Crosswalk Guardian

Authors: Colin Hackwelder, Zachary Zhao

Department of Electrical and Computer Engineering,
Carnegie Mellon University

***Abstract*—Many pedestrian walk signs nowadays have started to implement accessibility features, such as text-to-speech feedback or rapid ticking sounds, to help visually impaired know when it is safe to cross. However, there is currently no technology to guide blind pedestrians towards these blind-friendly crosswalks on their route to their destination. Our system hopes to simultaneously act as a route navigation system that directs users to their desired destination, while also helping them avoid routes or paths with blind-unfriendly crosswalks. In the worst case, if an unfriendly crosswalk is unavoidable, or purposefully chosen by the user, then we expect our product to warn them before they attempt to cross.**

***Index Terms*— Embedded Systems, Geocoding, GPS, HERE API, Raspberry Pi, Route Planning, Wearable Device**

I. Introduction

While many modern pedestrian walk signals have started to include accessibility features to assist visually impaired people, they are not widespread enough where we can assume they exist on any crosswalk, particularly those on quiet/less busy streets, or in rural areas. The sidewalks that do provide these features present a much safer environment for blind individuals, reducing their dependence on inconsistent factors, such as sounds of cars, or footsteps of other people around them, to know when it is safe to cross. Guiding visually impaired people towards these blind-friendly crosswalks can greatly mitigate the dangers of crosswalks for blind people, and help them safely navigate their way to their desired destination.

This thought process has led us to our idea of the Crosswalk Guardian, which is a product that we hope will simultaneously help blind users as a navigation tool (similar to Google Maps), as well as provide an added functionality of helping them avoid blind-unfriendly crosswalks, giving users a safe and complete user experience, tailored toward their needs. In the worst case, if blind-unfriendly crosswalks are unavoidable, or if the user deviates from a prescribed route onto an blind-unfriendly crosswalk, then we expect the Crosswalk Guardian to warn the user that the crosswalk they are about to cross lacks the appropriate accessibility features.

To be explicit, a blind-friendly crosswalk is simply a crosswalk that possesses some form of auditory feedback when the walk sign is turned on, allowing visually impaired people to know that it is safe to cross. As mentioned earlier, some forms of this include text-to-speech feedback, rapid ticking sounds, or periodic beeping sounds. Then, a blind-unfriendly crosswalk is just a crosswalk with the absence of any auditory signal. In other words, if external signals were not present, then a visually impaired person would not know if a blind-unfriendly crosswalk was turned on and off at a given moment.

While there have been studies done on route planning and navigation for blind individuals, focused on the optimization of route distance, and avoidance of obstacles (ie. road construction, natural disasters, etc.), none so far have experimented with the avoidance of blind-unfriendly crosswalks as a route optimization heuristic. Our project aims to implement this heuristic, with an overall goal of reducing the danger of land transport for visually impaired people.

II. Use-Case Requirements

From the qualitative description of our system, we proceed by introducing the use case requirements that will guide our design process and help us create a reliable system.

A. Periodic Updates

We want users to be frequently updated on the remaining distance of the step of the current route is (ie. how much distance until the next turn, crosswalk, etc.). Therefore, we expect our system to update the user every 10 seconds on the distance remaining on the current step. If the user is within 20m of a turn or crosswalk, then our system will update the user every 5 seconds instead, so that the user does not miss the turn or crosswalk. Qualitatively, our updates should be concise and clear. For example, a piece of concise feedback would be “Walk straight for 100m before turning left on Forbes Ave.”. This clearly tells the user the remaining distance on the current step, and alerts them where they will be turning next.

B. Location Accuracy

In order to reliably route the user to their desired location, our system must accurately detect their location. Specifically, we require that our system’s outputted location must be within 1m of the actual location of the user. This ensures that our feedback and directions given are accurate given their current location.

C. Long Battery Life

The system should have a battery life of 16 hours, so that it is able to sustain usage throughout the whole day without recharging.

D. Lightweight

Since the user will be carrying/wearing this device, we do not want it to be a burden for them by being too heavy. Therefore, our system should be less than 1kg in weight, so that it does not fatigue the user during usage.

E. Latency

The system should not take too long to provide feedback after it has detected the coordinates of the user. If it takes a significant amount of time, the user may have already moved a considerable amount of distance away from the coordinates detected, making the system’s feedback potentially inaccurate. Therefore we require that our device respond within 1 second after the coordinates of the user has been detected.

F. Reliability

Our device should give a valid route from the user's current location to their desired destination 100% of the time. Further, when the system detects that the user is attempting to cross a blind-unfriendly crosswalk, it should alert the user that they may be attempting to cross a blind-unfriendly crosswalk 100% of the time.

G. Unfriendly Crosswalk Avoidance

The path from the user to the destination should always contain the least amount of blind-unfriendly crosswalks, preferably zero if such a path exists.

H. Rerouting

If the user becomes lost and deviates from the route we had proposed for them, then our system should recognize this and reroute them within 30 seconds since they started to deviate.

III. Architecture and/or Principle of Operation

Our system can be divided into four subsystems:

1. Location and orientation detection
2. A backend server performing computations regarding route planning and navigation
3. A frontend interfacing with all the subsystems above, as well as providing auditory feedback to the user
4. A power supply connected to the Raspberry Pi, which will power the rest of the components via the Raspberry Pi power pins.

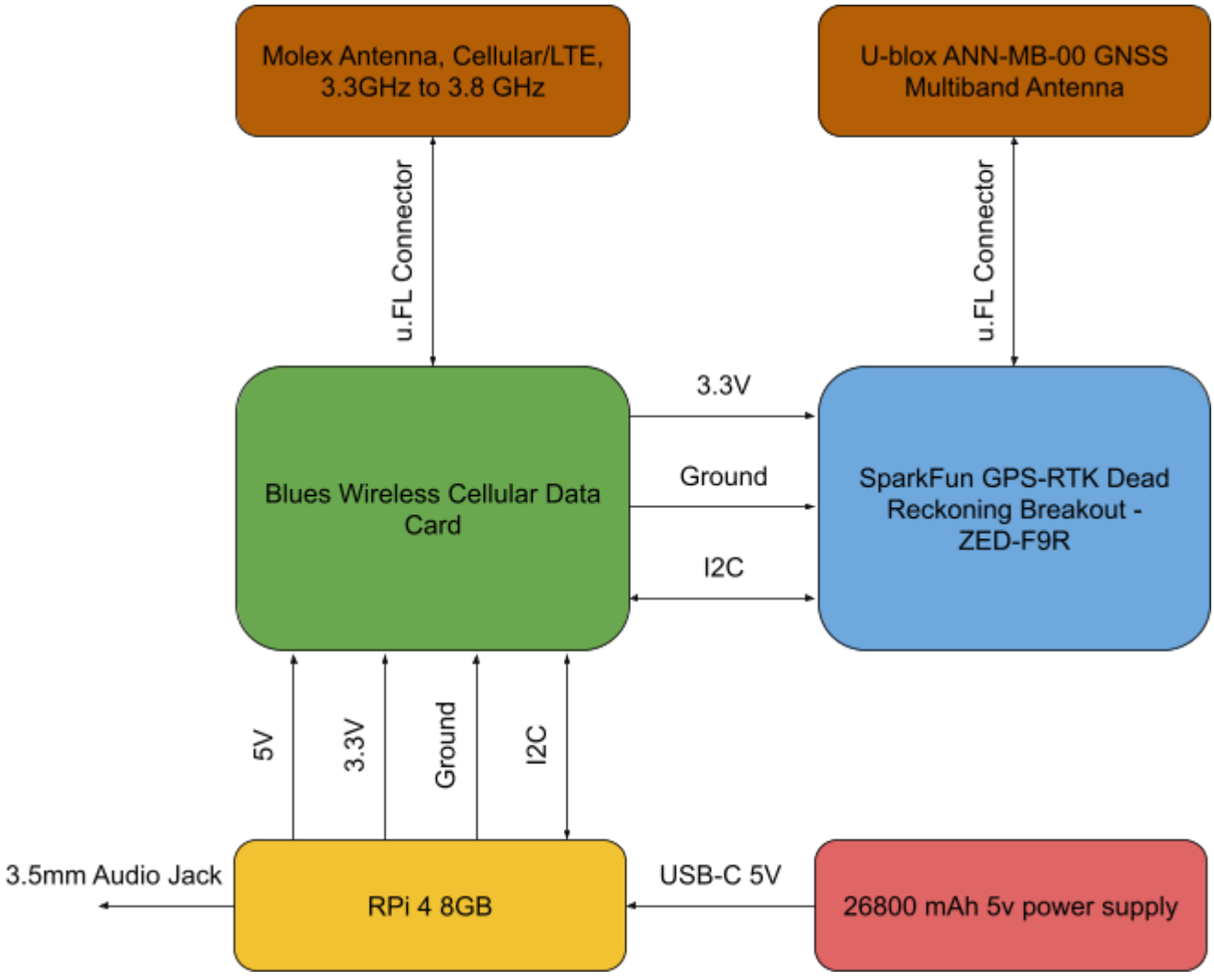

Fig. 1. The hardware system block diagram

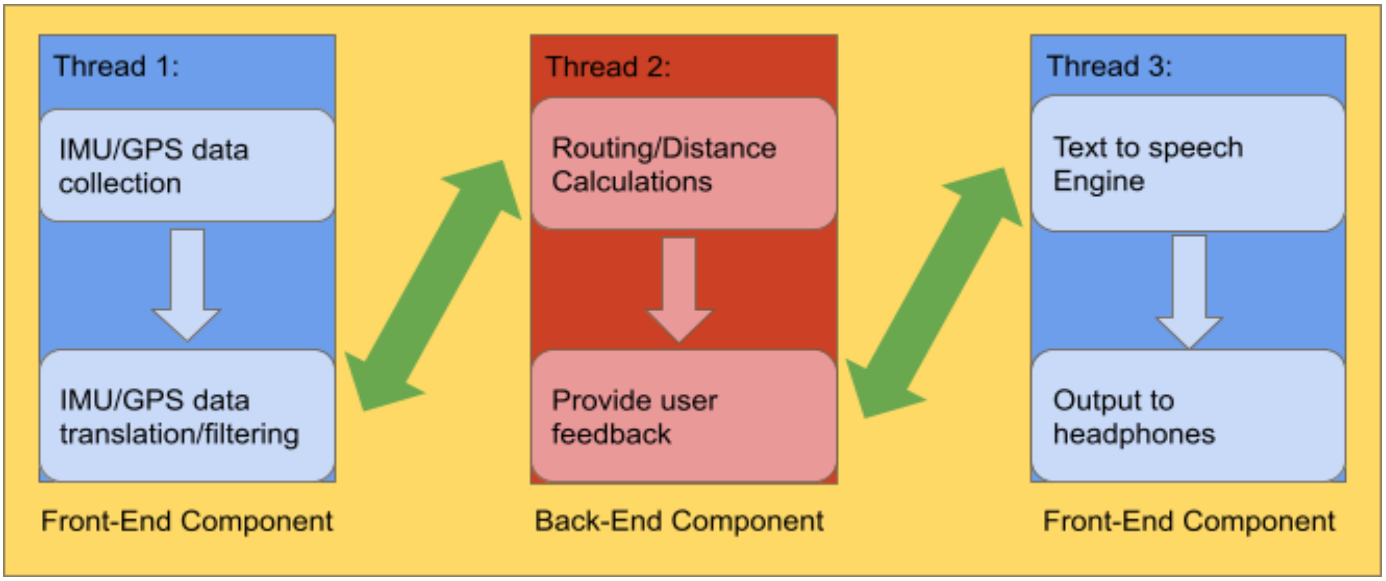

Fig. 2. The software system block diagram running on the Raspberry Pi 4

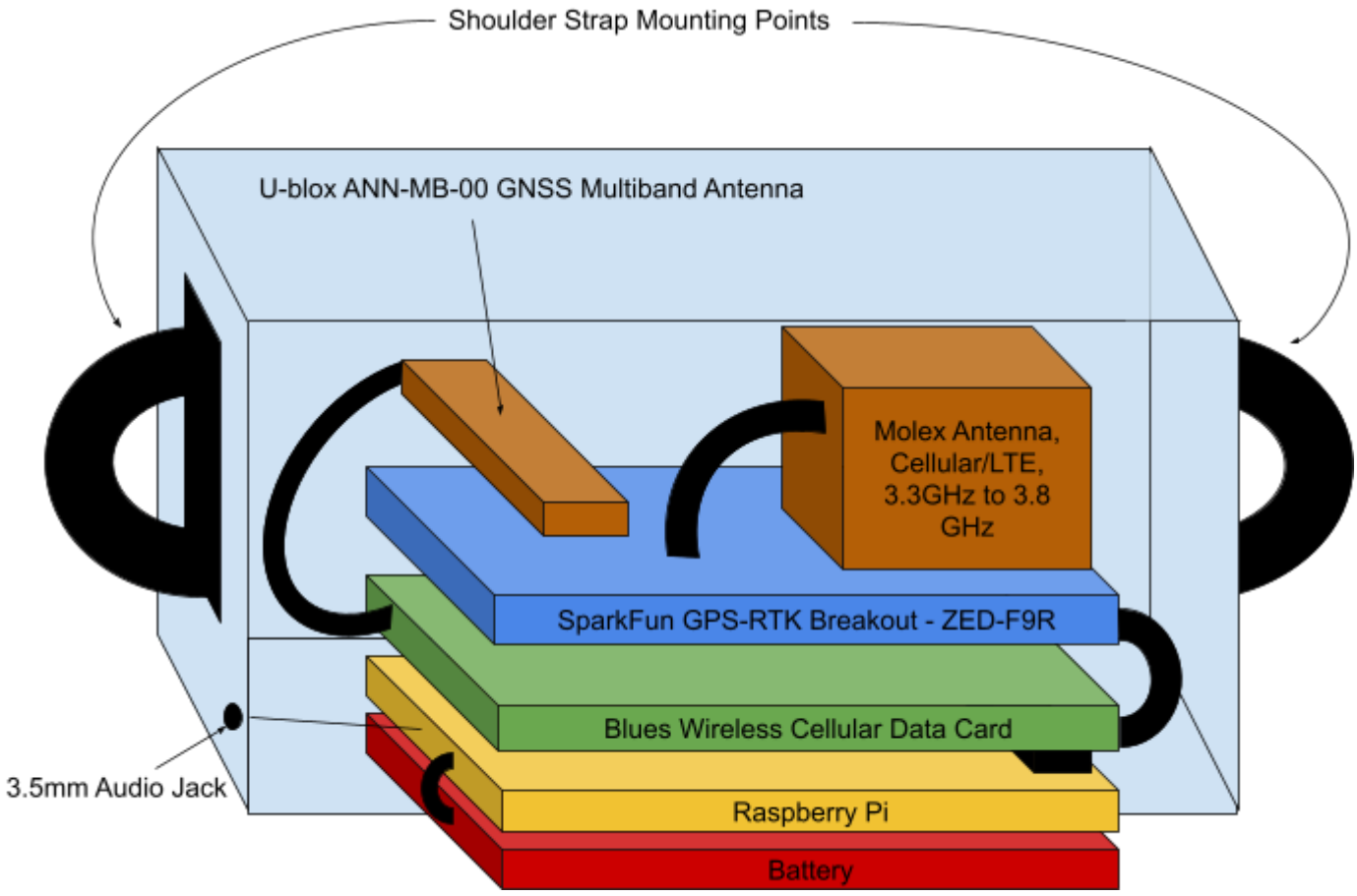

Fig. 3 The Physical Device and the Components

IV. Design Requirements

Based on our use case requirements in section II., we outline our design requirements below to meet these specifications.

A. Periodic Updates

This requirement is not meant to be particularly constraining, but rather as a tool to provide a good user experience. The frontend subsystem will keep track of the time elapsed since the last time it communicated the coordinates of the user with the backend. By default, it will wait 10 seconds before sending the next message to the backend server. However, if the backend indicates to the frontend that the user is near a turn or crosswalk, then the frontend will switch states and communicate with the frontend every 5 seconds, until the backend tells it to stop (ie. go back to default state). Assuming the backend server does not take more than 5 seconds to respond to the frontend request (which will be discussed in the latency section), we will comfortably meet this requirement.

B. Location Accuracy

To ensure a safe and good user experience, high location accuracy is crucial. In order to ensure high location accuracy, our system will use a U-blox ZED-F9R high precision Global Navigation Satellite Systems (GNSS) module. This module is able to gather location data from the four major GNSS constellations (GPS, GLONASS, Galileo and BeiDou) concurrently, allowing for sufficient satellite connections to maintain high accuracy. The module also provides IMU sensor fusion to maintain high accuracy when satellite connection is lost. Utilizing the ZED-F9R will allow us to achieve ~0.2 meter positional accuracy and ~0.3 degree heading accuracy. Maintaining these accuracies will allow the user to be confident that the correct information is given regarding location and direction with respect to their path.

C. Long Battery Life

Battery life is important to ensure that the user can travel all day without having to recharge the device's batteries. In order to meet this requirement, the device will have a 16 hour battery life so that a user can go all day without having to recharge. This number is based on the fact that the average person will sleep for about 8 hours a day, so the user should be able to use the device for the remaining 16 hours of a day. A battery of 26800 mAh at 5V is chosen to meet this requirement due to high availability and capacity. The Raspberry Pi draws about 1 Amp at 5V. The GPS/IMU draws about 130 mA at 3.3V. The Notecard draws about 150 mA at 3.3V, and will draw up to 750 mA at 5V during data transmission. Data transmission will not occur very frequently, so this number will not have a large effect on the overall power consumption. If we transmit data once every 5 seconds, we will assume that we will use about 750/5 = 150 mAh at 5 V. Combining these numbers gives us ~1335 mAh at 5V, which will give us about 20 hours of battery life. With 4 hours of slack in our calculations, we are confident that we can meet the 16 hour battery life use case.

D. Lightweight

Weight will heavily depend on the power consumption, given that the majority of weight in the system will be from the weight of the battery. The 26800 mAh battery we chose to use weighs 0.4 kg. With a use case constraint of 1 kg, 0.6 kg is left for the digital boards as well as the container for the device itself. The weight of the boards combined with the GNSS antenna is 0.35 kg. The last 0.25 kg will be used for the plastic container and mounting straps.

E. Latency

Following our use case requirements in section II., we want our device to respond within 1 second after the user's coordinate data is communicated. This specifically targets the backend server that will do most of the computational work regarding route planning and navigation.

To meet this requirement, we will first use an API (called HERE, which will be discussed in section V.) to plan the route for the user, at the beginning of the trip. Since communicating back and forth incurs a higher latency (and cost), we want to store the results of the request at the beginning of the trip, instead of communicating repeatedly during the trip. By doing so, we minimize the backend processing time to an estimate of at most 50ms. This will leave 950ms for the backend and frontend to communicate with each other, and for the frontend to transform the information given by the backend into text-to-speech feedback, which is ample time to fulfill our requirements.

In the case where the backend needs to reroute the user, it will communicate with the API again to receive the new route. According to HERE documentation, the 98th percentile latency is 350ms for routes under 100km. Although more constraining, this still leaves at least 650ms for all other communication for processing.

F. Reliability and Blind-unfriendly Crosswalk Avoidance

Our optimal backend system must always find a valid route between the user's current location, and their desired destination. It must also ensure that blind-unfriendly crosswalks are avoided unless there are no alternative routes available.

In the case where the user must or chooses to cross a blind-unfriendly crosswalk, the backend server must recognize that the user is planning to cross, and warn them that the crosswalk is blind-unfriendly.

G. Rerouting

Our optimal backend system must recognize when a user has deviated from the planned route, and reroute them within 30 seconds after they've deviated. Since the use case requirements state that the backend server will receive user coordinates at least every 10 seconds, it will have the capability to detect if a user is moving away from the next checkpoint. If their distance has increased from the next checkpoint for the last two communications (ie. 20 seconds), then the backend should assume that the user has deviated, and perform rerouting.

V. Design Trade Studies

There are several ways to implement each of our design requirements, and exploring the tradeoffs of each, and why we chose a certain method, will clarify our design.

A. Route Planning API

While designing our system, we contemplated whether we should implement route planning on our own, or use an API to do this for us. After some research, we realized that implementing route planning from scratch may be too ambitious, and instead decided to go with the latter option. There were several options for APIs that we could use:

1) Google Maps API

This was originally our first choice. The Google Maps API is quite comprehensive, has multi-language support (including Python, our backend language of choice), good map coverage, and the API is generally very easy to use. However, it lacks functionality to avoid certain areas or roads when doing route planning. There are custom alternatives to this; for example, Google Maps' Directions API allows users to specify whether they want multiple routes to the same destination. One way we could have leveraged this was to allow the API to find these routes, then filter out the ones that contained blind-unfriendly crosswalks. However, this would significantly increase our code complexity, while not providing a scalable or efficient solution.

2) OpenStreetMaps API

We also considered the OSM API for this task. However, while it is a free service, compared to the other APIs on this list, it does not provide nearly as much functionality, and is inherently scalable since it is a public API, and will not allow services (like ours) to make large amounts of calls to its API. Therefore, we decided this API would not be suitable for our needs.

3) HERE API

Finally, we discovered the HERE API. After looking into it, we believe it provides the best specifications for our needs. Specifically, it provides support for Python, and supports route planning and geocoding. In particular, it fills in the gaps of the Google Maps API, allowing us to specify points that should be avoided during route planning (ie. blind-friendly crosswalks). It also has reasonable latency (< 350ms for route planning), and is overall what we believe to be the best choice for our project.

B. Hardware

1) Processor

We chose a Raspberry Pi 4 due to the fact that most of the other hardware we are deciding to use is compatible with the board. We will also be using python to develop the software, and the Raspberry Pi provides a good platform to run all of the software that we need. We contemplated using an Arduino, however due to the need for more complicated threading, power requirements, as well as speed of development, we decided that the Raspberry Pi would be a better fit.

2) Location Device

We needed a high-precision location device that was capable of giving locational accuracy on the scale of the size of a sidewalk. Most crude GPS devices including most devices in smartphones are capable of providing GPS location within 10-40 meters of the device. This will not suit our requirements because of the possibility of giving false location information to the user. A benefit to the crude GPS devices is that they are cost effective, however we needed better accuracy so we decided to make the tradeoff of cost for higher accuracy. We chose the u-blox ZED-F9R GPS/IMU to provide us with high location accuracy even in poor satellite connection conditions.

VI. System Implementation

In this section, we hope to explicitly describe our implementation corresponding to our design and use case requirements.

A. Backend Server

The general implementation of the backend will be as follows:

1. Preprocessing and Database

At the start of the trip, the backend will first receive a message from the frontend, indicating the current user location, and the user's desired destination. Using these two data points, the backend will call the HERE API for route planning. Note that the backend will keep a database of unfriendly crosswalks. It will use this database as an input to the areas that need to be avoided when calling the API. The database will be implemented as a dictionary, where the postal codes are the keys, and the values are lists of JSON-formatted data, where each data point includes information on the latitudinal and longitudinal coordinates, and street names of a blind-unfriendly crosswalk in that postal code.

Once the HERE API returns a route, we will use a cache to store this route in local memory, and reference this cache from now on till the end of the trip, saving additional communication time between the API.

2. En-route

While the user is en-route to the destination, the backend will do as follows: Every time the backend receives a message from the frontend (which will communicate the user's latitudinal and longitudinal coordinates), the backend will use the Haversine formula to calculate the distance of the user from the next waypoint (the backend will keep track of an index that indicates which step of the route the user is on):

$$\operatorname{hav}(\theta) = \operatorname{hav}(\varphi_2 - \varphi_1) + \cos(\varphi_1)\cos(\varphi_2)\operatorname{hav}(\lambda_2 - \lambda_1)$$

where φ represents the latitude, and λ represents the longitude. Referencing the formula above, we will calculate the Haversine function of the differences in latitude and longitude as such:

$$\operatorname{hav}(\theta) = \sin^2\left(\frac{\theta}{2}\right) = \frac{1-\cos(\theta)}{2}$$

Finally, we can calculate distance:

$$d = r\,\mathrm{archav}(h) = 2r\arcsin\left(\sqrt{h}\right),$$

where *h* is *hav(Θ)*.

After this distance is calculated, we will return a String message to the frontend, which should then be directly used as text-to-speech feedback for the user.

When we detect that a user is within 20m of a turn and/or crosswalk, our system will increment its index to this next step of the route (ie. the turn, or crosswalk). During this step, we expect the system to give them feedback telling them to turn and/or cross. Once the user is 20m away from that step, we will assume that they have turned or crossed, and increment our index to the next step of the route.

3. Rerouting

The backend will keep track of the most recent user coordinates. When it is given a new set of coordinates, and calculating the Haversine distance on this point and the next waypoint gives a greater distance than the previous coordinates, it will assume the user is moving away from the next waypoint. If this happens two times in a row, then the backend will reroute the user by calling the HERE API using the user's most current coordinates, and their desired destination. Note that this may occasionally still give the same route. There are multiple ways to deal with this issue: we may choose to avoid the next step on the original route, so that the API is forced to choose a new route. Or, we may choose to set a fixed waypoint (possible using the HERE API) between the direction the user is currently walking in, and the destination, so that the API plans a route in that direction. Yet another way of dealing with this is to simply let the backend continuously reroute the user (every two iterations), until the user is closer to an alternative route. These are all viable solutions, and we will experiment with and choose the one offering the most simplicity and performance.

B. Front End

The front end will have three separate components, the location data gathering system, the cellular data interface, and the auditory feedback system.

1. Location Data

We will use a u-blox ZED-F9R GPS/IMU device in order to determine the user location with high accuracy. This device utilizes IMU sensor fusion to be able to provide high accuracy location information even in poor satellite connection conditions. We will communicate to this device via an I2C serial bus. This device provides latitude and longitude coordinates that will be able to be correlated to the coordinates provided by the HERE API. The device also provides heading information based on the cardinal directions. This information will be given to the backend server to determine if the user is on course and where they have to go next to reach their destination.

2. Cellular Data Interface

We will use a Blues Wireless LTE-M cellular data card to access the HERE API that the back-end needs to communicate with. We will communicate to this device via an I2C Serial Bus. The cellular data card will allow us to access the internet anywhere as long as we have access to cell towers. In remote areas this may be a problem, however as long as a user can use their cell phone they will be able to use this system as well.

3. Auditory Feedback

The system will provide a 3.5mm audio jack for the user to plug headphones of their liking into. Audio will be played periodically when the back-end determines that it is necessary. The front-end will then take in the back-end string of text and run it through an offline text-to-speech engine (pyttsx3). This audio text-to-speech file will then be played out of the 3.5mm audio jack through the Raspberry Pi audio interface.

C. Physical Device

The physical device will be a plastic box containing all of the components outlined in Figure 3. This box will be 3d printed and will have straps to attach to a person's shoulders so they can wear the device like they would a backpack. The user must wear this on their back due to the fact that the device calibration will be assuming that the user is wearing it in the correct orientation on their back. This will ensure that the location information is correct.

VII. Test, Verification and Validation

A. Tests for Battery Life

The battery life use case is 16 hours. To test the battery life of the device, we will use the device in various conditions for 2 hours. We will then see how much energy the device consumed and then multiply the energy value by 8 to achieve a 16 hour value of battery consumption, and compare against the energy capacity of our battery.

B. Tests for Location Accuracy

The location accuracy use case is 1 meter error from the actual location of the user. To test this, we will stand in various spots with known coordinates and then programmatically see what coordinates the device gives back. From these 2 sets of coordinates, we can determine the distance between them and see if the error was less than 1 meter. We will do this test at 20 various locations and if any reading has more than 1 meter error we will have considered the test to have failed.

C. Tests for Latency

The latency use case is less than 1 second latency from the point of time when we gather location data to the time when auditory feedback is given. To test this we will insert a timer into the software starting when we gather location data. The timer will stop when audio feedback begins. This elapsed time will determine if we have met the 1 second latency use case requirement

D. Tests for Weight

We will use a scale to determine the weight of the device. The use case is 1 kilogram, we will have considered to meet the requirement if our device is less than 1 kilogram in weight.

E. Tests for Reliability/Crosswalk Detection

We require a use case of 100% avoidance of blind-unfriendly crosswalks. To test this, we will run the routing algorithm 100 times and see if it includes any blind-unfriendly crosswalks. The algorithm should succeed every time in order to consider a success.

VIII. Project Management

A. Schedule

Our schedule is outlined in Figure 4. We aim to have the sensor systems integrated with the back-end around November 4th. We plan to continue improving our system based on the feedback we get from integration until November 15, where we aim to have a fully built and usable device. We have 2 weeks of slack after that point to improve on the functionality of the device, where we will be finished on November 29th.

B. Team Member Responsibilities

Colin will be working on the hardware implementation, the construction of the wearable device, and the process communications. Zach will be working on the back-end of the system including the routing and deciding what information to give back to the user based on the location information from the front-end.

C. Bill of Materials and Budget

See Figure 5 for the bills of materials and budget.

D. Risk Mitigation Plans

The critical risk factors in our design revolve around two major factors.

The first factor is the lack of wireless communication experience that our group has. Our device must communicate to the internet in order for our implementation to work, and we are counting on the cellular data card that we chose to perform our needs. However, if the cellular card does not work out we may be able to implement some sort of routing scheme using an offline database of streets and locations.

The second factor is the complexity of the user tracking with respect to the path that we are providing for them. This includes the re-routing functionality and making sure that the user is in fact where we think that they are. Given the amount of time that we have to make this work (about 7 weeks), this will be challenging. To mitigate this risk, we may reduce the scope to a smaller area that we have more information about to be able to provide a higher accuracy of feedback to the user.

IX. Related Work

1. Google Maps

Google Maps is a web based location service which provides directions to users based on their current location and their desired destination. Google Maps is made for smartphones and computers to be able to provide directions to users. Access to the real time directions functionality is limited as you must go through a smartphone for those features.

X. Summary

Our design aims to provide a high accuracy direction service for the visually impaired. A combination of GPS/IMU data will provide high accuracy location data to the system to allow for a reliable and pleasant user experience. With careful attention to power usage and weight, the user will be comfortable while using the device. Some upcoming challenges include the complexity of user tracking algorithms to ensure a high feedback accuracy so the user is not told false information. Other challenges include the combination of all of the physical hardware parts to make sure that all of the units work as needed for the system to be able to function properly.

Glossary of Acronyms

GNSS - Global Navigation Satellite Systems
GPS - Global Positioning Service
IMU - Inertial Measurement Unit
MQTT – Message Queuing Telemetry Transport
OBD – On-Board Diagnostics
RPi – Raspberry Pi

References

[1] Blues Wireless, Cellular Data Card, 2019
[2] Bhat, Natesh, “pyttsx3,” Jul 6, 2020.

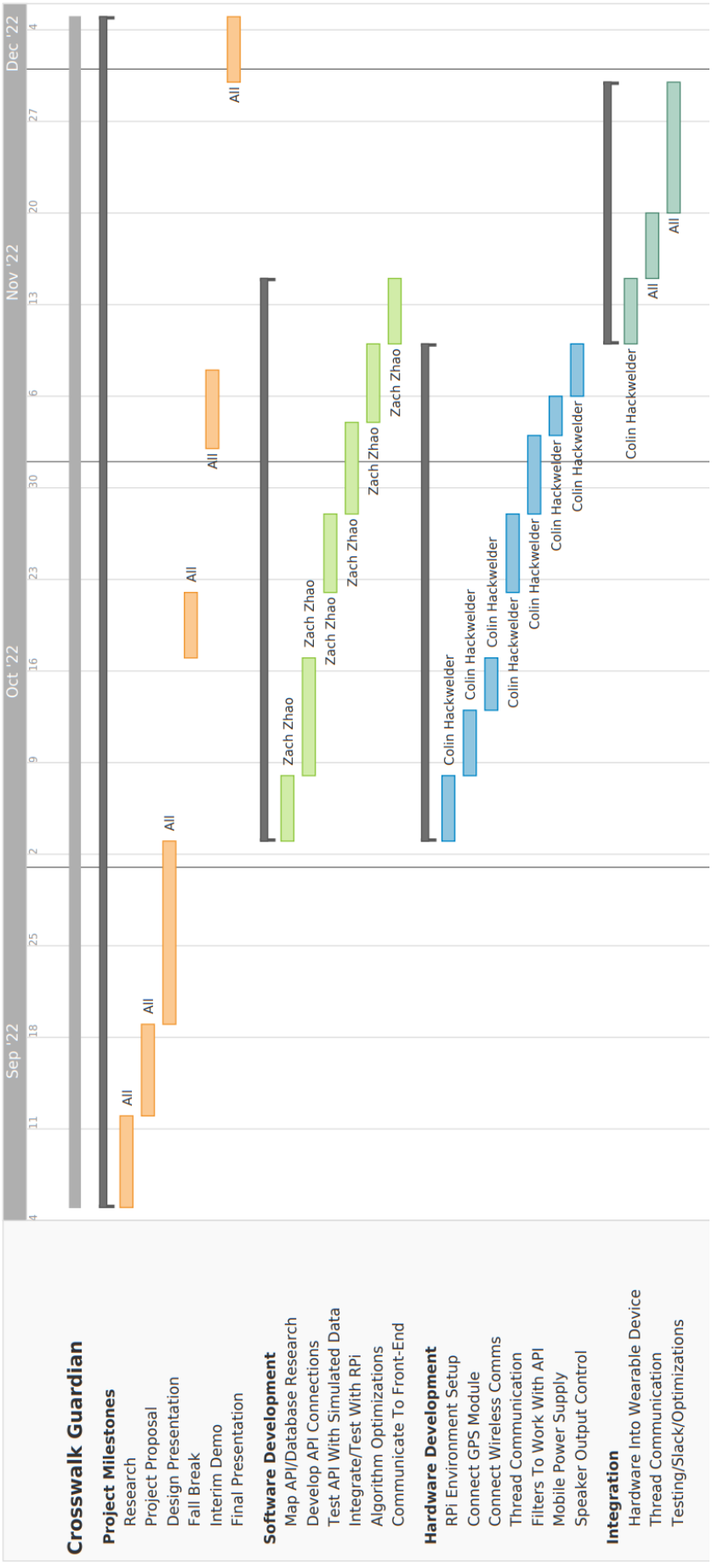

Fig. 4. Schedule - Gantt Chart

Description	Model #	Manufacturer	Quantity	Cost	Total
Raspberry Pi 4 8GB	Raspberry Pi 4 Model B	Raspberry Pi Foundation	1	0	0
Raspberry Pi Cellular Datacard Starter Kit	CARR-PI	Blues Wireless	1	79.00	79.00
u.FL to SMA Adapter Cable	WRL-09145	SparkFun	1	5.50	5.50
Grove to Qwiic Adapter Cable	PRT-15109	SparkFun	1	1.60	1.60
GPS-RTK ZED-F9R	GPS-16344	SparkFun	1	289.95	298.95
26800 mAh Portable USB-C Charger	CH-26800	Charmast	1	33.99	33.99
					419.04

Fig. 5. Bill of Materials