generation Page

Once the building has been selected, the user will then be presented with the route between them and the building they wish to travel to. We have successfully made the switch to utilizing Google Maps rather than the MapKit Apple provides in Xcode. We decided to utilize Google's application because we found it to provide the user with a much cleaner implementation of Maps application rather than trying to piece together one with the tools Apple provides. So while users will need to have the application preinstalled, the simplicity and functionality that Google maps provides we feel will be benefial to the users. So after the building has been selected, the walking directions are already preselected, and all they would need to do is click start.

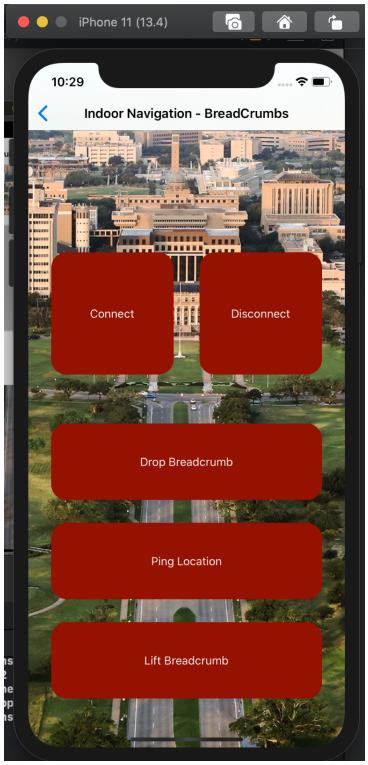


Figure 38: Indoor Navigation Page

If Indoor is selected under the navigation page, this is what the user will be presented with. This page will allow our users to connect to the Raspberry Pi on their Cane and begin sending signals to it to interact with the Indoor Navigation Software implemented in the Pi. User will first need to click the connect button to establish the one way connection through the use of a Mosquitto server that is hosted on the Pi itself. Once the connection is established, users will then be able to utilize the three functions this page provides by sending the appropriate signal to the Pi. These signals include, Drop Breadcrumb, which will drop a location marker for the user. Ping Location, which will notify the user of their current location in reference to the most recent Breadcrumb. Lift Breadcrumb, which will remove the location marker once the user has either arrived at the location or feels that the most recent marker is no longer necessary.

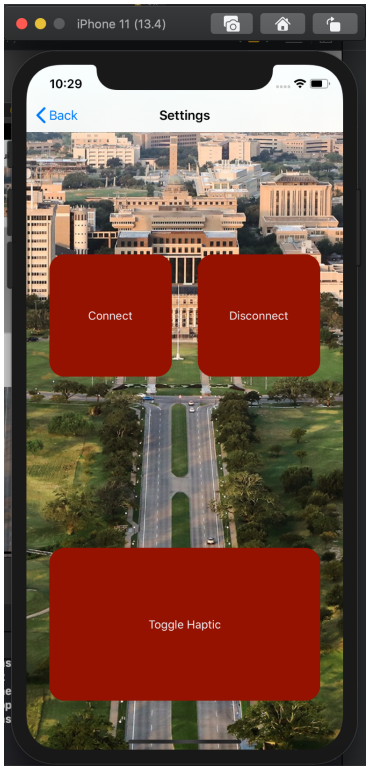


Figure 39: Settings Page

Here is the settings Screen the user will be presented with when they select the settings option in the main application page. Currently the only setting the user will be able to alter is whether they would like to havee haptic feedback come from their cane. In a similar process to the Indoor Navigation page, users first must connect to their device via the Connect button. Once a connection is established, selecting the Toggle Haptic button will either turn off or turn on the feedback system on the cane.

5.5 Indoor Navigation Hardware

To simulate user input to the pi, keystrokes were used to set an anchor. Then the sensor was moved along a ruler to keep track of distance travelled.

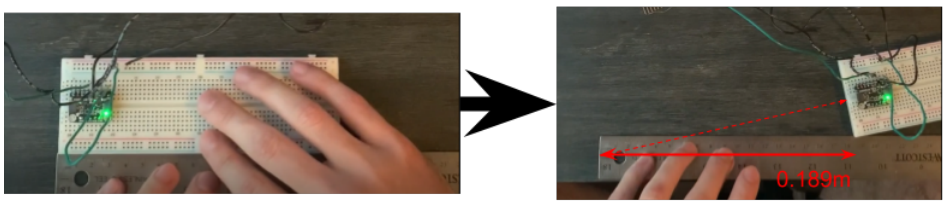


Figure 40: Test Setup

A keyboard interrupt stopped the measurements, and then output the calculated distance and angle.

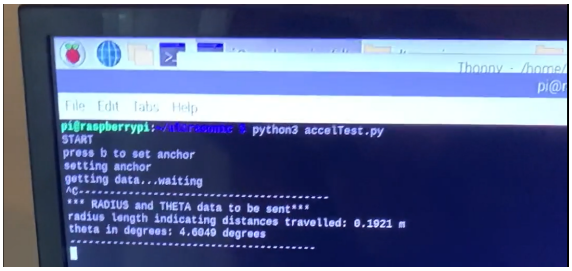


Figure 41: Calculated Distance and Angle

As shown by the example, which are screenshots of a recorded demo, the distance measurement is quite accurate. The sensor was moved 0.189m, and a distance of .1921m was calculated. However, the calculation of the angle is lacks the same preciseness. There should be ways to fine-tune the measurements of both the distance, and the angle. Namely, tinkering with the refresh rates, and the sensitivities, of the sensors. The library installed allows for this, We did change the values a little and it improved upon the values we were getting at the very beginning.

5.6 Indoor Navigation Software

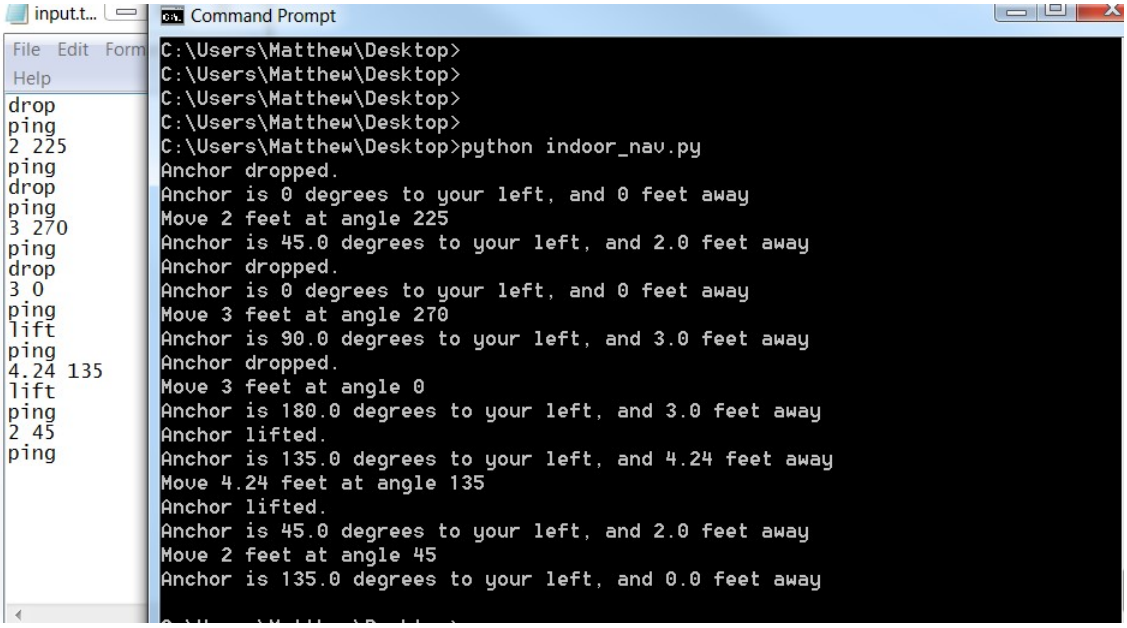


Figure 42: Indoor Navigation Output

Simulated output from the accelerometer, gyroscope, and user input via the iOS app were collated into a single simulated input file. That file was fed into the software, and the output (see example above) was checked against predicted results. This process was repeated numerous times to ensure that the system functioned as desired, which eventually it did.

6 User Manual

6.1 Raspberry pi3 setup

The Pi serves as the main controller of all the input the device receives, and the output delivered to the user. To format it, a microSD card must be reformatted using SD 5.0.1 (or newest available version), then you will want to download NOOBS (with Raspbian) onto the microSD. Insert the microSD into the pi, and hookup a keyboard, mouse, and screen. The pi should set itself up, and you are free to code. Ensure python3 is installed (it should be already) and enable SPI and I2C usig the pi config tool. Launch it with the following command:

```
sudo raspi-config
```

Then navigate the menu and enable SPI and I2C.To finish the set up, install the Raspberry Pi GPIO library using the command:

```
pip3 install RPI.GPIO
```

And run the following command to install *adafruit_blinka*:

```
pip3 install adafruit-blinka
```

This library allows you to interact withe the accelerometer and gyroscope.We used the GPIO pins for the pi,as a result, we consulted the following pinout diagram sourced from raspberrypi.org.

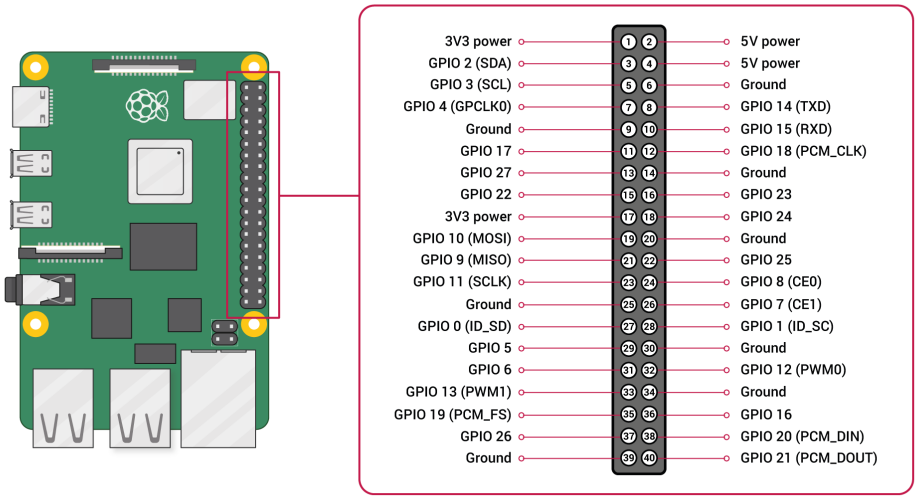


Figure 43: RaspberryPi3 GPIO pinout

6.2 Indoor Navigation Hardware

The hardware involved with tracking the users movements through an indoor area involves the pi, an accelerometer, and a gyroscopic sensor. For our purposes, we chose the Adafruit LSM6DSOX, which contains both the accelerometer, and the gyro sensors.

To connect the sensor to the pi, first attach the VIN pin from the sensor to the 5V source on the pi. Then connect ground to ground. After, connect the SCL and SDA sensor pins to separate GPIO pins. If the sensor is properly connected, a green light will turn on like in the image below

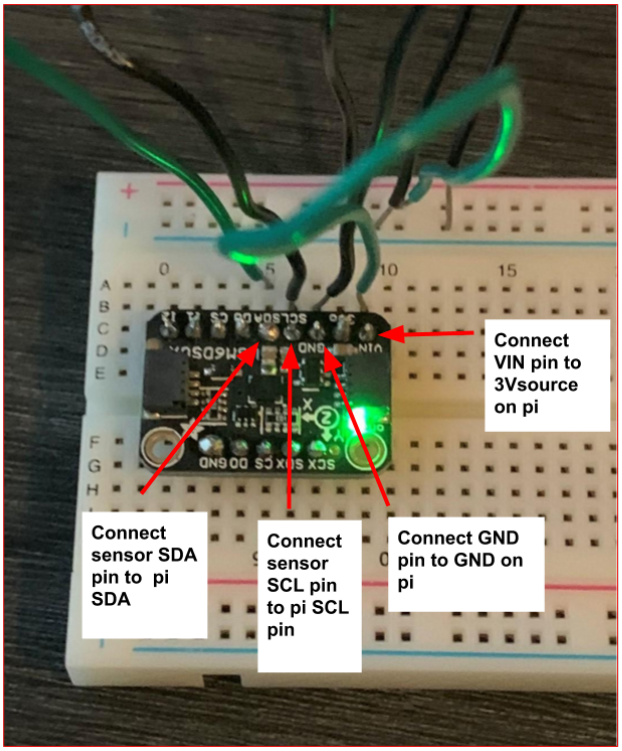


Figure 44: LSM6DSOX diagram

In order to receive input from the sensor into a python script, the Adafruit CircuitPython LSM6DS library should be installed. Since blinka is already installed, the following command can be run:

```
sudo pip3 install adafruit-circuitpython-lsm6ds
```

In order to create a functional script the following have to imported at the top of the code:

```
import time
import board
import busio
import   adafruit_lsm6ds
```

The following is added at the top of the script to create an I2c connection:

```
i2c = busio.I2C(board.SCL, board.SDA)
sensor = adafruit_lsm6ds.LSM6DSOX(i2c)
```

The script we used can be run using:

```
python3 accelTest.py
```

User inputs are asked for, in order to proceed with the measurement. To stop the measurement, the user needs to input a keyboard interrupt (CTRL+c)

6.3 Indoor Navigation Software

The software is run using the command 'indoor_nav.py'. It takes input from a text file in the same directory called 'input.txt' in the following format. Each command should be placed on its own line.

- To simulate a user turn, 'turn', followed by a space and the angle in degrees. E.g. 'turn 60' to turn 60 degrees.
- To simulate user movement, simply put the distance followed by the angle. E.g. '2 90' to move two feet at an angle of 90 degrees.
- To simulate an anchor dropping or lifting, put 'drop' or 'lift' respectively.
- To output the user's current location relative to the last anchor point placed, put 'ping'.
- To wait some period of time before executing the next command, put 'wait' followed by the time in seconds. E.g. 'wait 5' to wait 5 seconds.

7 Course debriefing

7.1 Team Management

Our team management style was very loose, but effective for us. We all came into the project with an understanding of what was required from us, and we had a conversation early on where we very openly laid out agreed on standards for how we would communicate, and how we would manage potential time conflicts. When we began a task, we clearly divided the work, but gave each person room to handle their own work without heavy interference from the rest of the group. This allowed for accountability without leading to micromanagement. If we were to begin again, one step we might want to take is to be more detailed and allow more slack in how we planned our schedule. On the whole however, our approach worked very well for us.

7.2 Safety and Ethical Concerns

There were many safety concerns we had to consider. Assuming our device went into production, and was used by its intended target audience. Our users would quite literally be putting their lives in the device's (and our) hands. If our obstacle detection failed to warn a user of a fast moving object coming towards them, they could be seriously injured. If our outdoor navigation system gave a user a wrong turn, they could end up walking into traffic, or walking into a dangerous construction zone. For our obstacle detection, We made sure to address these concerns by ensuring that our sensors were accurate within their intended range, and that the the motors gave prompt feedback if an obstacle was detected. We made sure that our outdoor navigation system retrieved accurate data from Google Maps, that no detail was lost. Something that could be changed would be using an arduino, since it would be dedicated to the code it runs (there would be no operating system interrupts). Also, an ultrasonic sensors can send varying values based on the objects distance, whereas a raspberry pi GPIO pin can only recognize high and low values (1,0).

7.3 Testing Success and Real World Applicability

The tests for our subsystems were mostly positive. The ultrasonic sensors and vibrating motors worked together and had the readings that we expected. However, the motors had very thin and loose wires and proved to be quite ineffective since the cane would be constantly bumping on the ground and moving.

Our camera system also had some drawbacks. We were unable to use "YOLO V3" as we had intended at first since our hardware was not capable of getting a fluent video with that software. Once we opted for using "TensorFlow Lite" we were able to get good identification of objects, but we had an FPS of around 2, which isn't the most optimal for quickly detecting an object and warning the user. Furthermore, since the cane would be in constant movement, it could result on inaccurate or bad readings.

The iOS App worked fully on our scope for both indoor and outdoor navigation. It communicated well with the Pi and performed its task correctly. Unfortunately, due to the "stay at home" restrictions from the pandemic, we weren't able to do a full-extent testing on the outdoor navigation to see how precise it was. However, before the pandemic we had tested how well Google Maps would work for our project and found that it worked very well. When it comes to our breadcrumb system; we were able to get a working system but again, because the systems were spread out (not all in the same physical place), we were not able to perform the entirety of tests that we had planned for calibration.

8 Budgets

8.1 Purchased Items

Item	Quantity	Estimated Cost
Raspberry Pi 3 B	1	$45
Ultrasonic Sensors	3	$6
Gyroscope	1	$15
Camera	2	$60
Vibration Motor	3	$5
Speaker	1	$5
Battery	1	$39
White Cane	1	$15
GPS Module	1	$37
Total	**19**	**$227**

Figure 45: Itemized Budget

8.2 Costs Summary

Starting from the top of the itemized budget, we have the Raspberry Pi 3. While there are other cheaper options for our microprocessor, most did not satisfy our needs for this project. What is unique about the RPI 3 is that it contains 2 camera sensor ports which in our implementation of the object detection feature is very necessary. It also contained much more processing power and ram compared to its arduino counter part, so we felt that with the multiple scripts and server that we would need running that it made more sense to utilize this device. Further testing could have been done to see if the Arduino could have handled all of our needs but due to time and budget constraints we choose to go with what we knew would satisfy our requirements.

The Ultrasonic sensors were another vital part of the system with them being directly responsible in allowing our object detecting system to work. These sensors were readily available, fairly cheap, and reliable so we felt that these would be a good fit in our design.

We initially purchased the Gyroscope module for several reasons. First and for most we did not want to hinder the development of the indoor navigation system by waiting to see if we could utilize the sensors available on the iOS device. Even if we could utilize the smartphones sensors, we did not want to isolate this product to exclusively those who own an iOS device as well as not make the Cane itself 100%

dependent on the phone.

The Cameras were another vital component of this project. They were utilized in the object detection system where they would be place at a fixed distance from one another, then utilizing a premade Computer Vision algorithm, we were able to see what objects were in front of the user and warn them accordingly. These were the cheapest camera's that were supported with the Raspberry Pi and it proved to work just fine, however performance of the image mapping could have been improved with the higher megapixel model but the practical improvements from such an upgrade aren't known to be that substantial.

Vibration motors allowed us to warn the user of incoming objects, where the motors themselves would be placed on the grip of the handle. With these being very cheap and reliable, we had no question in their utilization.

While initially we thought the speaker would be necessary, we focused our attention on the haptic feedback system rather than an auditory feedback system as we felt like it would be less confusing for the general user. We still purchased it due to its low price point and potential use case and could still be explored for implementation however the current prototype does not require it.

In order to power the entire system, a rechargeable battery is necessary to keep the Pi operational. We also wouldn't want users to have to recharge the battery several times a day as it would be very disruptive to their lifestyle, therefore we opted for a slightly bigger battery with a 10,000 maH capacity.

We included the cost of White Cane but we found a spare one we could use for the development of this project however in our potential manufacturing process, we would have an option to include the cane as well as not include it as a majority of our users probably already own one but still give the option to purchase it just in case.

Similar to the gyroscope, we did not want to hinder development of the indoor navigation system by relying on the iOS device's sensors. However, after development of our prototype, it seems that we are able to use the device's sensors but for similar reasons as the gyroscope, we did not want the cane to be dependent on the device to function correctly.

8.3 Manufacturing Costs

As mentioned in the costs summary, some of the parts could be eliminated for mass production but if the goal is to create a 100% independent device then our part requirements would be: Raspberry Pi 3, Ultrasonic Sensors, Gyroscope, Cameras, Vibration Motors, Battery, and GPS module. Which will result in a manufacturing

cost of $207 dollars including the sales tax we paid for these individual components. If we were purchasing these wholesale, our cost would be approximately $190 dollars with an optional purchase of a $15 dollar White Cane. Making our solution much cheaper than the $500 dollar equivalents currently on the market.

9 Future Plan

Due to the unprecedented pandemic hindering our ability to work together in person and come up with a physical final product, there were many features we unfortunately could not add or implement in time. The first being actual audio feedback from the speakers. While we can easily output audio on any of our devices, we could not get the speaker properly hooked up to the Pi as it required a special connection that we could not get shipped in. Another feature we planned on was a button that when pressed, would contact emergency services(or another number designated by the user). This was to be used in cases of emergencies.

Additionally we would get better hardware that could be used for the machine learning software "TensorFlow Lite" on the camera. With the current Raspberry Pi 3, we were only limited to a max FPS of around 2. By adding something like the Intel Neural Compute Stick 2, we could easily go from 2 FPS on the Pi 3 to around 15 FPS. Upgrading to the new Pi 4 along with the INCS2 would net around 30 FPS, well suitable for real time applications. We would also get better vibrating motors with sturdier wires since the ones we have are really thin and fragile, so they can easily fall out of their ports.

Finally, by being able to meet up and work on the project together we would have the time and ability of getting our subsystems together and further calibrate our prototype.

10 References

References

[1] SmartCane, "What is SmartCane", http://smartcane.saksham.org/overview/, (2016)

[2] UltraCane, "About the UltraCane," http://www.ultracane.com/about_the_ultracane, (19 July 2014)

[3] H. R. Shaj, D.B. Uchil, S. S. Rane, P. Shete. "Smart Stick for Blind Using Arduino, Ultrasonic Sensor and Android". Department of Computer Engineering K.J. Somaiya College of Engineering, Mumbai India. International Journal of Engineering Science and Computing, April 2017.

[4] R. K. Megalingam, A. Nambissan, A. Thambi, A. Gopinath, and M. Nandakumar, Sound and touch based smart cane: Better walking experience for visually challenged, 2015.

[5] V. Kulyukin, C. Gharpure, J Nicholson, and S. Pavithran. Rfid in robot assisted indoor navigation for the visually impaired. 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2004.

[6] M. Aggravi, D. Prattichizzo, G. Salvietti, "Haptic Assistive Bracelets for Blind Skier Guidance" , AH '16: Proceedings of the 7th Augmented Human International Conference 2016.

[7] Apple Developer Documentation, developer.apple.com/documentation/.

[8] "REXQualis - North America." Rexqualis Industries,Ingenious & fun DIY electronics and kits. Accessed March 27, 2020. http://www.rexqualis.com/download/.

[9] "GPIO" https://www.raspberrypi.org/documentation/usage/gpio/