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 15 seconds on the distance remaining on the current step. If the user is within 15m 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 near 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.

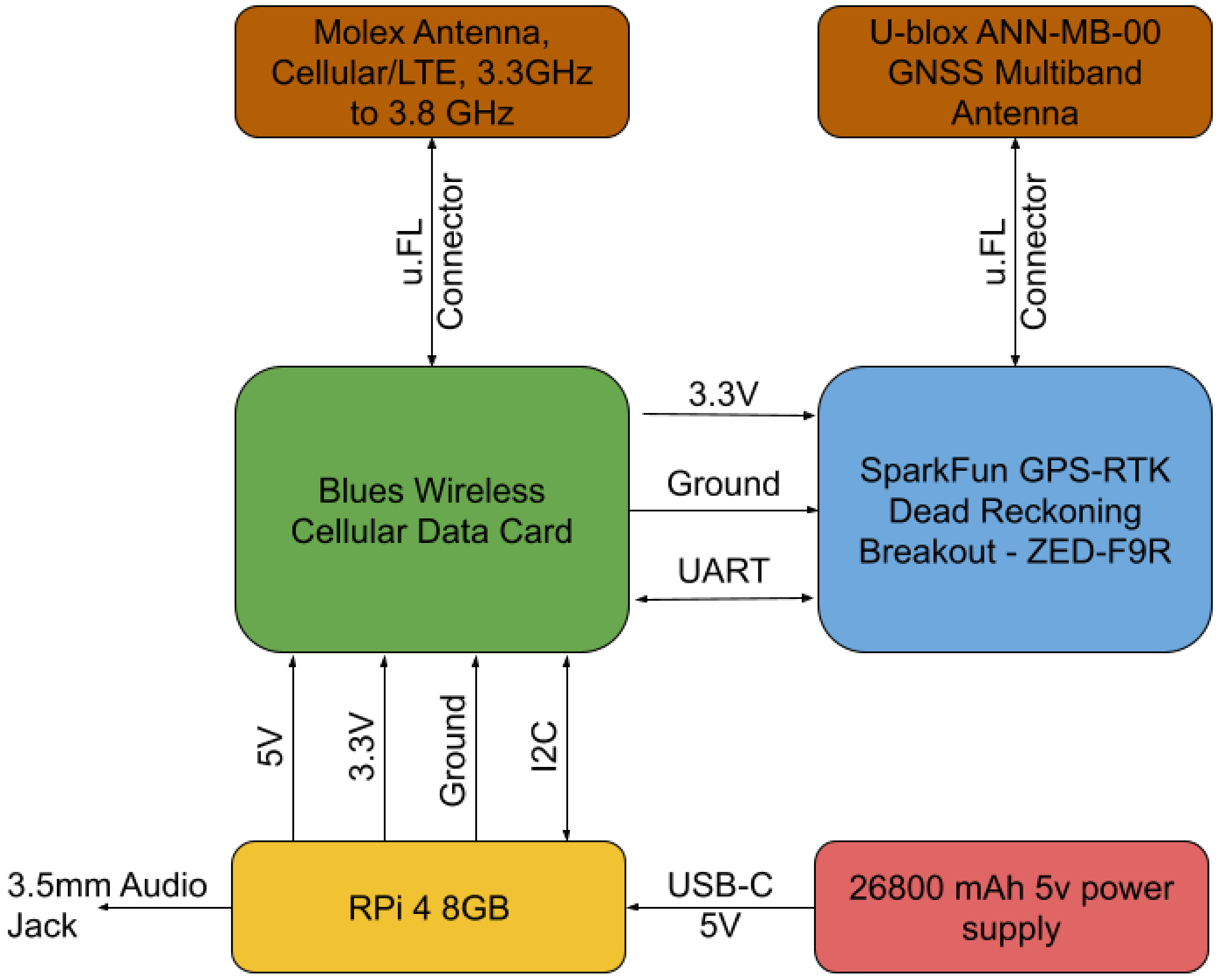
H. Rerouting

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.


Molex Antenna, Cellular/LTE, 3.3GHz to 3.8 GHz
U-blox ANN-MB-00 GNSS Multiband Antenna
u.FL Connector
u.FL Connector
Blues Wireless Cellular Data Card
3.3V
Ground
UART
SparkFun GPS-RTK Dead Reckoning Breakout - ZED-F9R
5V
3.3V
Ground
I2C
3.5mm Audio Jack
RPi 4 8GB
USB-C 5V
26800 mAh 5v power supply

The hardware system block diagram

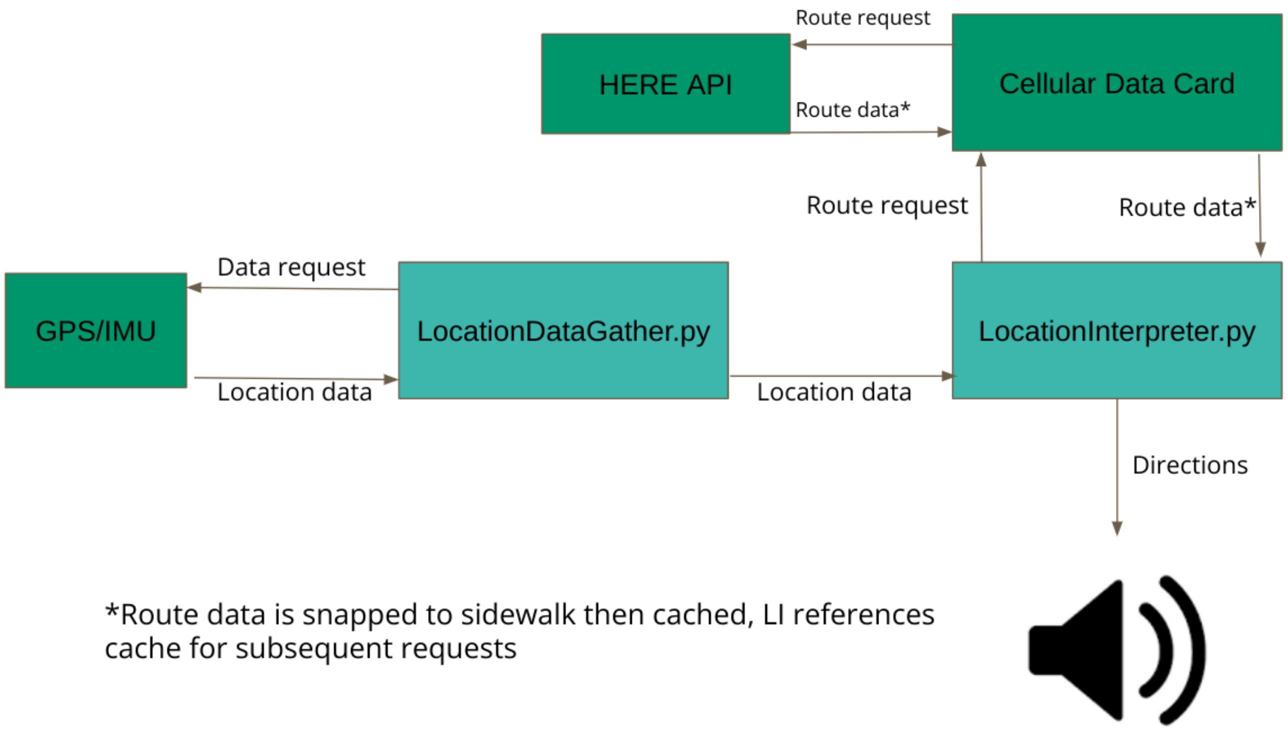


Fig. 2. The software system block diagram running on the Raspberry Pi 4

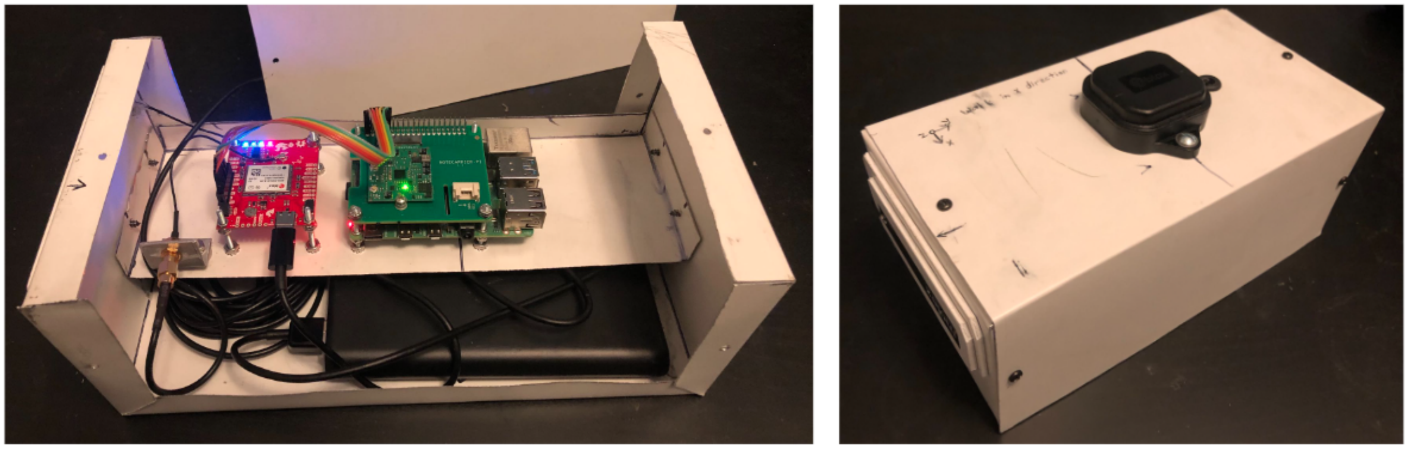


Fig. 3. Overall system

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 15 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 ~1 meter positional accuracy and ~5 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. These accuracies are also required so that the device can reliably determine what side of the road that the user is on so that it can avoid necessary blind-unfriendly crosswalks.

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 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 may be 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, Database and Caching

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, and will use this database as an input to the areas that need to be avoided when calling the API. Since the database itself is not the core focus of our project, we will use a lightweight dictionary implementation, 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. Our cache will be implemented as a nested list. Each element in the outer list will represent a checkpoint en-route to the destination, and each element will contain the directions to be performed at that location (ie. turn left/right, etc.), and also the latitude and longitude coordinates of that checkpoint.

Once we have the route results from the API, the directions must be snapped to the sidewalk that the user will be on when starting the route. This is to ensure that we are properly avoiding blind-unfriendly intersections only when they need to be crossed. In order to do this, a graph of all sidewalk corners is kept in our database, and we iterate through the API route, snapping to the closest/next sidewalk corner until we reach the end of the route and have a full route following the sidewalk.

2. Coordinate to Distance Conversion

In order to reliably route our user, we must be constantly calculating the distance between the user and their next checkpoint en-route to their destination. However, both the GPS coordinates and the routing data that the HERE API gives us consists of latitude, longitude coordinates. Therefore, we need a method to translate these coordinates into actual distance.

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):

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

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:

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

Finally, we can calculate distance:

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

where h is $hav(\Theta)$.

After this distance is calculated, we may choose to provide feedback to the user, if appropriate. This will be discussed further in the subsections below.

3. State Transition Model/Routing Scheme

As aforementioned, the HERE API returns a route consisting of checkpoints, represented as latitude, longitude coordinates. However, the user obviously does not need to be exactly at a coordinate, in order for our system to realize that the user has reached the checkpoint (that would be almost impossible to accurately measure). Therefore, we provide some leeway: If the user is within 5 meters of the coordinates of a checkpoint (measured by the Haversine Function), then our system will start routing the user to the next checkpoint.

Using this scheme, a similar issue arises. What if the user is still on the originally prescribed route, but was not close enough to a checkpoint (within 5 meters, to be exact), before proceeding to the next checkpoint? Our system must be able to differentiate between these "slightly-off" errors, versus when the user really has deviated from the route, and proceed accordingly. This scenario provides a need to implement a state transition model.

Our model is shown below in Fig. 4. The system's start state, and the state it will predominantly stay in, is the *Same Checkpoint* state. This state signifies that the user is on their way to the upcoming checkpoint. If at any point, the device detects the user's distance with the checkpoint is increasing, it will initiate a reroute by contacting the HERE API (indicated by the *Rerouting User* state). Meanwhile, if the device sees that the user's distance to the checkpoint is less than 15 meters, then it will proceed to the *Near Checkpoint* state. In the standard case, if the user's distance to the checkpoint becomes less than 5 meters, then we increment the cache, which will allow us to begin routing the user to the next checkpoint, and revert back to the *Same Checkpoint* state. However, if the user's distance to the checkpoint ever increases in this state, then we must refer to the user's distance to the **next** checkpoint. If the user's distance to the next checkpoint has also increased, then the user has probably deviated from the route, and we will reroute them. However, if it has decreased, then the user is most likely still on course, but has just slightly missed the checkpoint radius, so we still increment the cache, and continue routing them to the next checkpoint.

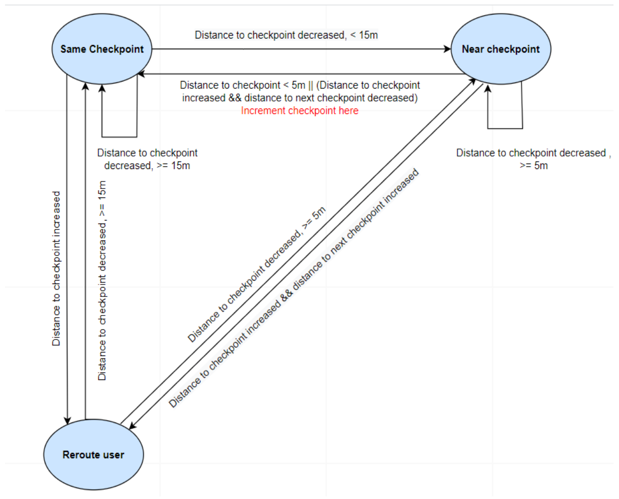

Same Checkpoint
Near checkpoint
Reroute user
Distance to checkpoint decreased, < 15m
Distance to checkpoint < 5m || (Distance to checkpoint increased && distance to next checkpoint decreased)
Increment checkpoint here
Distance to checkpoint decreased, >= 15m
Distance to checkpoint decreased , >= 5m
Distance to checkpoint increased
Distance to checkpoint decreased, >= 15m
Distance to checkpoint decreased, >= 5m
Distance to checkpoint increased && distance to next checkpoint increased

Fig 4. The state transition diagram allowing us to reliably route the usesr

4. Feedback to User

Referencing our Design Requirements section A and our state transition diagram, we will only be providing verbal feedback to the user every 15 seconds if they are in the *Same Checkpoint* state, and every 5 seconds if they are in the *Near Checkpoint* state. We will still be requesting user coordinates from the GPS around once every second, but will not always be providing audio feedback in order to reduce unnecessary noise to the user.

There are certain times where we must "force" the system to give feedback, even if not enough time has passed yet since the last feedback. The first case is when the system detects a distance increase in the *Same Checkpoint* state. Then, we need to reroute the user and provide feedback immediately on the updated route. The second case is when the user is within 5 meters of the checkpoint. Then, we need to immediately tell them to perform an action (ie. turn) and proceed to the next checkpoint.

The feedback given to the user while in the *Same Checkpoint* state is as follows: **Continue on (*street name)* for x meters.** We extract the street name from the data that the HERE API returns, and we calculate **x** using the Haversine Function. In the *Near Checkpoint* state, we give feedback as such: **In x meters, (*perform action)*. x** is calculated as before, while ***perform action*** is the action prescribed by the HERE API data. Finally, when the user is near an unfriendly crosswalk (within 10 meters, which the system checks for every iteration), it will provide feedback: **You are near an unfriendly crosswalk. Please be careful.**

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 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 communicate to this device via UART serial protocol. 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. Initially when designing the system, the IMU sensor fusion was a plus due to the fact that we thought it would allow for a more precise location of the user to be derived. However, the IMU was meant for faster moving and more rigid applications where wheel-tick sensor inputs can be provided. For our application, the IMU fusion hurt our accuracy when compared to just using a 3D satellite fix.

2. Cellular Data Interface

We 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 allows 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 is an aluminum box containing all of the components outlined in Figure 3, sitting inside of a drawstring bag. The box was drawn out on a piece of aluminum and then bent to provide a bottom carriage portion as well as a top cap. The two parts are screwed together, protecting the internals.

VII. Test, Verification and Validation

A. *Tests for Battery Life*

The battery life use case is 16 hours. To test the battery life, a USB ammeter was used to provide peak current usage information. The device draws 1.1A at 4.93V at peak usage. With a 26800 mAh battery at 4.93V, we are able to achieve a battery life of 24 hours

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 walked with the device along known locations and viewed the distance error from the known locations after gathering the location data. In open environments we are able to achieve 0.9 meter error. This satisfies our use case requirement of 1 meter. However, in urban environments up against tall buildings, we get 5 meters error. This does not meet the use case requirement of 1 meter, however we are still able to make the device work by assuming that the user will not cross the middle of the street and they will stay on the sidewalk. We then snap their location to the nearest point on the sidewalk to find where they most likely are. Location accuracy is improved when near an intersection so we can rely on the coordinates from the device to make sure the user is not deviating from the route at intersections.

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 inserted a timer into the software starting when we gather location data. The timer stops when audio feedback begins. When not rerouting (using a cached route), the latency is 0.0046 seconds. However, when rerouting, our latency is 1.72 seconds. This is primarily due to the fact that the Blues Wireless cellular data card acts as a proxy when communicating to external APIs, so our latency is double that of what it would be without going through a proxy. Due to the fact that we do not reroute often, this extra delay is not detrimental to the design, as we can regather the users location data again once we get the rerouted data back.

D. *Tests for Weight*

We used a scale to determine the weight of the device. The use case is 1 kilogram. The weight of the device was 0.85kg, meeting our use case requirement. We believe that a heavier design that provides a more rigid structure would have provided better IMU sensor fusion results due to the fact that the GPS antenna would not be moving as much relative to the IMU. However, this more rigid structure would most likely go over our 1kg use case requirement.

E. *Tests for Reliability/Crosswalk Detection*

We require a use case of 100% avoidance of blind-unfriendly crosswalks. To test this, we ran the routing algorithm 50 times, and successfully avoided blind-unfriendly crosswalks each time, meeting our use case requirements.

Further, we want to test that the device detects blind-unfriendly crosswalks that are nearby (within 10 meters) 100% of the time. To do so, we ran 10 tests, where we began outside of the 10 meter range, and walked within the 10 meter range. We were able to receive feedback from the system indicating that we were near an unfriendly crosswalk every time, which meets our use case requirements as well.

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, the process communications, and interfacing with the hardware, as well as helping Zach with some extra software design and validation. 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.

We ended up not using the grove to I2C connector due to the lack of I2C libraries to communicate to the ZED-F9R. Instead we used UART with some extra wires we had. We also used a set of earbuds and USB-C cables that we already had to get feedback from the device and power the device.

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. Ethical Issues

Our design raises several ethical issues. One of the main ethical issues is that our design is tracking the location of users and some users may not like their location being tracked. Although we are not keeping their location data, we are still interfacing with the HERE API which is an external API that we have no control over. If there is a security breach of the HERE API, or our device itself, a user's location history may be trackable.

Another major ethical issue is that our device is attempting to tell users where to turn, and if we tell them to turn in the wrong direction, the user may turn into a dangerous location, such as an unsafe street or corridor. This could have adverse side-effects on the user in extreme situations. However, we advise that users still be using cane when walking to make sure that they are not walking off of the sidewalk into a street or other dangerous situation.

Other ethical issues have to do with the reliability of the device, specifically regarding the possibility of losing connection to the cellular data network or losing power halfway through the route. If the device loses power or we can no longer reroute the user, they may be stuck halfway through the route and the device would not be able to route them the rest of the way to their destination. The user would be negatively affected in this situation as they may not know where to go.

X. 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.

XI. 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.

Our group learned a lot of lessons. One lesson on the hardware side of design is that more attention should be given to the information on the datasheets of the devices to make sure that the device will actually perform as needed. In our case, we thought that the GPS/IMU sensor fusion would help our design, however the IMU sensor fusion actually hurt our design due to our slow-moving, pedestrian use case. We ended up not using the IMU portion of the device as we could get better accuracy without it. Another lesson we learned about was to fully research and understand the limits and flaws of APIs that we intend to use. Using the HERE API as a specific example, it would often switch between side-specific and non side-specific street coordinates, which was a huge pain point for us during the implementation of routing.

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.

[3] *Build apps with here maps API and SDK Platform Access: Here Developer*. Build apps with HERE Maps API and SDK Platform Access | HERE Developer. (n.d.). Retrieved December 17, 2022, from https://developer.here.com/

[4] Google. (n.d.). Google maps platform | google developers. Retrieved December 17, 2022, from https://developers.google.com/maps

[5] Upadhyay, A. (2018, June 20). *Haversine formula - calculate geographic distance on Earth*. Haversine formula - Calculate geographic distance on earth. Retrieved December 17, 2022, from https://www.igismap.com/haversine-formula-calculate-geographic-distance-earth/

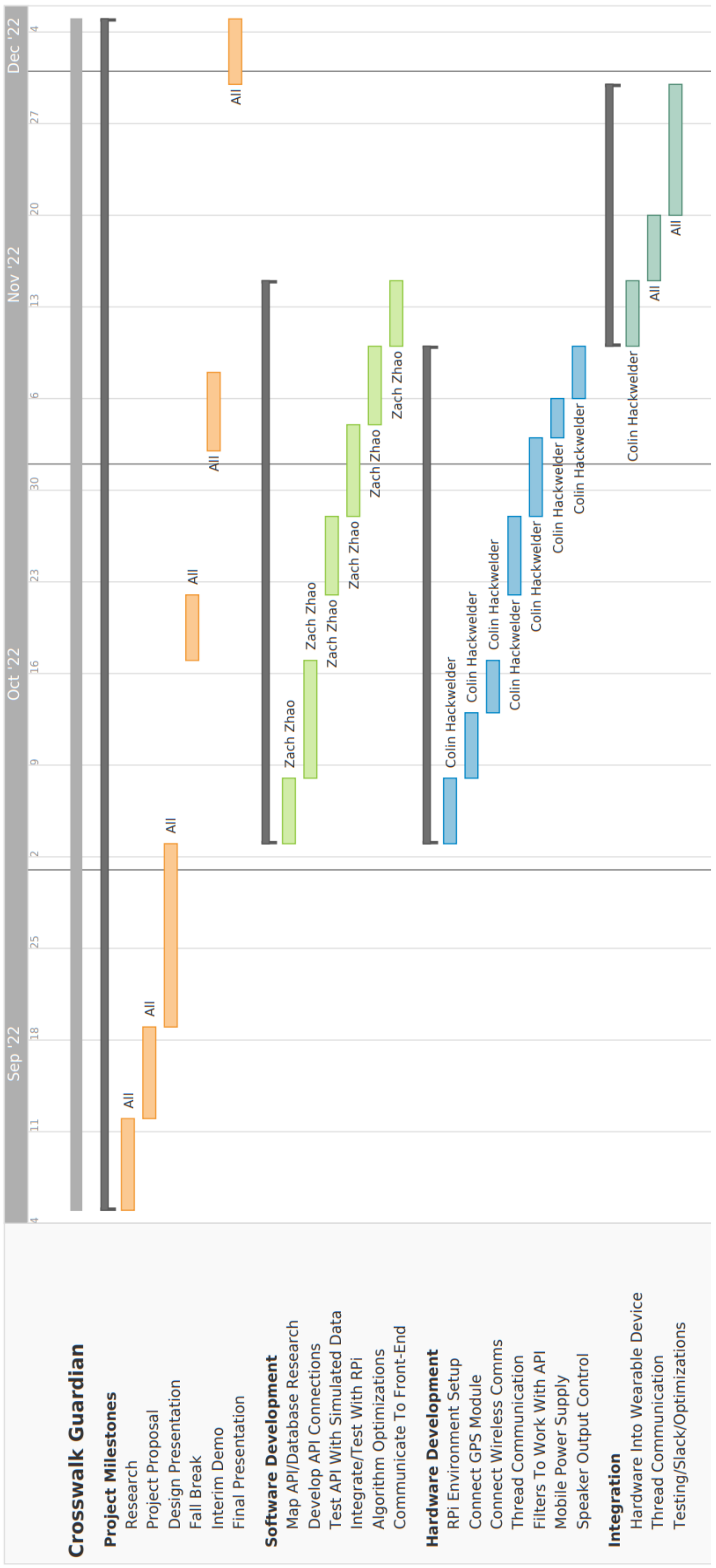


Fig. 4. Schedule - Gantt Chart

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	QTShine	1	26.95	26.95
Drawstring Bag	BG-22x18	BeeGreen	1	9.99	9.99
					421.99

Fig. 5. Bill of Materials