Indoor Positioning System and Geo Fencing

FootPrint

A PROJECT REPORT

BY

Saahil Kamath,

Rishank S Nair,

Manav Gurumoorthy &

Aishwarya Ray

Guided by Prof. K.V.S Hari

Electrical and communication

Indian Institute of Science

Bangalore, 560012

July 2016

ACKNOWLEDGEMENT

We would like to thank Prof.K.V.S.Hari for giving us the opportunity to work on this project. We would also like to thank him for his valuable guidance and support, which was a constant source of inspiration throughout the project tenure.

We also owe a debt of gratitude to Inertial Elements for sharing their source code with us which gave us insight to the working of the device as well as Bluetooth communication. Their website showed us how to use the device, where to attach it and how it's range limitations.

We would also like to extend our heartfelt thanks to our parents, for their support and inspiring us to work.

We also express deep and sincere gratitude to the other interns in the lab whose encouragement, suggestion and very constructive criticism have contributed immensely to the evolution of ideas in our project.

Footprint: An Open-source Android based Indoor Positioning application with an integrated Geo-fencing algorithm using a foot-mounted Inertial Navigation System.

ABSTRACT

The accurate positioning of an object inside a closed space has been required for a variety of reasons. Though it has been around for approximately two decades, the use of foot mounted inertial navigation systems hasn't really caught on in a consumer space. As a solution, we present an open source Android based application along with an existing foot mounted navigation module.

In addition to functioning as a positioning system the application also implements a geo-fence. When the device crosses a user defined boundary, the phone vibrates as an alert. This can be used for monitoring the elderly or toddlers in order to prevent them from exiting a user defined safe zone.

Table of Contents:

1. Introduction---------- 5
2. About the Module---------- 6
3. DaRe App and Bluetooth Communication---------- 8
4. Command Flow Sequence---------- 9
5. Position Plotting---------- 13
6. Ray Casting---------- 17
7. Individual Contributions---------- 21
8. The “Footprint” App---------- 23
9. FAQs---------- 25
10. Conclusion---------- 26
11. Future Works---------- 27
12. Bibliography---------- 28

Chapter 1
INTRODUCTION

The applications of an IPS along with an integrated Geo-fence are infinite. Our project focuses on using an Indoor positioning system along with a geo-fence to monitor the movements of an elderly person. If the elderly person moves out of the user defined geo-fence, the phone vibrates as an alert.

The Footprint system is a combination of the "**Footprint**" app along with the OSMIUM MIMU22BT inertial navigation module. The OSMIUM MIMU 22BT module can either be foot mounted or mounted onto a walking stick in accordance to the wearers' comfort.

The application records the local co-ordinates taking the origin as the point at which tracking was started i.e. (0, 0, 0) and converts it to global co-ordinates on the XYZ plane and plots the real time position of the wearer.

The app also creates 3 text files stored in a folder named 'Footprint Coordinates' in the phone's memory each time the application is run. The first file contains the step count, total distance travelled in m, total time and average speed in Kmph. The second contains the X,Y,Z co-ordinates of the wearer in the duration of the tracking and the third text file contains a log of the start time, end time and android version on which the app is running.

GETTING STARTED

We started the development by understanding the source code of the DaRe app which was developed by the same team that made the module. The purpose of this was to understand how the data was being transmitted by the module and received on the application.

We then started on developing our application. The first step was to construct the front end of the app i.e. the GUI (Graphical User Interface). After a lot of step backs we realised that the only way to code the back end of the app was to use a systematic and methodical approach. The back end was divided into modules for Bluetooth communication, position plotting and a module to create the geo-fence.

The first challenge was working with Bluetooth in android which required extensive and in depth knowledge in working with threads in Java. We read through various Bluetooth terminal app source codes as well as learnt through videos and sites on the internet.

Once Bluetooth connection was established the next step was being able to read and write data over the connection. Using the same approach as above we achieved data transfer over Bluetooth. Once this was done, we needed to understand how to decipher the data being read and also to understand the functioning of the MIMU22BT.

Chapter 2
ABOUT THE MODULE

About the MIMU22BT:

The Osmium MIMU22BT is a miniaturized Multiple Inertial Measurement Units (MIMU) based wireless inertial navigation module with on-board floating point processing capability and data memory. A Bluetooth module provides a wireless data link. It has on-board Li-ion battery charging circuit which makes it battery operable and enables dynamic switching between battery and USB powering. The IMUs' placement scheme, with their sensitivity axes in the opposite directions, mitigates effect of systematic errors. A dedicated micro-B USB connector is provided for JTAG programming and debugging. The module consists of a single board without any protruding connectors and has low profile.

MIMU22BTP: Encased MIMU22BT

The small form factor of MIMU22BT module makes it suitable for foot-mounted inertial navigation and other applications based on wearable sensors. Presence of an on-board floating point processing capability, along with four IMUs, makes navigational computation possible inside the module itself, which in turn results in very accurate tracking of wearer.

MIMU22BT is supported by an open source embedded code written in C, which is easily configurable to run any user implemented algorithm. The software is configurable to work as a standalone ZUPT-aided inertial navigation system and as a displacement & heading change sensor. MATLAB code is also available for communication.

The module has wide applications in pedestrian tracking and gait analysis. It is an easy to use research platform for Biomedical researchers, Behavioural scientists and Ubiquitous Computing researchers.

Feature summary:

• Inertial measurement using four nine-Axis IMUs (Gyro +Accelerometer + Magnetometer)

• Processing using AT32UC3C 32-bits floating point microcontroller with 512 Kb internal flash memory

• Wireless communication enabled by Bluetooth serial interface module version v3.0

• USB 2.0 communication through USB micro-B connector

• Onboard 8 Mb DataFlash memory and Pressure sensor

• Power options: Li-ion battery, USB

• On-board USB battery charging

• Max continuous current: 100mA

• LED indications for battery charging, device powering and general-purpose

• Power on-off control switch

• IMUs' orientation to minimize systematic errors

• JTAG programming and debugging through a dedicated USB micro connector

• Weight & Size: 3.5 g, 20.9mm x 22.7mm x 5.5mm.

Chapter 3

DaRe APP AND BLUETOOTH COMMUNICATION

The DaRe app was developed to specifically work along with the OSMIUM MIMU22BT. As we had to work with this module we thought this was a logical first step.

We went through the entire source code as well as the instructions manual for the module. The purpose being, to understand how the data being measured by the module was being transmitted to the DaRe application. Deciphering the code gave us crucial information on the way the app is supposed to receive data from MIMU22BT and process it.

Function

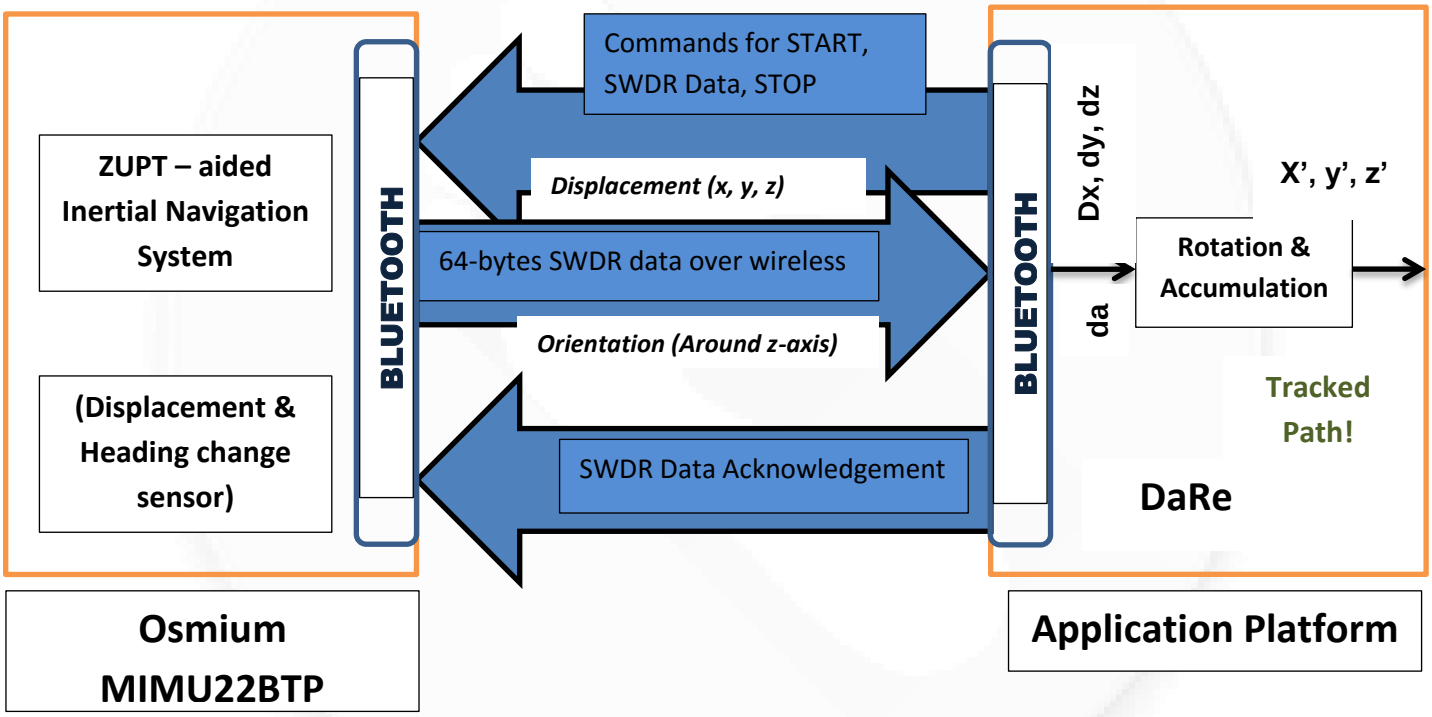


Fig 1.1: *Interfacing of Osmium MIMU22BT with an application platform*

Chapter 4
COMMAND FLOW SEQUENCE

Step1: START – Starts receiving data packet from Osmium.

Step2: Sends acknowledgement for the last data packet received from Osmium.

Step3: SWDR – Perform stepwise dead reckoning (SWDR) on the received data.

[Steps 2 & 3 are continuously executed till the STOP command is received. On receiving the STOP command, Step4 is executed.]

Step4: STOP - (I) stop processing in Osmium.

(II) Stop all outputs in Osmium.

Step-wise Description

Step1: START – Starts receiving data packet from Osmium.

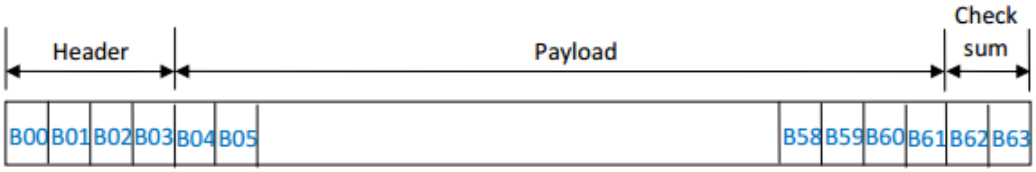
The command to START transmitting stepwise dead reckoning data consists of 3 bytes.

The START command results in the following:-

A. Osmium transmits a 4-byte acknowledgement in response to START command from the application platform:{A0,34,00,D4}

   A0: Acknowledgement state
   34: Start command state
   00 D4: Checksum

B. Following this 4-byte acknowledgement, the Osmium MIMU22BTP sends 64 bytes of tracking data in the form of packets via the Bluetooth connection. Below figure illustrates how the data packet looks like:



B00: State of the Header
B01-B02: Data packet number
B03: Number of bytes in Payload

B04-B07: dx: Displacement in x (Type float)
B08-B11: dy: Displacement in y (Type float)
B12-B15: dz: Displacement in z (Type float)
B16-B19: da: Change in angle around z axis (Type float)

B20-B59: 10 entries (4 bytes each) of a 4x4 symmetric covariance matrix

B60-B61: Step counter

B62-B63: Check sum

Data Packet Description

The 64-byte tracking data is divided accordingly:-

i. **Header** – The first 4 bytes are 'Header':

a. Among these 4 bytes first is the header which is STATE_OUTPUT_HEADER which is 170 (0xAA).

b. 2nd and 3rd byte jointly tells the data packet number sent by the device since the beginning of stepwise dead reckoning. Thus the packet number is a 16-bits variable.

c. 4th byte depicts the payload i.e. the number of bytes (0x3A) this data packet contains which consists of required information asked by the user.

ii. **Payload** – The next 56 bytes consists of 14 "float" type data elements. (14 * 4 (size of 1 float element) = 56)

a. First 4 elements consisted of the delta vector i.e. the change in position x, y, z and the angle of rotation around z-axis (the change in the angle of the x-y plane). As these elements are of type float, thus are of 32 bits. The change in position is between two successive ZUPT instances, i.e. the displacement vector from one step to the next one.

b. The other 10 elements of the payload are used to form the covariance matrix, which is a 4x4 symmetric matrix, thus 10 elements. These are not used in the step-wise dead reckoning. These are used in data fusion from two MIMU22BTP devices in case of dual foot mounted system.

iii. **Step Counter** – Following the Payload is the 2-byte Step counter which is a counter taking record of ZUPT instances observed. This is the number of times velocity of MIMU22BTP goes down below the pre-defined threshold (i.e. reaches zero velocity). Hence, it is the number of steps taken by the wearer when the device is used for foot-mounted pedestrian tracking.

iv. **Checksum** – The last 2 bytes consist of the check sum which is the sum of all the bytes received prior to these. This is used to cross-check the correctness of the data transferred.

Step2 : Sends acknowledgement for the last data packet received from Osmium.

ACK consists of 5 bytes:

- 1st byte: 01

- 2nd byte: P1 (First byte of data packet number of last data packet received)

- 3rd byte: P2 (Second byte of data packet number of last data packet received)

- 4th byte: Quotient of {(1+P1+P2) div 256}

- 5th byte: Remainder of {(1+P1+P2) div 256}

Step3 : SWDR – Perform stepwise dead reckoning (SWDR) on the received data

MIMU22BTP transmits tracking data with respect to its own frame which itself is not fixed with respect to the user's global reference frame. Therefore the data should undergo rotational transformation before being presented to the end user in a global reference frame. (Here global refers to the coordinate axis in which the system is initialized.)

The first four bytes of Payload are the position and heading change vector, i.e. dx, dy, dz, da (da is the change in angle of rotation about z-axis). These values are the output from the device at every ZUPT instance (ZUPT instance = Foot at complete standstill = Step detection). As mentioned earlier, each set of delta values describe movement between two successive steps. The delta values prior and after each ZUPT instance, are independent to each other as the system is reset every ZUPT, i.e. the delta values in one data packet is independent of data value in data packet transmitted just before, just after and all other. The position and heading change vector are with reference to the coordinate frame of last ZUPT instance. The 'da' corresponds to the deviation in the orientation of MIMU22BTP. It is rotation of the x-y plane about the z-axis, and z-axis is always aligned with the gravity.

The da is thus used for updating the orientation of the system, and obtaining the global reference frame. The two dimensions of displacement vector (dx, dy) are transformed to the global frame by applying rotation and thus (x,y) in the global frame is obtained. As we do not consider any change in alignment in z-axis, updating the z coordinate is performed by simply adding present dz to the z obtained for previous ZUPT.

Thus x,y,z coordinates in the global reference frame are obtained at every ZUPT.

Step 4: STOP-(i) Stop processing in Osmium (ii) Stop all outputs in Osmium

(i) Command to stop processing in Osmium by sending 3 bytes command: {0x32, 0x00, 0x32}

(ii) Command to stop all outputs in Osmium by sending 3 bytes command : {0x22, 0x00, 0x22}

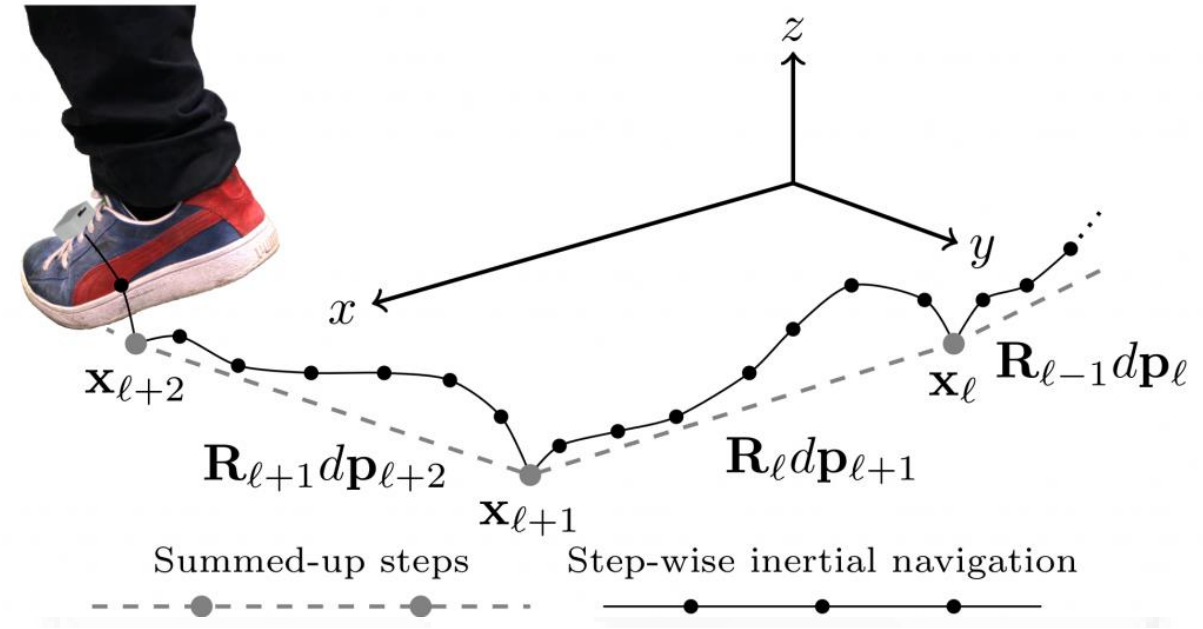


Figure 1

Chapter 5
POSITION PLOTTING

The X, Y, Z data received from the sensor module is displayed on the app interface after being processed.

Other attributes such as step count, distance, speed and a master clock displaying the duration of the tracking was also added to make the information gathered more detailed for various purposes.

We decided to make it possible to plot this data graphically. We started off the approach by using a graphView class using which we managed to create a static 2D plot. Instances of the initial code output is shown in the figures below.

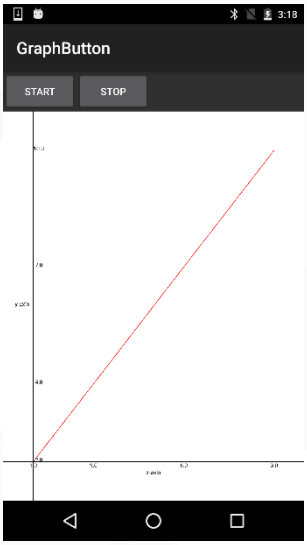


Figure 2 :2D static plotting

The main issue we faced with this was that the data wasn't being plotted in real time, though the plotting was accurate, it couldn't be used to meet our needs. This graph required all the data points to be pre-defined.

To solve this we improved on our original approach, we created a plotDynamic class that could plot one point at a time on a canvas. Dynamic plotting required the graph to refresh after every entry of the user. We used the function 'invalidate ()' which forced replotting of the canvas. So every time a new point was obtained we called 'invalidate ()' so that the new point would be plotted on the same canvas. To debug we linked this to the click of a button, which gave us a feed of random values, to simulate the incoming serial data. What we found was that every time we clicked the start button the point moved to latest X, Y values. This class was plotting the points dynamically.

We also had to find the centre of the canvas because we needed this to be our plot's origin. Conveniently we found two functions in the Canvas class getHeight and getWidth which gave the height and width of the canvas in terms of pixels respectively. Dividing these values by 2 we get the co-ordinates of our origin.

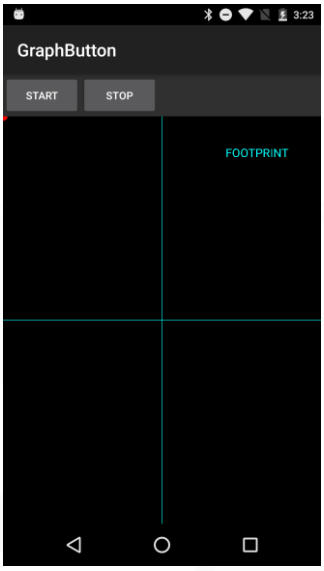


Figure 3: Dynamic Plotting without axes plotting

We were quick to draw the axes lines passing through this point. When we ran the app we found that the point was nowhere to be seen on the screen.

What we didn't realise was that because we had shifted our axes to the centre of the screen we had to correspondingly shift the points we wanted to plot on the screen.

We rectified this error by adding half the canvas width to the abscissa and subtracting half the canvas height from the ordinate, thereby giving us the relative position of the point with respect to our defined axis. We then got the desired results. The correction to be made was obtained by trial and error method.

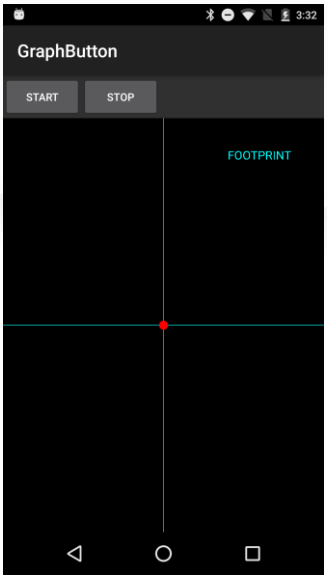


Figure 4: Dynamic plotting after axes correction

During the plotting the biggest challenge we predicted would be the implementation into the final app.

We had a touchEvent View in the application to allow us to input the points for the geo-fence by just drawing on the screen using his finger. While doing this we realised that it would be better to plot directly onto the touch view rather than to superimpose the touch view onto the graphview.

Doing this was very similar to the plotting on the GraphView. We just plotted onto a touch view rather than a graphView. The corrections that we had to make on the plot was also similar to what we had to in the touchView. But the main advantage of this was that we no longer had to worry about the axes from the graphView to have to coincide with the axis on geo-fence.

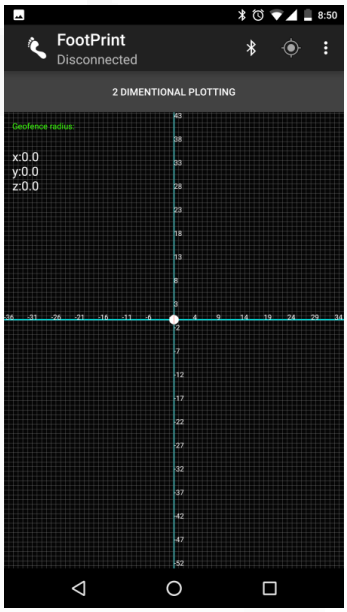


Figure 5 : implementation on to the App

While working with Canvas we found a Paint class which gave us control over the colour of the lines, point and text on the touchView. Using this we were able to change the colour of the geo-fence with respect to the position of the point to be checked. The geo-fence is green in colour as long as the person is located within, whereas it turns red when the wearer breaches the fence.

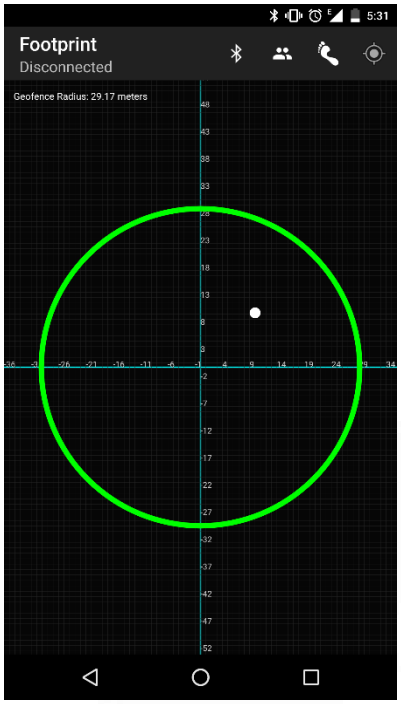


Figure 7 : Geo-Fence (inside)

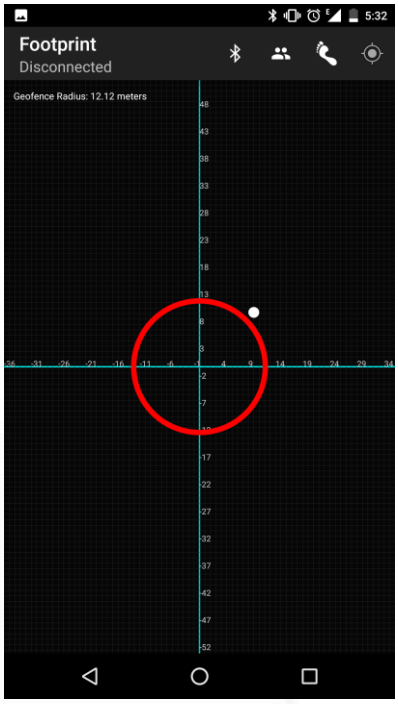


Figure 6 : Outside the Geo-fence

By calculating the relative position of the point with respect to the users' view, we were able to get a precise approximation of the users' location in 3D space. To get a 3 dimensional view of the point we rotated the Canvas plane such that all three axes were visible. By mathematically calculating the relative height from the ground in the users' view we were able to display the position in space.

In the future, we hope to also be able to show the wearers' trajectory continuously during tracking. We also hope to implement a polygonal rather than a circular boundary, along with being able to input a 3 dimensional geo-fence.

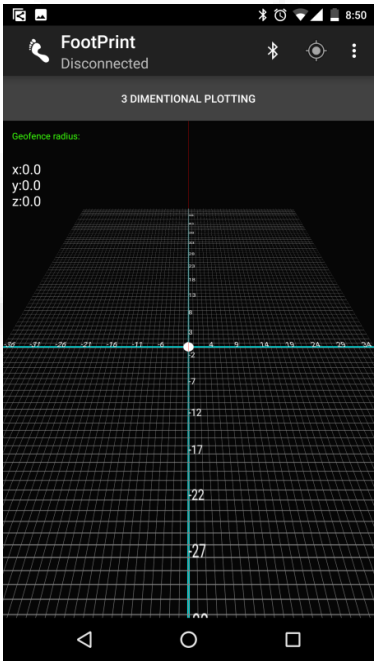


Figure 8 : 3D plotting

Chapter 6
RAY CASTING

In order to implement geo-fencing in our application, we considered to use the popular “Ray casting” algorithm.

About the Ray Casting Algorithm

In computational geometry, the point-in-polygon (PIP) problem asks to determine whether a given point in the plane lies inside, outside or on the boundary of a polygon.

A simple way to determine whether a point lies inside or outside a simple polygon is to check how many times a ray, starting from the point and going on in a particular fixed direction, intersects the edges of the polygon. If the point lies on the outside of the polygon, the ray will intersect the edges of the polygon an even number of times. If the point lies on the inside of the polygon, then the ray intersects the edge an odd number of times.

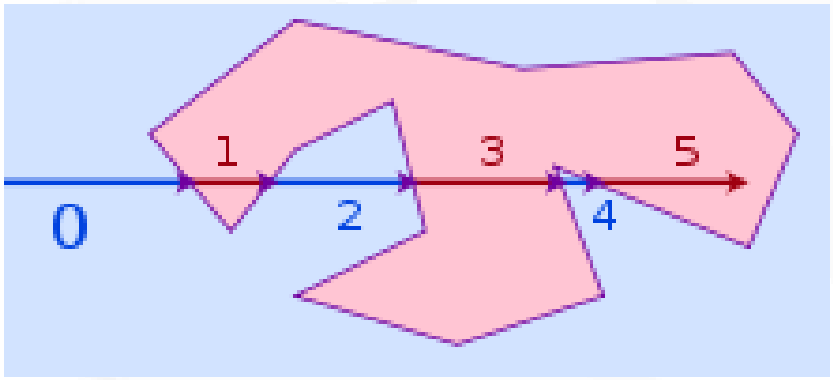


Figure 9

https://upload.wikimedia.org/wikipedia/commons/thumb/c/c9/RecursiveEvenPolygon.svg/220px-RecursiveEvenPolygon.svg.png

Coding the Ray Casting Algorithm in Java

We developed the Java code for the Ray casting algorithm on the “Eclipse Mars” IDE and used a JDK of version 1.7.

We used a modular approach to develop the java code.

1) Creating classes for the basic components – Point, Line & Polygon
   A ‘point’ is a position with an ‘x’ coordinate & a ‘y’ coordinate. The x and y coordinates are considered to be of type ‘double’.

```
public class Point {
    private double x;
    private double y;

    public Point(double x, double y)
    {
        this.x = x;
        this.y = y;
    }
    public double getX()
    {
        return x;
    }
    public double getY()
    {
        return y;
    }

};
```

A ‘line’ is a straight line with direction. It has a ‘from’ point and a ‘to’ point and is used to represent the edges of the polygon. Here, ‘from’ and ‘to’ are taken as objects of class ‘Point’.

```
public class Line {
    private Point from;
    private Point to;

    public Line(Point from, Point to){
        this.from = from;
        this.to = to;
    }
    public Point getFrom(){
        return from;
    }
    public Point getTo(){
        return to;
    }

}
```

A ‘polygon’ is a multi-sided shape and hence, is represented as an array of objects of class ‘Line’ in our code.

```
public class Polygon {
private Point[] points;
public Polygon(Point[] points){
    this.points = points;
}
public Point[] getPoints(){
    return points;
}
}
```

2) **Creating a method to check if the required point lies inside or outside the defined polygon.**

This method will perform the following steps:

i) Calculate and define the lines of the polygon
ii) Filter out those edges of the polygon (or lines) which intersect with the 'y' coordinate of our required point.
iii) Calculate the exact points on which the lines intersect with the 'y' coordinate of our required point.
iv) Sort the points by position using the 'x' coordinate.
v) Using the ray-casting algorithm for checking if the required point lies inside or outside the defined boundary of the polygon.

3) **Calculating and defining the lines of a polygon.**

The polygon is implemented as a linked list of lines.

The polygon is closed by connecting the last point to the first point of the array of 'Point' objects in a 'Polygon'.

4) **Filtering the edges of the polygon (i.e. lines) that intersect with the y-axis.**

5) **Calculating the x-intersection points of the respective y-intersection points.**

We calculate the intersection points by calculating the 'x' coordinates of the respective 'y' coordinate using the simple mathematical formula "y = mx + C".

6) **Sorting the points using the 'x' coordinate**

The calculated intersection points are sorted by their respective 'x' coordinate using "Collections.Sort" and the "Comparator" interface.

7) **Checking whether required point lies inside or outside the boundary of the polygon.**

We check the position of the required point using the ray-casting algorithm. Initially we declare the position of the required point (a variable of type Boolean) to be outside the polygon and at each point of intersection, we invert the status of the position till we reach the 'x' coordinate of the required point.

Getting Input for the Java Program

There are two sets of input that we need for the geo-fencing to be implemented on Java:

1) **Input of coordinates for the user-defined geo-fence**

   The input of coordinates for the user-defined geo-fence (i.e. the boundary of the required polygon) is taken graphically from the user on the Android app interface. The graphical input is converted into a text file containing the respective coordinates. The java code then, reads these as points for the required polygon from the text file.

2) **Input of coordinates of the point whose position (w.r.t. the polygon boundary) we want to find**

   The input of coordinates of the point whose position (w.r.t. the polygon boundary) we want to find, is taken in the Java program as the user keeps pressing 'y' when asked if he/she would want to continue giving input. In the app however, the code receives this input from the MIMU22BT device via the Android application.

- **Limitations**

1) The input received by the Android application from the MIMU22BT device was too huge and heavy for the Java code implementing Ray-casting to handle.
2) Problems in locating the text file storing the coordinates were also encountered.

Chapter 7
INDIVIDUAL CONTRIBUTIONS

Developing the app was a combined effort by all the team members under the continuous guidance of **Prof. K V S Hari**, Head of ECE Dept., IISc Bangalore. Everyone made contributions to all aspects of the app. Below given is the list of team members along with their specific contributions to the app:

1. Saahil Kamath [2nd year, ECE branch, PES University, Bangalore] :

- Designed the complete GUI[Graphical User Interface] of the application. Giving it lot of features, as well as making it as user friendly as possible.
- Contributed to a major portion of the back end coding of the app, including Bluetooth communication, deciphering received data from the device and displaying coordinates.
- Helped design the graph, plotting data points dynamically as well as show trajectory in 3D space. Also helped implement Ray Casting algorithm in the app, along with making the geo fence.
- Completed the app, joining all the various components, such as the main activity, graph and the algorithm on Android Studio.

2. Rishank S Nair [2nd year, ECE branch, VIT University, Vellore]:

- Deciphered the DaRe source code in order to obtain a detailed understanding of the working of the device. It also helped understand how Bluetooth communication occurs between the devices and how the app is supposed to read and process data received from the device.
- Helped in the back end coding of the app, covering everything from Bluetooth communication to establishing plotting in a graph view.
- Helped with the testing and debugging of the 2D static and dynamic plotting code so as to make it compatible with the requirements of the app.

3. Manav Gurumoorthy [1st year, ECE branch, NIE University Mysore] :

- Developed the code for 2D static as well as dynamic plotting of data. Modified it to suit the needs of the FOOTPRINT application.
- Understood the working of canvas and was able to refresh the graph view each time a new data point was input.
- Helped with the final plotting of data in the application, as well as in the conversion of 2D to 3D plane.

4. Aishwarya Ray [1st year, CSE branch, Purdue University, West Lafayette, USA]:

- Developed the Ray Casting algorithm, used to check whether a given point is located within a user specified boundary. Modified it to suit the needs of the FOOTPRINT application.
- Helped decipher the DaRe application source code to understand working of the MIMU22BT, along with the processing of data received from it.
- Understood and helped with the plotting of data on the graph view.

The process of developing the app had a lot of positives to take away. It increased our knowledge in Android as well as Java programming. It also gave us an in depth understanding of the working of Bluetooth and communication over it. The project helped us learn how to systematically approach a problem and also how to implement our theoretical knowledge into a practical application.

Chapter 8
The “FootPrint” Application

CONTACT US AT:
IIScFootprint@gmail.com

About the App

The FOOTPRINT app is a pedestrian dead reckoning and navigation system, developed for the android platform
to work in tandem with the OSMIUM MIMU22BT, from Inertial Elements.
The app displays the person's X, Y, Z coordinates under the *Footprint Coordinates* section in the home page.
Footprint Details section displays the number of steps taken, distance, total time and average speed of the
person during the duration of the tracking.
The tracker icon on the top right of the display takes us to the graphical display of the person's motion
and the geo-fence.

Screenshots of the app

Figure 10: FootPrint Homescreen

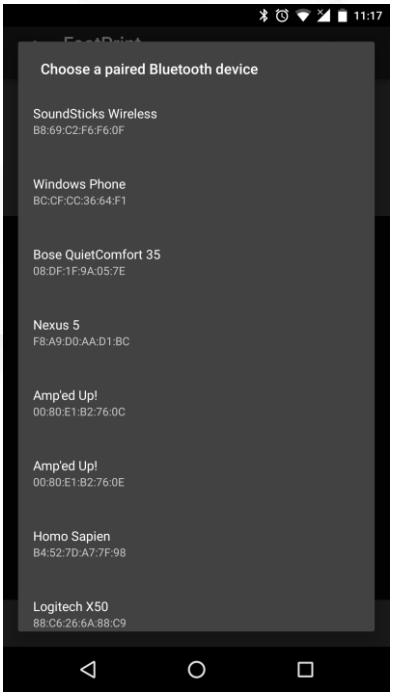


Figure 11: Bluetooth Select

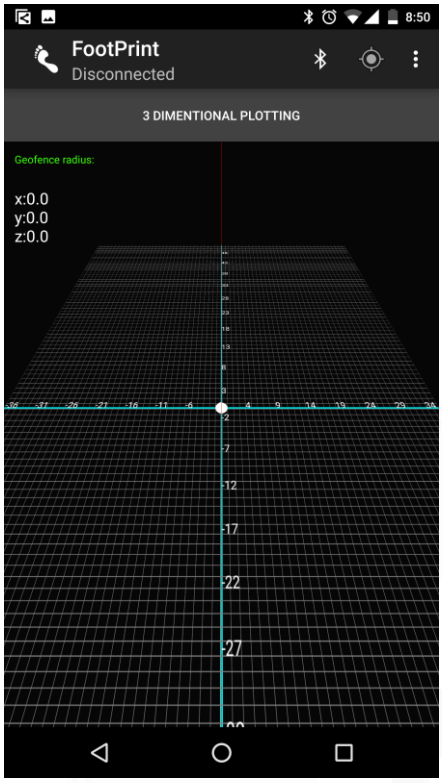


Figure 12: 3D plotting

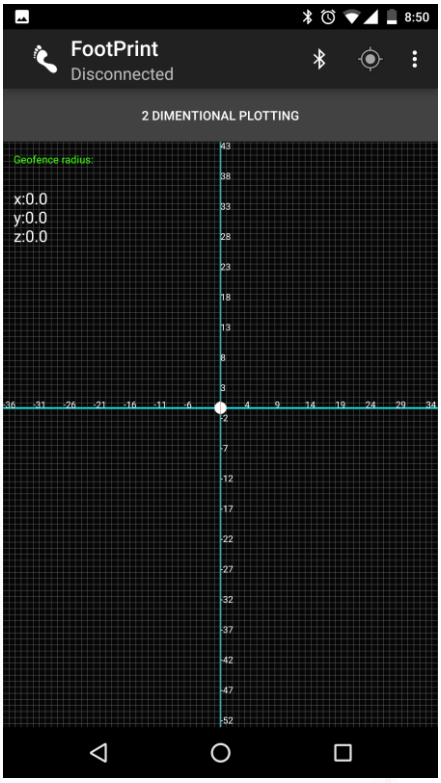


Figure 13: 2D plotting

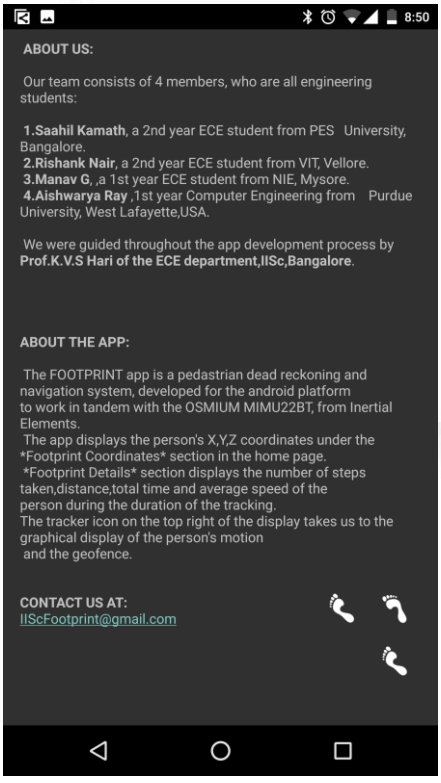


Figure 14: About Us

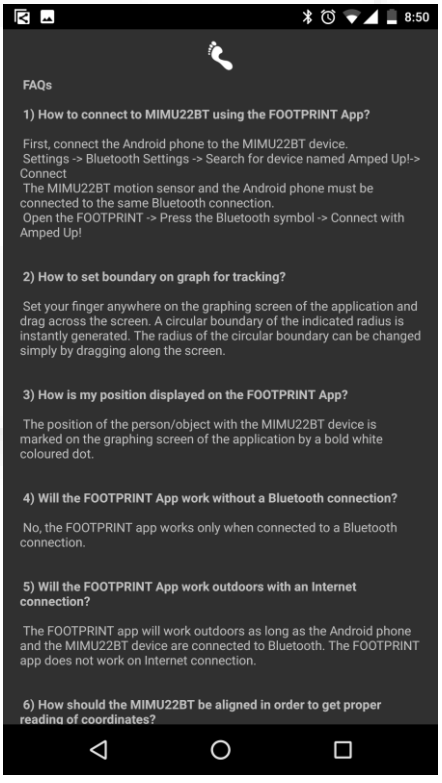


Figure 10: FAQs and Contact Us

Chapter 9
FAQs

1) How to connect to MIMU22BT using the FOOTPRINT App?

First, connect the Android phone to the MIMU22BT device. Settings -> Bluetooth Settings -> Search for device named Amped Up!-> Connect The MIMU22BT motion sensor and the Android phone must be connected to the same Bluetooth connection.
Open the FOOTPRINT -> Press the Bluetooth symbol -> Connect with Amped Up!

2) How to set boundary on graph for tracking?

Set your finger anywhere on the graphing screen of the application and drag across the screen. A circular boundary of the indicated radius is instantly generated. The radius of the circular boundary can be changed simply by dragging along the screen.

3) How is my position displayed on the FOOTPRINT App?

The position of the person/object with the MIMU22BT device is marked on the graphing screen of the application by a bold white colored dot.

4) Will the FOOTPRINT App work without a Bluetooth connection?

No, the FOOTPRINT app works only when connected to a Bluetooth connection.

5) Will the FOOTPRINT App work outdoors with an Internet connection?

The FOOTPRINT app will work outdoors as long as the Android phone and the MIMU22BT device are connected to Bluetooth. The FOOTPRINT app does not work on Internet connection.

6) How should the MIMU22BT be aligned in order to get proper reading of coordinates?

The direction reference axes are marked on the side of the MIMU22BT device. When using, the device is aligned such that positive X-axis on the app is taken as the direction in which the side marked as the positive X-axis on the device.

7) What should be done when the app stops responding to the device?

When the app stops responding to the device, check for the following:
i) the phone and the device are within the range of the Bluetooth connection.
ii) The device is charged.

Chapter 10

CONCLUSION

Footprint: An elegant and convenient solution for positioning, tracking and fencing.

To summarize, we have developed an application that tracks and fences an Osmium MIMU22BT module. The app shows the trajectory of the device to the user and also allows him to draw a geo fence around it. When the device has crossed the defined boundary the user is alerted using vibration or notification on the phone. The app also stores a text file of the data from the tracking session in a folder in the memory of the system for further use.

The applications of this project in real life are infinite and the app can be further developed to suit each one of them. Since the device currently communicates using Bluetooth, it has a limited range up to approximately 25mtrs which is inconvenient for most real world applications. This is something that can be looked into and improved upon.

It was a wonderful and learning experience for all of us while working on this project. The project took us through various phases of project development and gave us a real insight into the world of software engineering and android app development. The joy of working and the thrill involved in tackling the problems and challenges gave us a feel of developers industry.

It was due to this project that we came to know how professional applications are designed. We learnt how to breakdown problems and systematically solve them.

Chapter 11

FUTURE WORKS

The scope for future development of our project is immense. Some of the things we had in mind but were unable to implement are:

1. To implement the geo-fencing for fence that are polygonal rather than just circular.
2. To plot the trajectory of motion along with the position of the wearer.
3. To make a truly 3D plot in contrast to the current virtual 3D plotting.
4. To implement the layout of a building onto the plotting surface.
5. To increase the range of the transmitter we would like to explore other technologies.

Chapter 12

BIBLIOGRAPHY

- ***http://www.inertialelements.com/support.html***
- ***https://developer.android.com/index.html***
- ***http://www.openshoe.org/***