University of Dublin

TRINITY COLLEGE

Mountain View

Hugh Concannon
B.A.(Mod.) Computer Science
Final Year Project May 2018
Supervisor: Dr. Jeremy Jones

School of Computer Science and Statistics

O’Reilly Institute, Trinity College, Dublin 2, Ireland

Abstract

This project was originally titled “Web-based Phone App That Augments a Camera Image with the Names of Mountain Tops”, which I think summarises it fairly well.

There is a very distinct lack of general augmented reality web apps on the market, and an even more distinct lack of A.R. web apps that place landmarks onscreen. Generally when one wants to find out what landmark it is they’re looking at, they’ll go to Google Maps and find out. Which is all well and good, but since Google Maps always aligns the map such that the top of the screen is north, a sense of direction is required that some people lack. And when you’re walking through the mountains and they all look the same, Google Maps isn’t going to help you orientate yourself very much.

That’s where this project comes in. In this report I will outline the steps I took to create a web app which places the names of the mountaintops on top of a camera feed, with minimum interaction required from the user.

Contents

Reader’s Guide

Chapter 1: Introduction

Lays out the background, the aim of the project and motivation of the developer.

Chapter 2: Background and State of the Art

Explains key concepts behind the technology the project uses, the reasoning behind the developer’s choices, and examines other applications that fulfill a similar role.

Chapter 3: System Design and Planning

Explains the process taken by the developer to plan out how he would develop the project, and provides an in-depth look at the algorithm used by the application.

Chapter 4: Implementation

Provides a lower-level look at how the program is implemented, listing the technologies and programming used.

Chapter 5: Evaluation and Conclusion

Lays out the successes, aspects that could be improved, methods of testing, and a reflection on what the developer has learned throughout the project.

1 - Introduction:

Motivation

Ireland has many mountain ranges, as well as free-standing mountains and hills, and mountaineering is a common hobby among Irish people. Up to 67,600 people climb Carrauntoohil, Ireland's tallest mountain, per year[1]. I myself have been an avid mountain climber since I was small, as both my parents are keen mountaineers themselves. Thus, when my supervisor suggested this idea to me, I was more than eager to start on it.

Aim:

The purpose of this project was to create a webapp that would aid mountaineers during their journeys, highlighting which mountains are which so that the user can get their bearings, or just learn some local geography, with minimum effort, and to be easily extensible to other platforms such as a car's Head-Up Display – for a user driving through a mountain range, for example. This program requires no download, and is not OS-specific. All it requires is a reasonably up-to-date web browser and a camera.

2 - Background and State of the Art:

My project is a *web application.*

Web Application Definition:

A web application or web app is a client-server computer program which the client runs in a web browser. In a web application, the majority of computation is performed on the server side, as opposed to a *native app*, which runs directly on the mobile device. Web applications are generally written in a standard format, such as HTML and Javascript, whereas native apps are written in Java (for Android devices) or Objective-C or Swift (for iOS devices).

Why use a web application?

Web applications have many advantages over native applications, chief among them being:

- Will run in both Android and iOS (and, although this isn't a primary aim of the project, Windows, Linux and MacOS), requiring only a web browser and built-in camera.
- The majority of computation is performed server-side, so RAM and CPU usage is reduced.
- It does not require any installation, only internet access, to work for the first time on the device.

Architecture of a web application:

Most web applications follow a three-layer model. The three layers are called the Presentation Layer, the Application Layer and the Storage or Data Layer.

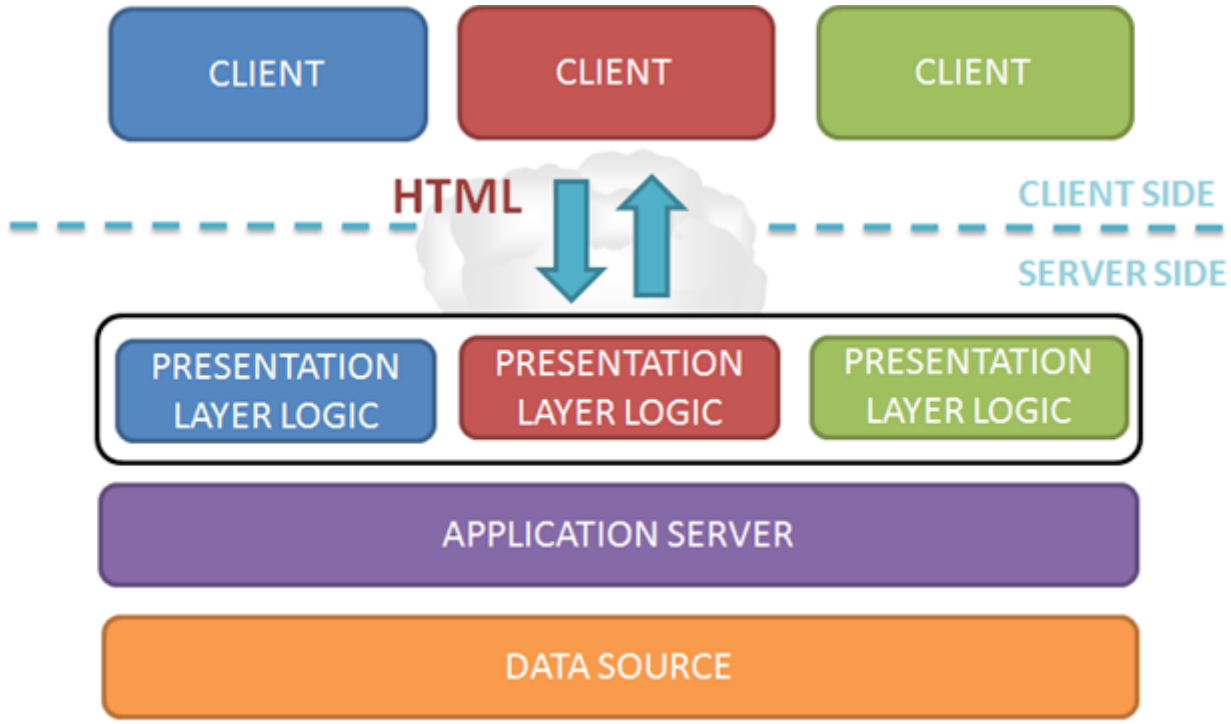


Figure 2.0 – ***An abstraction of most web applications***

Generally, the presentation layer constitutes the User Interface/front-end of the application, the application layer takes care of the functionality of the application, and the third layer is the database.

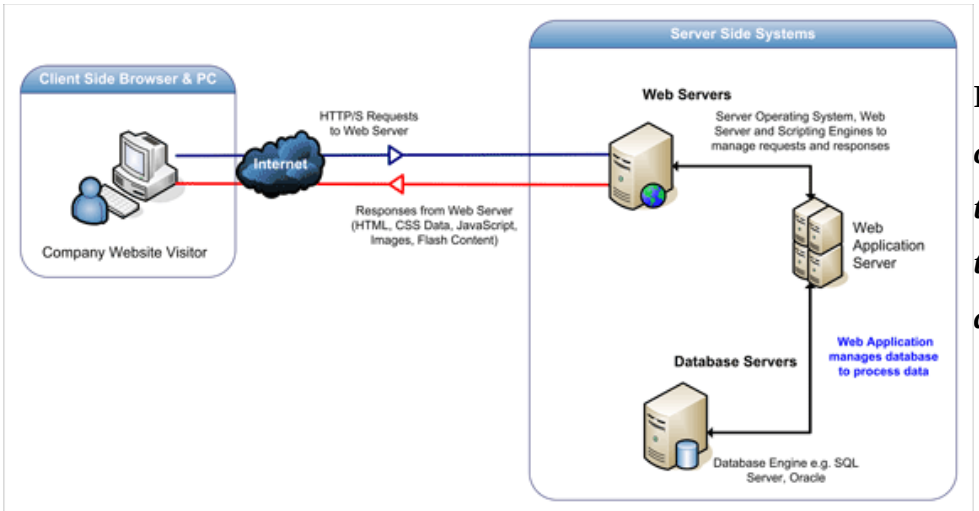


Figure 2.1 – ***A flow chart demonstrating the operation of a three-tier web application.***

Existing alternatives:

As I mentioned before, many landmark-finder applications exist already, almost all of which are native applications. In fact, I was unable to find any augmented reality web applications using Google or Bing (as of 23/4/18). Every app that augments reality, that I have been able to find, is either Android or iOS specific.

Nevertheless, here are one or two alternatives that, while being native apps, do what my web app does:

1. Wikitude[2]

Price: Free

Platforms: Android 1.5+, iOS 4.0+

Figure 2.2 – ***A screenshot of the Wikitude Android app.***

2. Layar[3]

Price: Free

Platforms: Android 1.5+, iOS 3.1+

Figure 2.3 – ***Stock image of Layar running on an Android phone.***

While these apps are free (and slightly more polished than mine), both of them require download and installation.

The Application Programming Interface:

Google Maps API

Google offers a vast array of services that can aid a programmer in developing anything pertaining to maps, geometry, route planning, geodesy (mapping coordinates onto Earth), altitudes, and landmarks.

This is the API my project used. The advantages of the Google Maps API are manifold, including:

- Free (for all intents and purposes – one had to pay to send more requests after reaching a certain limit, which was 2,500 per day. I only broke this limit once, when I accidentally included an infinite loop in my code).
- Easy to use: Once I got the hang of Javascript and including libraries, adding more Google Maps libraries was a matter of typing in a keyword in the <src> tag.
- Accurate: Google provides relatively high resolution elevations, accurate to 20-30 metres – more than sufficient for my project.

Setting up the libraries was a bit of a challenge, as I found Google's developer website unintuitive and hard to navigate. If you were to look up, for example, how to add the Elevations library to your code, you would be told all the information about the parameters the request took, the format the results were returned in, and lots of other potentially useful information... except how to include the library. Many times, I just looked up code examples to get the libraries to work, avoiding sites owned by Google because they would be consistently unhelpful. I noticed that Google pushes any pages about Google Development, that are not owned by Google, to the bottom of the results page, which was rather inconvenient. It seems Google only wants Google-sanctioned programs to use their code.

My API Key: AizaSyBx11PAjXcTybOGZKLAOBLCamxQd0pQAI8

Other APIs, Web Services and Libraries Used

Google Maps Javascript API

This is the API that allows you to include a map on your webpage. I required a map to visualise my algorithms at work (to be elaborated on in the Algorithms section). It required only a centre point (as latitude and longitude coordinates) and optionally, a zoom level, and it would the display a map on the webpage.

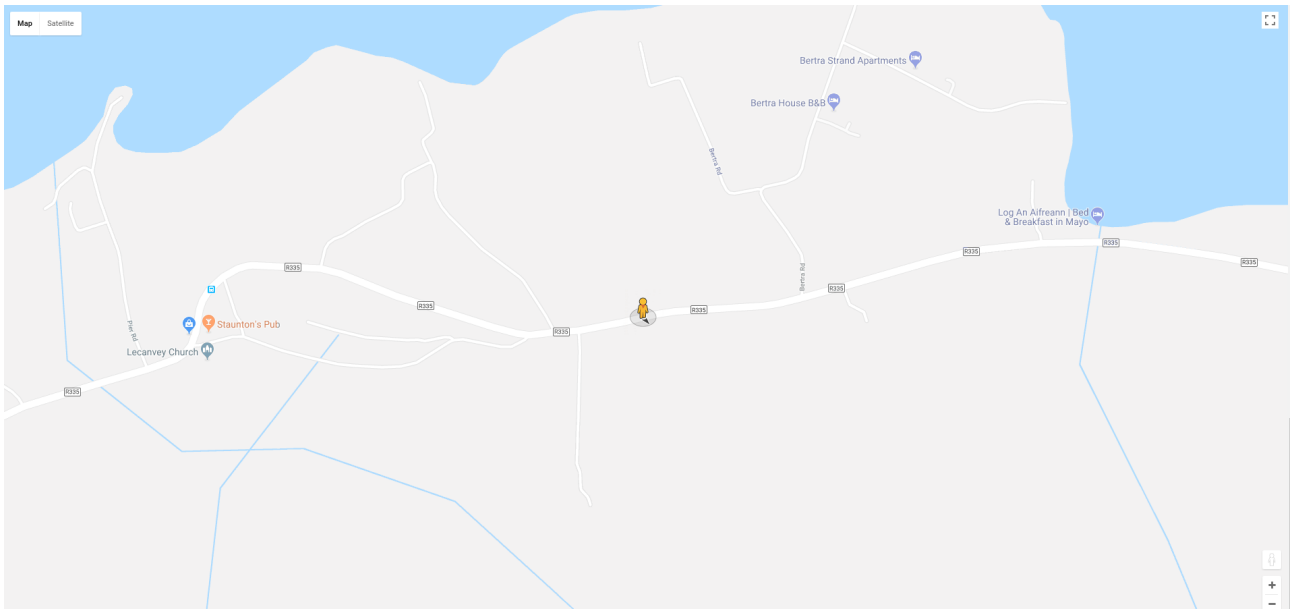


Figure 2.4 – ***An example of a Map element.***

Google Places API Web Service

This API allows you to search for nearby locations, based on a centre point, a search radius, and location category (e.g. restaurant, place of worship, hairdresser).

Figure 2.5 – ***An example of a data result provided by the Places API, in this case, Croagh Patrick. As you can see it provides a function for getting the latitude and longitude (in the lat() and lng() functions) as well as data such as the name of the landmark, and its "rating".***

Google Maps Elevation API

This API provides elevation data for all locations on the surface of the earth in metres above sea level, including locations under the sea (returned as negative values). In places for which Google doesn't have precise elevation data, it returns the average value of the four nearest locations it does have data for. It accepts requests in the form of a *path* (the co-ordinates of the start point and the endpoint) and the *number of samples to be provided* along said path.

```
elevator.getElevationAlongPath({'path':paths[i], 'samples':35}, (elevations, status) =>{
```

Figure 2.6 – ***The function which takes in a path and a number of samples, and passes the elevation data for said path in an array called "elevations", and a status, into a callback function.***

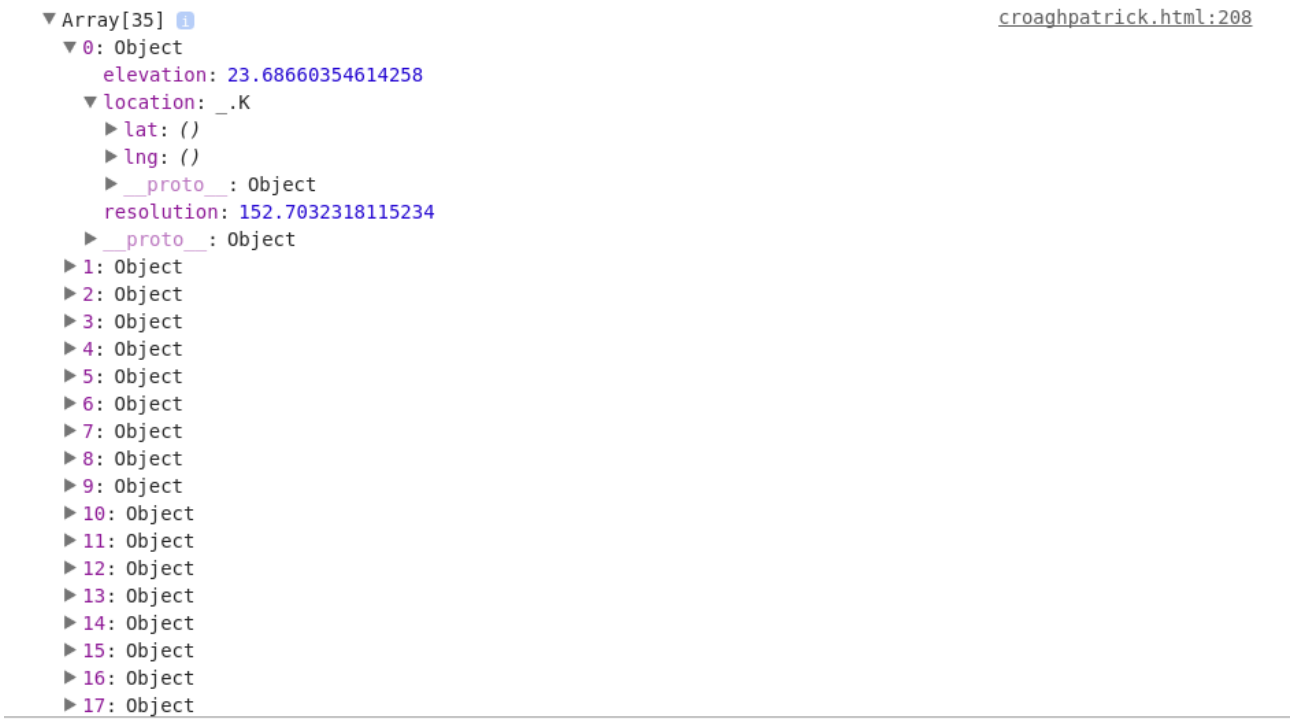


Figure 2.7 – An example of a data result provided by the Elevation API. It consists of an array of 35 samples of JSON objects – that is, 35 latitudes and longitudes and their elevations. The significance of the number 35 will be explained in the Algorithms section.

Google Street View Image API

This allows you to include a Google Street View Image on your website. To elaborate, you give a location, a bearing (in degrees of north), a pitch (whether the view should be looking up, or down, or straight ahead), and optionally a zoom level, and you get a Google Street View according to these parameters. I used this only for testing purposes, because obviously, going out to the Dublin mountains every time I needed to test my program wasn't feasible...

Figure 2.8 – An example of a Street View Image element on a webpage

Google Charts API

This API is, as you may expect, for graphing and charts. As with the Street View Image API, I only used this for debugging, to make sure my calculations were correct. For example, to test if the elevations provided by Google were correct, I fed into the Elevation API a path starting in front of a mountain and ending behind it, and fed those elevation results into the Charts API, and got a line chart which started low, peaked in the middle, and ended low.

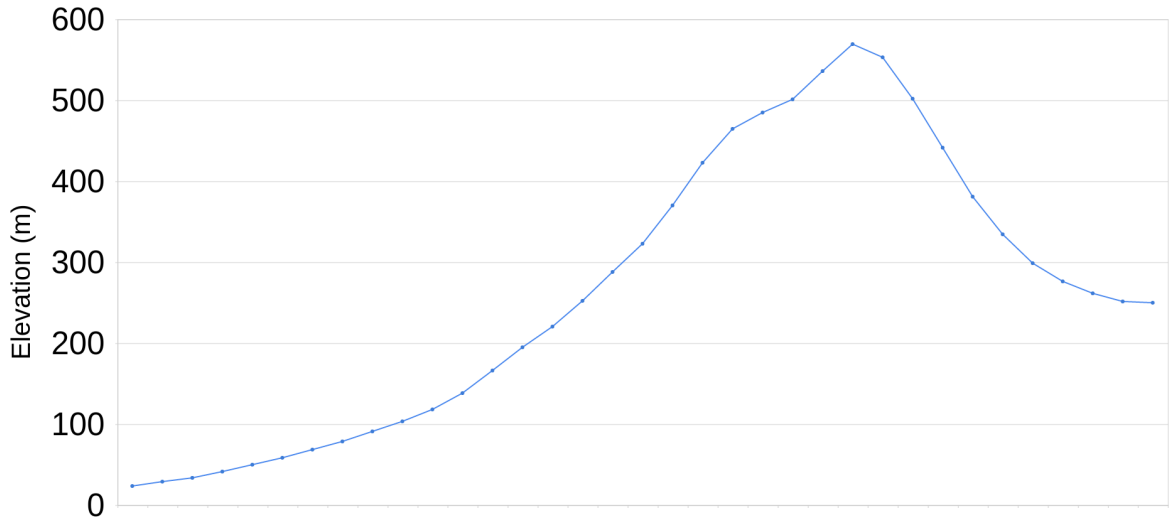


Figure 2.9 – A graph of the elevation data as described above, Croagh Patrick being the mountain in question.

Alternative APIs:

Before beginning this project, I had only really heard of Google Maps, but after researching I found some others too, including the Bing Maps API, and Leaflet, a free and open-source mapping API written in Javascript. Since Google Maps was the one I was most familiar with, and the most popular, thereby having more support on Stack Overflow and the likes, that's what I opted for.

Learning Resources:

Prior to this project, I had next to no knowledge of Javascript, and had to rely heavily on Stack Overflow for the first few weeks, mainly to get help with the small, generic errors I was getting, that you can get developing any JS program – for example, Javascript's object system, JSON, took me a while to get the hang of, before I realised how analogous it is to Java's object system. Stack Overflow was also invaluable for learning about callback functions, a concept my project relies very heavily on.

For problems relating to mathematics, Stack Overflow's maths section came in handy, as well as http://janmatuschek.de, which contained loads of valuable information relating to geodesy (that is, the mathematics concerning coordinates on a sphere). The majority of functions I used are from this website (albeit translated from pseudocode to Javascript).

For HTML and CSS, I knew enough at first to scrape together a decent-looking webpage with the bare minimum functionality required for testing. All I needed to know was how to layout the webpage and how to include scripts. As I progressed to including a live video of the camera feed, https://www.html5rocks.com/en/tutorials/video/basic provided eveything I needed to know. For drawing the names of the mountains onscreen, I used https://codepen.io/icutpeople/pen/whueK, which provides information on drawing text over a <video> element.

Lastly, most of the information I required pertaining to Google Maps was on their website, including thorough information on the functions they provided, the specific inputs the functions accepted, the formats of their outputs, how to layout the elements, and more.

3 - System Design and Planning:

Requirements:

Functional Requirements:

Accessing Location and Orientation:

Firstly, the web app needs the user's location and orientation to provide data on which mountains are in view of the user.

Accessing the Camera:

The web app needs to provide a camera feed from the user's device so that they know which mountain is which.

Draw Names of Mountains Onscreen:

The web app then shows the name of each mountain on top of each mountain onscreen.

Non-functional Requirements:

Reliability:

The user should be able to access the web app from any location on Earth, and the web app should always be online. This may be a problem if the mountain is far from an urban area, but mountains near to urban areas are likely to have a telecommunications mast nearby.

Performance:

The app should load and display the mountain names within a reasonable amount of time. The goal is ideally for the names to reload in an imperceptable amount of time, so that as the user rotates their phone, the names onscreen are adjusted instantly.

Usability:

The app should be easy to use, with minimum interaction from the user, and should look good.

UML Diagrams:

Activity Diagram:

Here is the algorithm of the program broken down into basic steps:

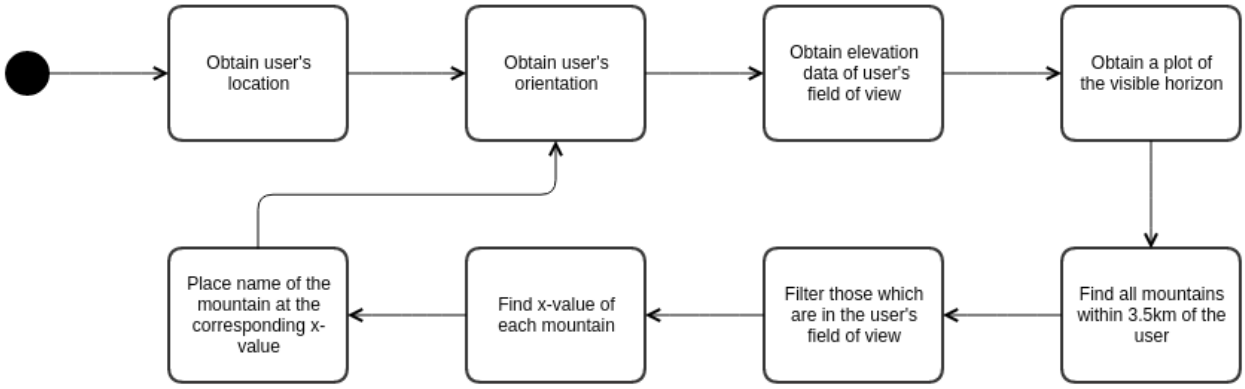
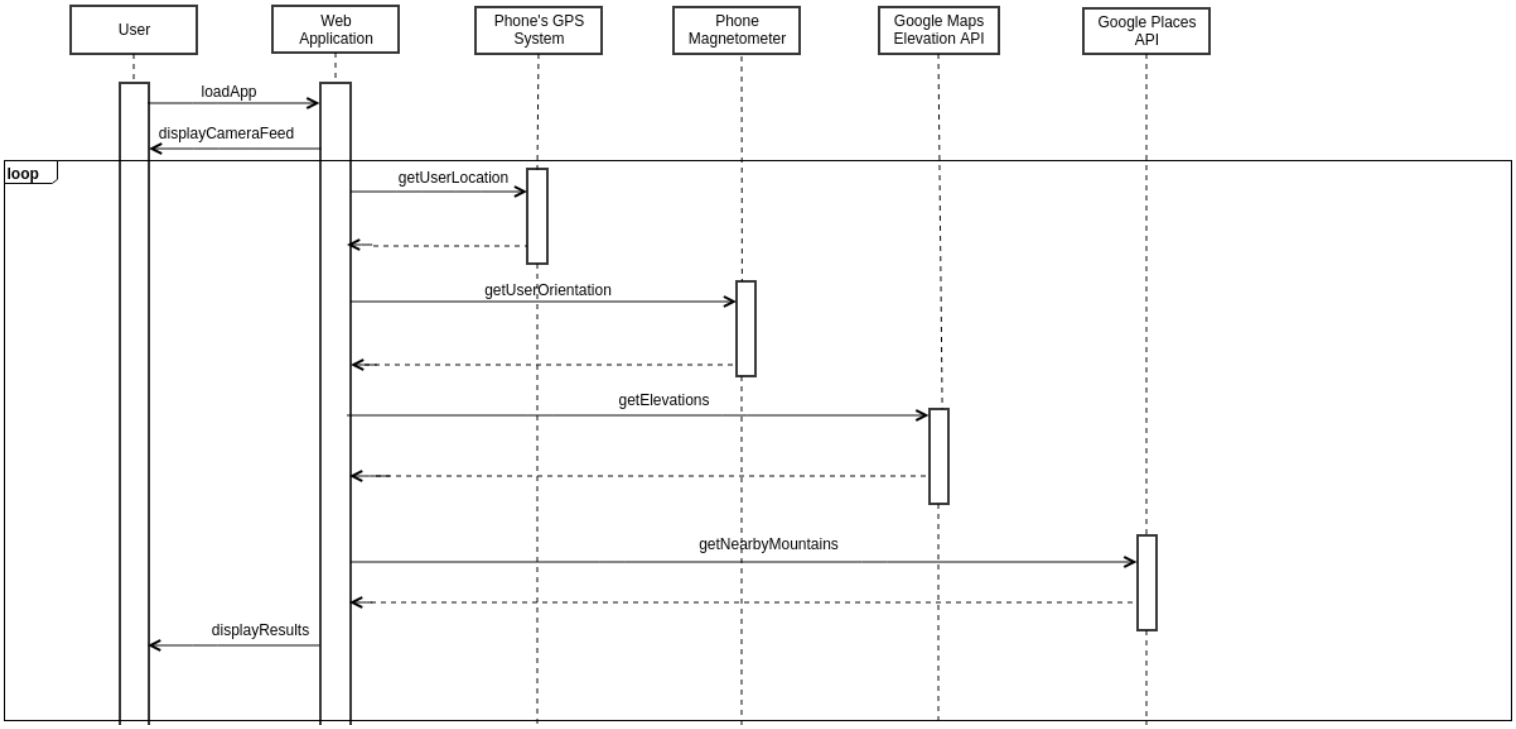


Figure 3.0 – ***A UML activity diagram of the application***

As you can see, the web page is always being updated with the user's orientation, rather than working once and waiting for the user to do something, like press a button. Every time the orientation is updated, the elevations of their field of view are requested again, the visible horizon is plotted, a search is performed for all the mountains with 3.5km of the user (3.5km is the distance to the horizon on a perfectly smooth Earth-sized sphere), those which are within view of the user are filtered, and their x-coordinate is determined (whether they appear on the left or right of the screen), and then their name is displayed where appropriate.

Sequence Diagram:

Figure 3.1 – ***A UML sequence diagram of the application***



Here you can see the different systems being used in the web application, the role they play, and the order they operate in.

User Interface:

Having a simple, easy-to-use user interface is essential for keeping the user's attention, and ensuring they come back for more.

My app consists only of a camera feed. The only interaction the user does with it is rotating their phone to point at a mountain. Thus there is no need for any buttons or text, aside from the mountain names.

Figure 3.2 – ***A mockup of the web app's userface.***

The Algorithm:

The process of placing the name of a mountain above said mountain on the screen of the device consists of the following steps:

1. Obtain the user’s location.
2. Obtain the user’s orientation.
3. Plot the horizon as apparent to the user.
4. Find which mountains are visible to the user based on their location and orientation.
5. Find out where the mountains are in relation to the user, and how high they are in relation to the user.
6. With this information, place the name of the mountain on the screen, above the mountain.

I will now go in-depth into each step.

1. Obtaining the user’s location:

All modern mobile phones have a built-in geolocation system. The geolocation system works by one or both of the following:

(i) GPS – Global Positioning System. GPS is a satellite-based radionavigation system owned by the United States Government and operated by the United States Airforce. It consists of several satellites carrying very stable atomic clocks that are synchronised with eachother. The receiver listens to the satellites, which are continuously broadcasting data regarding their current time and position, and determines its location based on differences between the satellites’ clocks.

(ii) Phone Signal Strength. Mobile phones are continuously communicating with nearby radio masts. A phone’s location can thus be calculated from the relative strengths of the nearest radio masts.

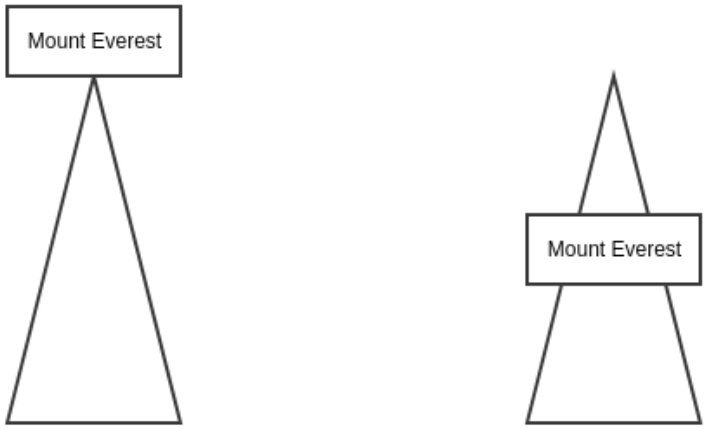
2. Obtaining the user's orientation:

All modern phones also have a built-in piece of hardware called a *magnetometer* – that is, a device for measuring the strength and direction of magnetic fields. The phone's orientation along its x, y and z axes can be found from finding out the strength and direction of the Earth's magnetic field. For this project, we only care about the phone's rotation around its y-axis, as this is its north-south bearing.

3. Plot the horizon as apparent to the user:

All this means is, getting the the visible horizon – the points in the user's field of view that they can't see behind. For example, if you are looking out to sea, the horizon is a straight line. If you are looking out to sea but there is an island 100 metres in front of you, the horizon will be a straight line that is raised in the middle.

The reason we want the visible horizon is to be able to put the names on the mountains *above* the mountains, as opposed to *on* the mountains. And to be able to do this, we need to know what the horizon looks like. See figure 3.3.



Good *Bad*

Figure 3.3

How to plot the visible horizon:

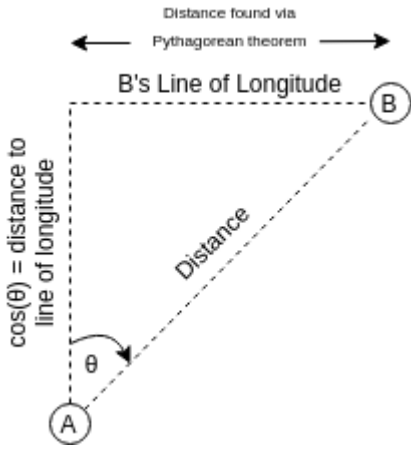
(i) We now know the user's orientation. We can assume that the phone camera has a field of view of approximately 90 degrees. Thus, from -45 to +45 degrees, in 3 degree sweeps, we draw paths 3.5km long, as this is the farthest the horizon can possibly be, resulting in 31 lines. To draw the paths, we need a start point (in this case, the user's location) and an end point. To calculate the end point, we use this formula:

```
//takes in coordinates, distance to next point, and the angle created by the line between them, with respect to north
//return coordinates of said point
function coordsGivenPointDistanceAndAngle(p, dist, angle){

    var distanceFromLineOfLongitude = Math.cos(angle*Math.PI/180) * dist;
    var latitudeOfB = distanceFromLineOfLongitude / ONE_DEGREE_OF_LATITUDE;
    var oneDegreeOfLongitude = (Math.PI / 180) * RADIUS_OF_EARTH * (Math.cos(latitudeOfB));
    var longitudeOfB = Math.sqrt((dist * dist) - (distanceFromLineOfLongitude*distanceFromLineOfLongitude))/ oneDegreeOfLongitude;
    if(angle>180) {longitudeOfB *= -1;}
    var b = {lat:latitudeOfB+p.lat, lng:longitudeOfB+p.lng};
    b.lat = precisionRound(b.lat,6);
    b.lng = precisionRound(b.lng,6);
    return b;
}
```

This function, as you can see in its comment, takes in a point, represented as a latitude and longitude, the distance to the point whose coordinates you want, in metres, and the angle that is created by the line passing through both points with respect to north, in degrees. In this case, the user's location, 3500, and the user's orientation + or - some number between 45.

Created on draw.io



It works by getting the distances as shown in this diagram, and converting those distances to differences in coordinates, then adding those differences to the input point's latitude and longitude. The magnitude of one degree of longitude in metres varies depending on one's distance from the equator. For example, if I am 1 metre south of the north pole, I just need to walk $\sqrt{2}$ metres and I'll be 1 metre east of the north pole, and will have increased my longitude by 90º. However, if I'm at the equator, increasing my longitude by 90º means travelling the distance between Ecuador and Congo. All of Javascript's trigonometric functions annoyingly deal only in radians, which took me an embarrassingly long time to realise.

Once the paths are calculated, we get a virtual fan coming out of the user's phone.

See figure 3.4 for a visualisation.

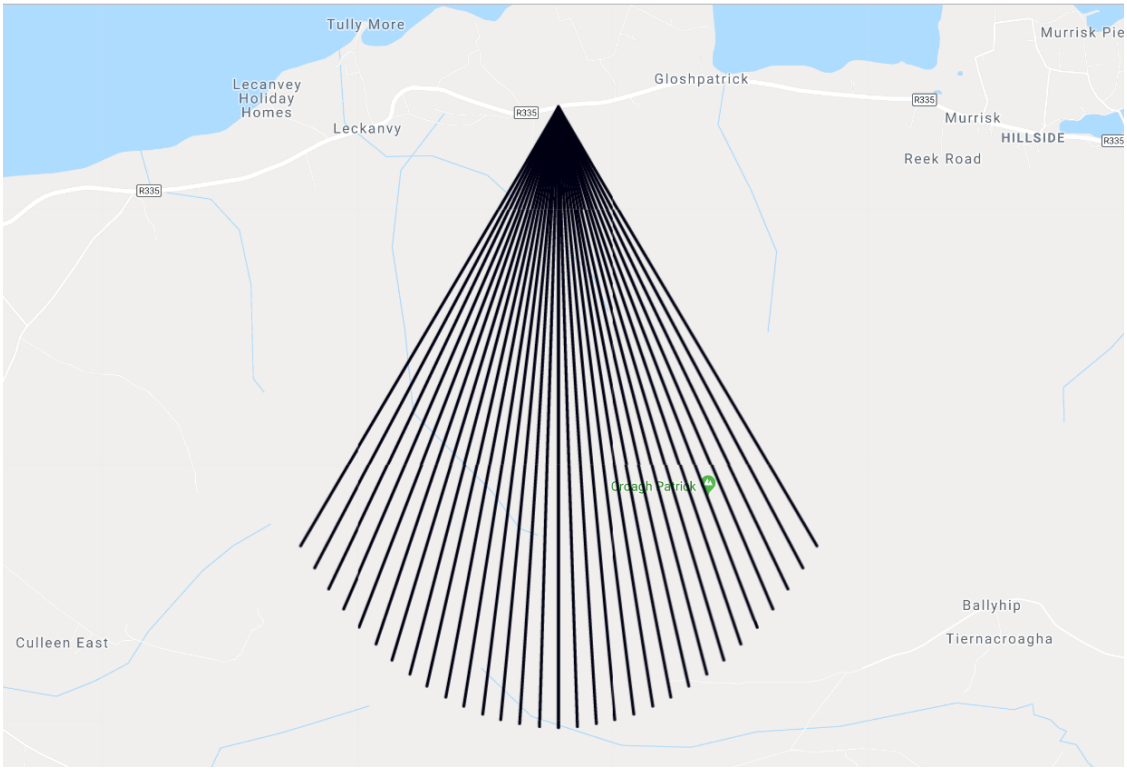


Figure 3.4 – ***A visualisation of a user, who is facing directly south, looking towards Croagh Patrick. A fan is drawn from their orientation minus 45 degrees, i.e. the left side of the screen, to their orientation + 45 degrees, i.e. the right side of the screen.***

(ii) Next, we request elevation data for each path. In this project I request 35 samples, that is, one sample for every hundred metres along each line.

(iii) Next, we find the point along each line that creates the greatest angle with the user. Nothing will be visible behind this point. The formula for calculating this angle is:

$$\tan^{-1}\left(\frac{height\ \ of\ \ mountain\ \ -\ \ height\ \ of\ \ user}{distance\ \ from\ \ user\ \ to\ \ mountain}\right)$$

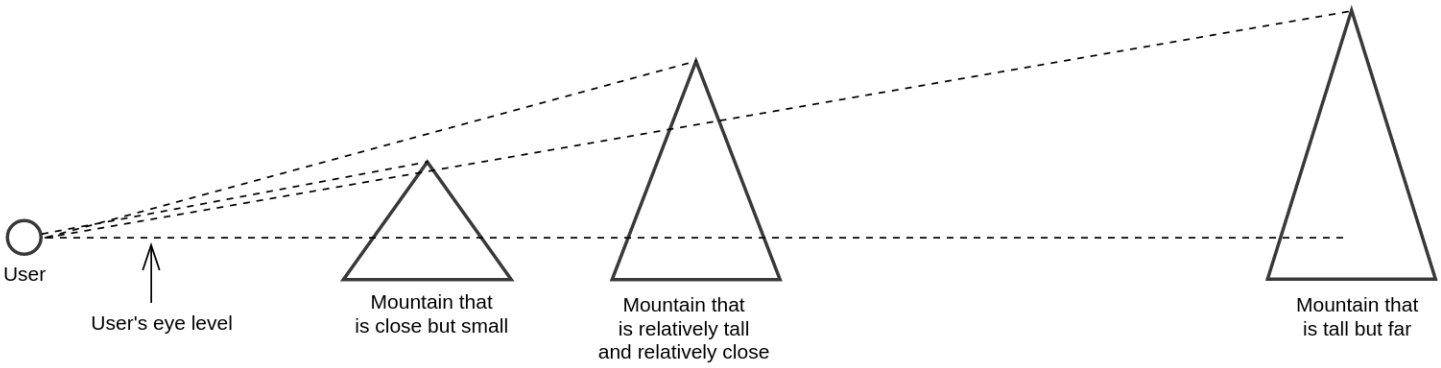


Figure 3.5 – ***A visualisation of the angles that each point along a line creates with the user. The mountain in the middle creates the largest angle, despite not being the tallest mountain. The user cannot see anything behind this mountain.***

The result looks something like this:

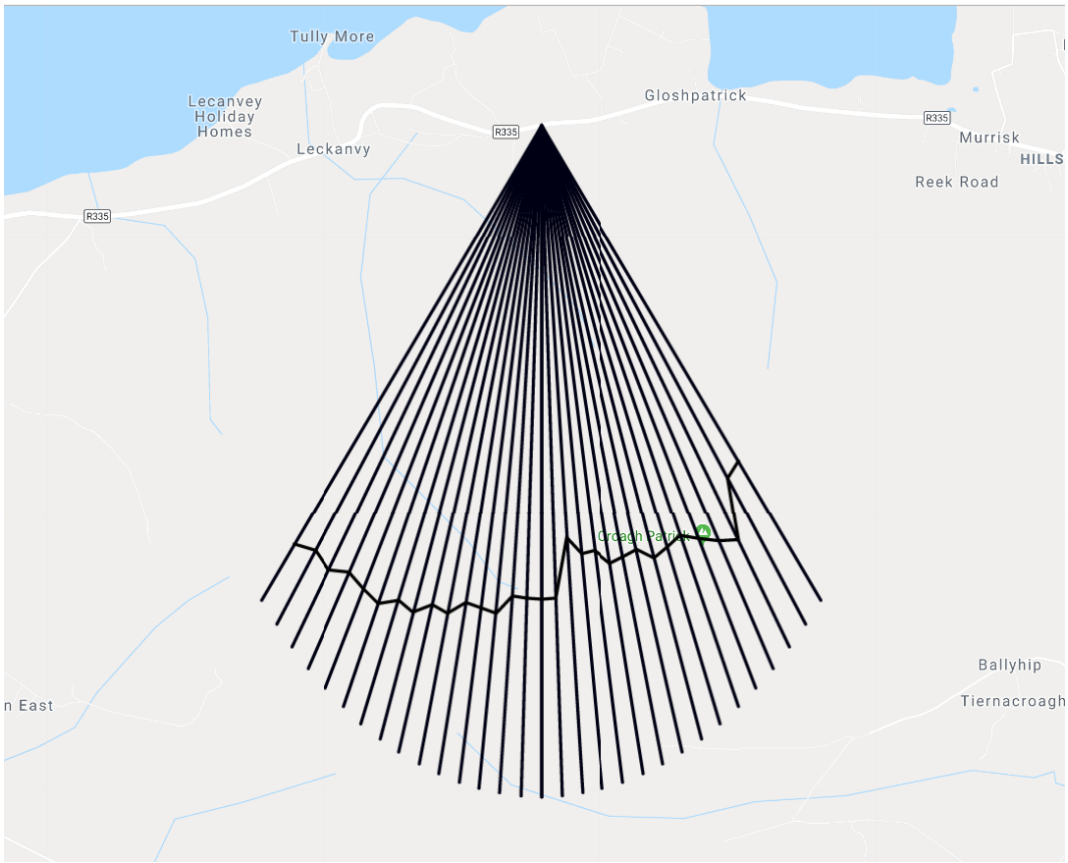


Figure 3.6 – ***After finding the greatest angles created by points along the lines of the fan, we draw a line which represents the visible horizon. We cannot see anything behind this line. Note that the peak of Croagh Patrick lies exactly on the line.***

(iv) Having done this for every line in the fan, we now have a list of the biggest angles from left to right. The next thing we do is scale them all to some value between 0 and 1, by dividing it by 45 degrees. This will be explained in greater detail in a minute. But first I must explain some assumptions that are made about the camera.

We are assuming that the camera has a field of view of 90 degrees along the y-axis (the top-to-bottom field of view). We are also assuming the user is pointing the camera parallel to sea-level, i.e., straight ahead, and not at the top of the mountain. To illustrate:

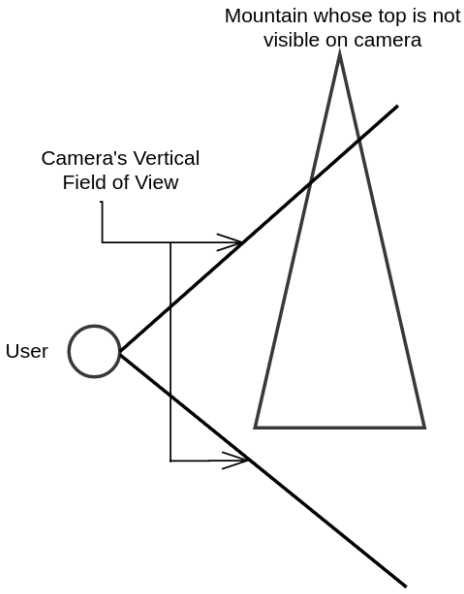


Figure 3.7 – ***The top of the mountain is creating an angle greater than 45 degrees with the camera's "eye level", so the top is cut off.***

Now I will explain why the values of the angles are scaled. This is what determines the y-values of each mountaintop on the screen. ***Note: In HTML, y-values increase as you move down. This comes into play in step 4 of the algorithm that is explained below***

- Each point on the horizon creates an angle with respect to the camera. We are assuming that points creating an angle greater than 45 degrees will be cut off.
- Therefore, a point that creates an angle of 45 degrees will be at the very top of the screen.
- It also follows that a point that creates an angle of 0 degrees will be in the **middle** of the screen.
- To determine a horizon point's y-value on screen, we follow this formula:
    1) *For every positive angle less than or equal to 45 degrees:*
    2) *Divide by 45*
    3) *Multiply the result by the height of the video feed*
    4) *Subtract this value from the height of the video feed*

- This results in high points being assigned y-values at the top of the screen, and points at eye-level being assigned values in the middle of the screen.

Example: a mountaintop creates an angle of 45 degrees with the user.
Divide this by 45 and we get 1.
Multiply it by the height of the video feed, we'll say the video feed is 100 pixels. 100* 1 = 100.
Now subtract this from the height of the video feed: 100 – 100 = 0. As I mentioned before, y-values increase as you move down the screen, which means 0 is the topmost row of pixels. So this mountaintop will appear at the very top of the screen.

(v) The next step is to find the x-coordinate of each point. This is quite easy, as we know we have 31 angles going left to right, so we just the angle's place in the array, divide it by the length of the array - 1, and multiply that by the width of the video feed. For example, taking the camera element to be 100 pixels wide, the x-coordinate of angle number 0 will be 0/30 * 100 = 0. Thus, it will appear on the far left. Angle number 30 will be assigned 30/30 * 100 = 100, i.e. the far right.

We now have a fairly accurate representation of the horizon:

Figure 3.8 – ***Mässersee in Switzerland. A very conspicuous mountain, providing a very distinctive horizon to draw.***

4. Find which mountains are visible to the user based on their location and orientation.

We now have a small section of Earth to search for mountains, and we know where to put their names on screen. The steps taken for finding which mountains are visible are as follows:

- Send a request to Google Places API for mountains within 3500m of the user.
- Filter those which lie within the user's field of view.

The first step here is very simple, only a couple lines of code. The second step, however, requires a bit of geometry.

Determining whether a mountain is within the user's field of view is analogous to determining wheter a point **p** lies within a sector, which is defined by its centre **c**, its radius **r**, and the two lines **left** and **right** coming from the centre. There is also the extra step of making sure it is on the visible horizon.

The algorithm for checking if a point lies within a sector is as follows:

- If the distance from the point to the centre is greater than the radius, then the point does not lie within the sector.
- Otherwise, if the angle created by the line from **p** to **c** , with respect to north, is greater than the angle created by the line **left** with respect to north, and less than the angle created by the line **right** with respect to north, then the point lies within the sector. See figure 3.9.

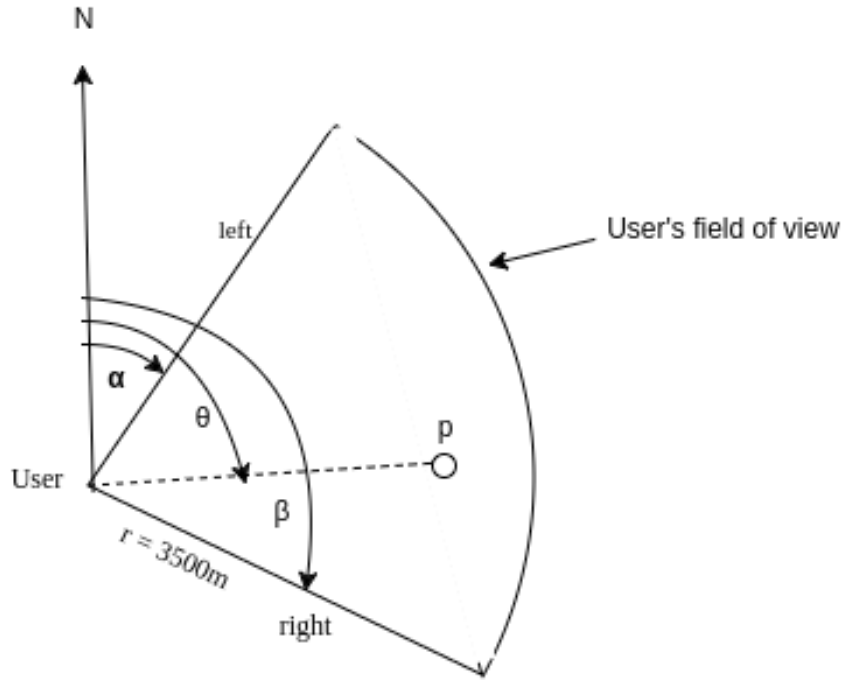


Figure 3.9 – ***If θ lies between α and β, and is closer than the radius r, then the point p lies within the sector.***

Finding α, β, and θ:

The formula to find the angles created by a line which is defined by two points **a** and **b** with respect to the north-south line is this:

- Find the slope of the line formed by a and b.
  - This is done according to the following formula:

$$\frac{latitude\ of\ a\ -\ latitude\ of\ b}{longitude\ of\ a\ -\ longitude\ of\ b}$$

- Convert the slope to degrees with $\tan^{-1}$(slope).
- This gives us the angle created by the line with the x-axis that we're all familiar with. However, this is different to the angle created with the N-S line.

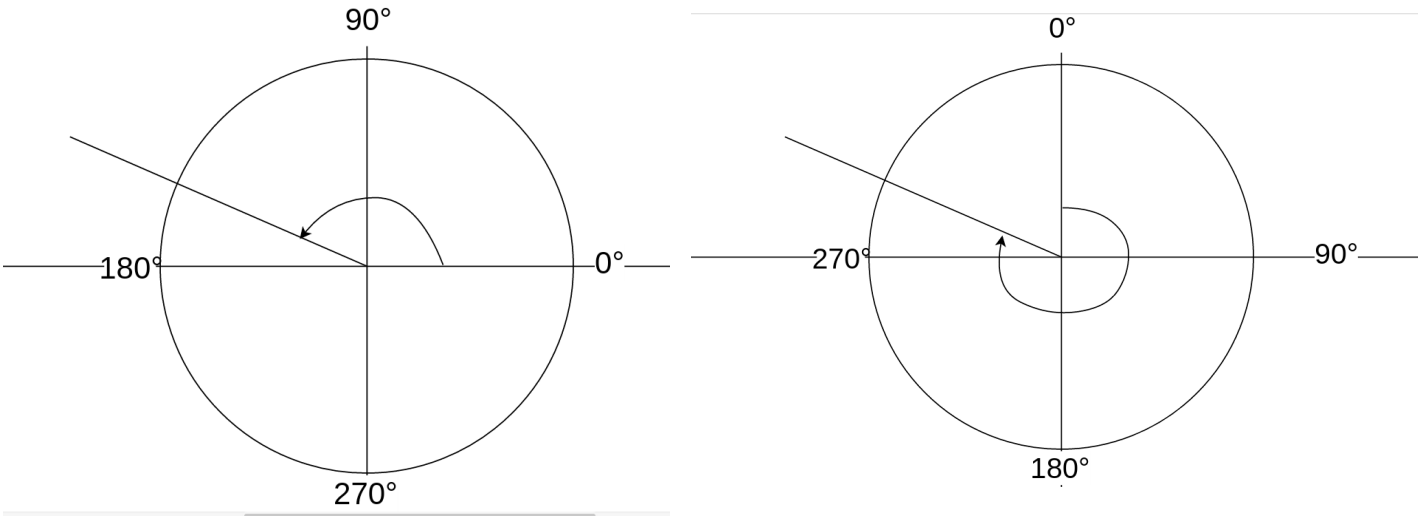
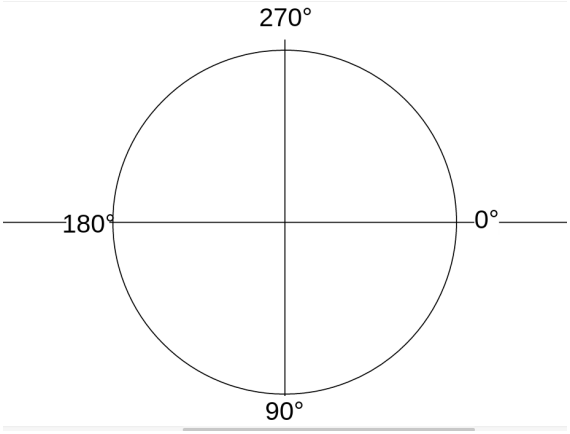


Figure 3.10 – ***What we have (left) vs. what we want (right).***

- Converting the angle from left to right consists of two simple steps, the order of which doesn't matter. Firstly, you'll notice that, for the diagram on the left, the angle increases in an anti-clockwise direction, whereas for the diagram on the right, the angle increases as it subtends clockwise. To flip an angle, simply subtract it from 360. This gives us a unit circle like so:



- Secondly, the two circles now differ from eachother by 90 degrees. To fix this, we add 90 to the angle (and modulo 360 in case of overflow). This gives us the angle in terms of a compass bearing, as illustrated in figure 3.10 (right).
- These steps can then be simplified to subtraction from 450 and modulo-ing with 360.

So now we can filter mountains based on whether they are in the user's field of view. But what about mountains that are hiding behind bigger mountains – i.e. mountains that are beyond the visible horizon? Well first, we need to find where they are in the user's field of view, in terms of

being left, right, or center. There is a very simple formula for this, using the angles formed in figure 3.9:

$$\frac{\theta - \alpha}{\beta - \alpha}$$

This returns a number between 0 and 1, 0 meaning that the mountain in question is on the far left of the user's field of vision, 1 meaning that the mountain is on the far right, and 0.5 meaning the mountain is dead ahead.

We now check if the mountain is on the visible horizon. To do this, we simply get the distance from the mountain to the relevant horizon point. If they are within a tolerance of eachother (100m), we can draw the name of the mountain onscreen. If not, then we disregard the mountain, as the user in all likelihood can't see it. To find the distance between two points, I used the Haversine formula[4][5], a function which takes in the coordinates of two points on a sphere and the radius of the sphere, and returns the distance between the two points:

Haversine formula:

$$a = \sin^2(\Delta\varphi/2) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \sin^2(\Delta\lambda/2)$$
$$c = 2 \cdot \operatorname{atan2}(\sqrt{a}, \sqrt{(1-a)})$$
$$d = R \cdot c$$

Source: https://www.movable-type.co.uk/scripts/latlong.html

Where φ_1 and φ_2 are the latitudes of the two points, λ_1 and λ_2 are the longitudes of the two points, $\Delta\varphi$ and $\Delta\lambda$ are the differences between the two latitudes and longitudes respectively, and R is the radius of the Earth in metres.

We finally have all the mountains that the user can see, and know where to draw their names. This is done using a HTML5 <canvas> element with some simple Javascript, which I will elaborate on in the following chapter.

4 - Implementation:

Selected Technologies:

HTML 5

Hypertext Markup Language is the standard markup language used to create webpages and web applications. It describes the structure of the webpage. Different browsers tend to interpret HTML in different ways (Internet Explorer for example, is notorious for being behind the curve when it comes to HTML features, that is to say, it doesn't recognise many element tags). For this web application, there is a somewhat minimalistic HTML layout – most of the HTML is contained in a single tag.

CSS 3

Cascading Style Sheets (CSS) is a style sheet language used for describing the presentation of a document written in a markup language like HTML. Whereas HTML describes the structure of a webpage, CSS describes the visual aspects of the webpage, such as layout, colours and fonts. For this web application, CSS is used to describe the size of the camera feed.

Javascript

Javascript is a high-level, dynamically-typed, interpreted programming language. Javascript, HTML and CSS are the three core technologies of the World Wide Web. Javascript programs are generally event-driven (i.e. the flow of the program is determined by events such as user actions, sensor outputs, or messages from other programs) and the Javascript in this web app is no exception.

Debian Linux

This web app is being hosted on SCSS's Linux server MacNeill, which is running a distribution of Linux called Debian 3.16. The web app can be accessed at:

https://macneill.scss.tcd.ie/~concannh/

Program Summary

HTML – The general structure:

The web app consists of an index.html file, and a functions.js file. The index.html file is what is displayed to the user, and the functions.js file contains the myriad mathematical functions the web app uses.

This web app's structure looks like this:
A <head> tag, in which all the CSS syles are defined, all the Google API libraries are imported, and all the mathematical functions from the funtions.js file, are imported.
A <body> tag, which contains:

- A tag for the camera feed.
- An tag on which the visible horizon is plotted, and overlaid on the camera feed.
- A <script> for configuring the camera feed (whether to use to the front-facing or rear-facing camera, whether to include audio, etc.).
- A <script> that contains the main program.

CSS – A summary of the layout:

The majority of the CSS in the web app is contained in the <style> tag which in turn is in the <head> tag. It defines the size of the camera feed to be the width and height of the browser window.

Javascript – A summary of the two scripts:

There are two scripts in this web app, one of which runs once, upon loading the web app, which gets permission to access the device's camera, and configures the camera feed, the other one which contains the main program which runs every 50^{th} *orientation event.*

What are Orientation Events?

Events in HTML are simply “things that happen” to HTML elements. When Javascript is used in HTML pages, Javascript can react to these events. This web app listens for Orientation Events – that is, an update on the orientation of the device. The most common use for an orientation event is when a user rotates their phone to landscape orientation to watch a video. The browser sees that the orientation has changed drastically, and redraws the webpage rotated 90º.

As previously mentioned, all modern phones have magnetometers which detect the phone’s orientation about the 3 axes. The phone’s operating system then sends this information to the browser, or indeed, any application that asks for it. This web app only cares about the orientation about the y-axis.

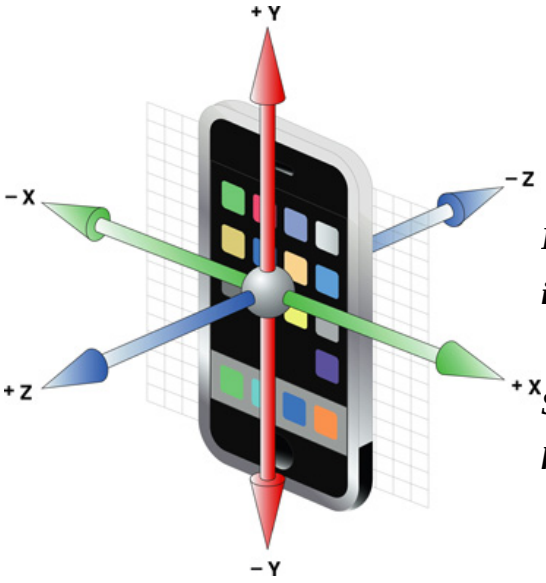


Figure 4.0 – How the three different axes are interpreted.

Source: https://i.stack.imgur.com/eyibk.jpg

By experimentation I found that phones generally send 10-20 orientation events per second. It is for this reason that the web app only listens to every 50 orientation events – listening to every orientation event overloads the browser and causes it to freeze.

Upon loading the webpage, a “deviceorientation” listener is added, with a callback to the “handleOrientation” function. This means that every time the browser receives an orientation event, the handleOrientation function is called. The orientation information is passed into this function as a Javascript Object.

The Anatomy of an Orientation Event:

The orientation event `event` consists of four values:

`event.alpha` – Representing the device’s orientation about the z-axis[6]

`event.beta` – Representing the device’s orientation about the x-axis[7]

`event.gamma` – Representing the device’s orientation about the y-axis[8]

`event.absolute` – A boolean indicating whether or not the device is providing orientation data absolutely (that is, in reference to the Earth's coordinate frame) or using some arbitrary frame determined by the device.[9]

The handleOrientation function also acts as the “main()” for this web app. What follows is a high-level look at the handleOrientation function, in pseudocode.

var orientationEventCounter = 50

Upon receipt of an orientation event:
If orientationEventCounter == 50, then
get device location
extract y-axis orientation data from the event
draw out a virtual fan representing the user’s field of view, using the user’s location and orientation.
get elevation data for the user’s field of view with Google’s Elevation Service
calculate the user’s visible horizon using the elevation data
find mountains that are visible to the user
draw mountain names onscreen
orientationEventCounter = 0
Else
orientationEventCounter = orientationEventCounter + 1

Most of this algorithm has been covered in detail in the previous chapter, except for the drawing of the names.

Drawing the Mountain Names:

HTML <canvas> tags allow you to draw lines, shapes, and text using Javascript. To overlay lines/text on a particular , simply create another immediately after it, with a <canvas> element inside.

```
<video id="video" autoplay></video>
<div id="overlay">
        <canvas id="canvas">No canvas support</canvas>
    </div>
</div>
```

Figure 4.1 – *A canvas being overlaid above a video feed.*

The that follows the being drawn on (in the case above, the one whose id is “overlay”) must have its CSS attributes defined like so:

```
#overlay {
    background: rgba(255,255,255,0);
    position: absolute;
    top: 0; right: 0; bottom: 0; left: 0;
    display: flex;
    align-items: center;
    justify-content: center;
    z-index: 5;
}
```

This CSS code does a few things:

- It makes the transparent – its *background* attribute is defined as “rgba(255, 255, 255, 0)”. RGBA stands for Red, Green, Blue, Alpha, where Alpha is the level of opacity of the element. Since it’s set to 0, the is now completely see-through.
- Makes the appear centered on the screen, so that its centre matches up with that of the camera feed.
- The *z-index* having a value of 5 attribute places the in front of the camera feed.

Deep in the Javascript, after all the relevant mountains have been found, and the x and y coordinates of their peaks have been calculated, we define the canvas size, and a variable for drawing the names onscreen.

```
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");
var video = document.getElementById("video");
canvas.width = video.clientWidth;
canvas.height = video.clientHeight;
ctx.beginPath();
```

Figure 4.2 – ***Here, the canvas.getContext() function allows you to draw on the <canvas> element. The size of the <canvas> is then defined to be the same as the video feed.***

Drawing the name of a mountain is done with the following line:

```
ctx.fillText(results[j].name, x + 45, y+10);
```

Where results[j] refers to the current mountain name, x and y refer to its coordinates, and the values added to them are to prevent the first letter of the name from being cut off by the border of the box containing it.

Before

After

5 – Evaluation and Conclusion

Successes

Functional requirements met

All functional requirements outlined in chapter 3 were easily met. The app works exactly as it should, without throwing any warnings or errors. Furthermore, the app has been tested on both iOS and Android and works on both operating systems.

Non-functional requirements met

Reliability

Throughout the development process, which lasted several months, the host server, *macneill.scss.tcd.ie*, was never taken down for maintenance, nor did it crash at any point. This satisfies the reliability requirement.

Usability

The web app consists of merely a camera feed, and thus minimal interaction with the user is achieved.

While the app as a whole is a success, it is still rather slow. I was unable to find a way to update the field of view and mountaintop names with high frequency, without causing the browser to become overloaded. Thus the app unfortunately does not update in real time like I had hoped it would at the beginning of the project. Rather, it updates every one to two seconds.

Difficulties Faced

Prior to starting this project I had next to no knowledge of Javascript. As mentioned in the Learning Resources section, I had to rely very heavily on Stack Overflow to get used to the language's idiosyncrasies. Namely, Javascript Object Notation (JSON), asynchronicity and callback functions. Coming from a Java and C background, Javascript's asynchronicity was very difficult to get used to, and I plain just didn't like it. I found it similar to learning Haskell, in that it was a different programming paradigm to what I was used to. Now that I'm well-versed in Javascript, my opinion on asynchronicity hasn't changed much, although I can appreciate some of the reasons why it's used.

The major difficulty I had regarding Javascript's asynchronicity was when I needed the elevations for the paths of the virtual fan that was drawn out from the user (recall figure 3.4 in chapter 3). Getting the elevations required an asynchronous function call to Google's Elevations Service, inside a for-loop. It originally looked like this:

```
for every path in the fan
    getElevations(currentPath, callbackFunction(elevations){
        store elevation data in a 2D array
    })

calculate visible horizon with elevation data
```

What you're looking at is a for-loop iterating over every path in the fan, getting elevation data for each path, and passing said data into a callback function which simply adds the elevation data to an array. This would mean that, once the for-loop completed, there would be a 2-D array of elevation data points for every path. At this point, the program would continue to calculate the visible horizon.

However, this proved to be impossible. Why? Because the for-loop would finish before all the elevation results returned from Google's servers, meaning I would end up with an array with data missing. What's more is, I would not know which elevation data was missing, since it all arrived from Google in a random order. The for-loop would send the requests and continue on through the rest of the program, despite the callback functions not having completed. This was a major

headache for me, and is in my opinion a flaw in the asynchronous programming paradigm, or at least something that could be fixed. Annoyingly, Google's Elevation Service does not have a synchronous version.

To get around this, I had to do a somewhat hacky recursive function:

```
index = 0
function loop(paths){
        getElevations(paths[index], callbackFunction(elevations){
                add elevations to global array
                index++;
                if(index < paths.length)
                        loop(paths)
                else
                        calculate visible horizon with elevation data
        })
}
```

This is a function that contains a callback function, wherein the callback function increments a global variable to iterate over the *paths* list, and then calls the outer function again. This means that each callback function is called only when the one preceding it has finished. It is not very pretty, but I would argue that it is a rather clever way of getting around Javascript's asynchronicity.

Some other difficulties I faced were those pertaining to coordinate geometry, trigonometry and geodesy. I hadn't really done much of any of these since the Leaving Certificate, and I found I had forgotten nearly everything I learned, even the trigonometric functions *sin*, *cos*, and *tan*, which my project obviously relies heavily on, and had to restart from square one. However, after a week or two I was back to my old standard.

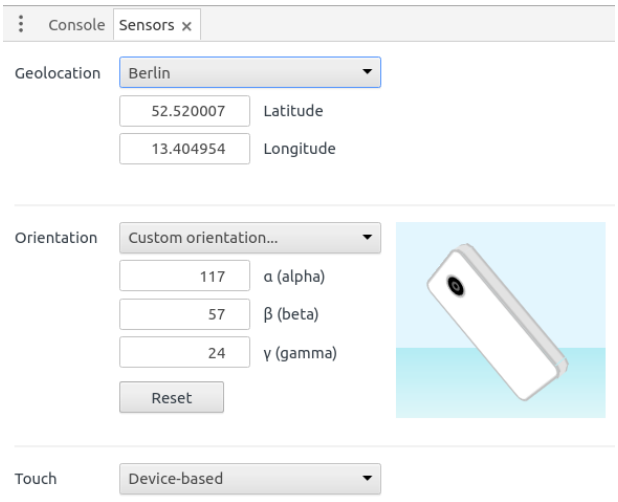
Testing

Throughout the development of the project, I tested my project with a “whitebox” testing method – I would have an idea of what the output should be, and compare it with the actual output. I hardcoded different locations and orientations into the program (e.g. Croagh Patrick to the left of the user, Mässersee straight ahead of the user). An example where the output was not as I expected was when I noticed that the horizon seemed to be inverted along the x-axis, i.e. there were dips where the mountains were, and rises where the valleys were. This is how I found out that in HTML, y-values increase as you go downward, as I mentioned in the Algorithm section.

The majority of testing was performed on desktop, due to Chrome Mobile not having a console feature. I tested the web app on a Samsung Galaxy Core Prime (2 years old), a Huawei MYA-L11 (2 months old) and an iPhone of unknown age and model. Surprisingly to me, it ran best on the iPhone. Unsurprisingly it ran very slowly on the 2-year-old Samsung.

I attempted once to use Chrome’s debugger, going line-by-line through the code, but it would skip over any callback functions that were called by any Google APIs, which I don’t see the purpose of. And since the majority of the code is within a callback function, I could access very little of it through the debugger. Thus I ended up just printing out variable values to the console, which suited me fine, as this is how I normally test and debug programs anyway.

Chrome made the development process very straightforward. It has a feature in the developer’s console that allows you to emulate a device by changing the screen size and location, as well as orientation along all three axes. You can also send orientation events at will. This is a great feature that Firefox, my primary browser, sorely lacks.



Conclusion

This report has been an in-depth look at the design, background and implementation of a web-based phone app that augments a camera image with the names of mountaintops. I have hopefully gone into enough detail such that anyone who has this write-up can easily reproduce this app, or a similar app, in the language of their choice, with very little research of their own. Or if not, at least gain some of the knowledge that I have accumulated over months and compressed into a few paragraphs and diagrams.

While this app is not completely finished, mainly in terms of its speed, I believe that optimizing it would be fairly easy to do, and that its speed could be at least doubled, as of the writing of this report.

Having finished this project, I would now consider myself to be proficient in JavaScript, having started as a novice. I am quite pleased with myself, because at the start I was reluctant to take on such a complex project in a language I knew so little. I now believe that being thrown into the deep end and learning the features and eccentricities of a language as you go is the best way to learn a new language, as opposed to taking online courses like those provided by codecademy.com and the like.

6 – Bibliography

[1] - https://www.rte.ie/news/2015/0608/706712-carrauntoohil/

[2] - https://www.wikitude.com/geo-augmented-reality/

[3] - https://play.google.com/store/apps/details?id=com.layar&hl=en

[4] - https://en.wikipedia.org/wiki/Haversine_formula

[5] - https://stackoverflow.com/a/1502821

[6] - https://developer.mozilla.org/en-US/docs/Web/API/DeviceOrientationEvent/alpha

[7] - https://developer.mozilla.org/en-US/docs/Web/API/DeviceOrientationEvent/beta

[8] - https://developer.mozilla.org/en-US/docs/Web/API/DeviceOrientationEvent/gamma

[9] - https://developer.mozilla.org/en-US/docs/Web/API/DeviceOrientationEvent/absolute

7 – Appendix

Code Snippets for Minor Functions

What follows is some code snippets of functions that were used but were not explained in great detail, or were of minor importance.

Code for determining whether a point lies within a sector as defined by its centre point, its furthest and leftmost point, and its furthest and righmost point:

```
function liesWithinSector(point, centre, left, right){
    if(distanceBetweenTwoPoints(point, centre) > DISTANCE_TO_HORIZON){
        return false;
    }
    var θ = angleCreated(centre, point)
    var α = angleCreated(centre, left)
    var β = angleCreated(centre, right)
    return(θ > α && θ < β)
}
```

```
//finds the angle created by a line formed by points a and b with respect to the north-south line
//in degrees
function angleCreated(a, b){
    var slope = (b.lat - a.lat) / (b.lng - a.lng)
    var gradient = Math.atan(slope);
    var degrees = gradient * 180 / Math.PI;
    //see if b is east of a, in which case add 180 degrees
    //this makes function only work if a is the user's location and b
    //is some other point
    if(b.lng < a.lng) degrees += 180;
    return (changeAngleToBearing(degrees));
}
function changeAngleToBearing(angle){
    return (360 - angle + 90) % 360;
}
```

All constants used in the program:

```
const DISTANCE_BETWEEN_ELEVATION_POINTS = 100; // to create a 2d array of elevations that are 100 metres away from eachother
const RADIUS_OF_EARTH = 6378100;
const ONE_DEGREE_OF_LATITUDE = 111000;
const DISTANCE_TO_HORIZON = 3500; //distance in metres, on average
```

Scaling functions that were mentioned but not elaborated on:

```
//returns theta as a scale of leftness or rightness
function xCoordOfLandmark(point, centre, left, right){
    var θ = angleCreated(centre, point)
    var α = angleCreated(centre, left)
    var β = angleCreated(centre, right)
    return (θ - α) / (β - α);
}

function yScale(arr){
    var scaledValues = [];
    for(var i = 0; i < arr.length; i++){
        scaledValues.push(arr[i] /(Math.PI/4));
    }
    return scaledValues;
}
```

What an Elevation request looks like:

```
elevator.getElevationAlongPath({'path':paths[i], 'samples':35}, (elevations, status) =>{
```

What a a request to get the phone's location looks like:

```
navigator.geolocation.getCurrentPosition(function(position) {
    var pos = {
        lat: position.coords.latitude,
        lng: position.coords.longitude
    };
```

Drawing one of the fan lines. More specifically, path number i, on a "map" element:

```
new google.maps.Polyline({path: paths[i],strokeColor: '#000011',strokeOpacity: 1,map: map});
```

Creating a request for all mountains within 3500m of the user, sending it to Google, and iterating over the results:

```
var posLatLng = new google.maps.LatLng(pos.lat, pos.lng);
var request = {
    location: posLatLng,
    radius: DISTANCE_TO_HORIZON,
    type: ['natural_feature']
  };

var service = new google.maps.places.PlacesService(map);
service.nearbySearch(request, (results, status)=>{
    if(status == google.maps.places.PlacesServiceStatus.OK){
        for (var j = 0; j < results.length; j++) {
```

Images, Graphs and Diagrams, and their sources

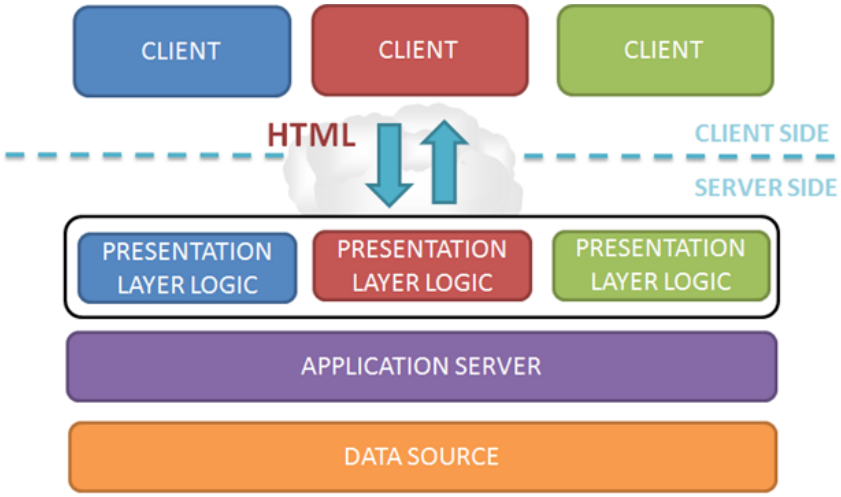


Figure 2.0 – *Source:*

https://devcentral.f5.com/articles/the-new-distribution-of-the-3-tiered-architecture-changes-everything

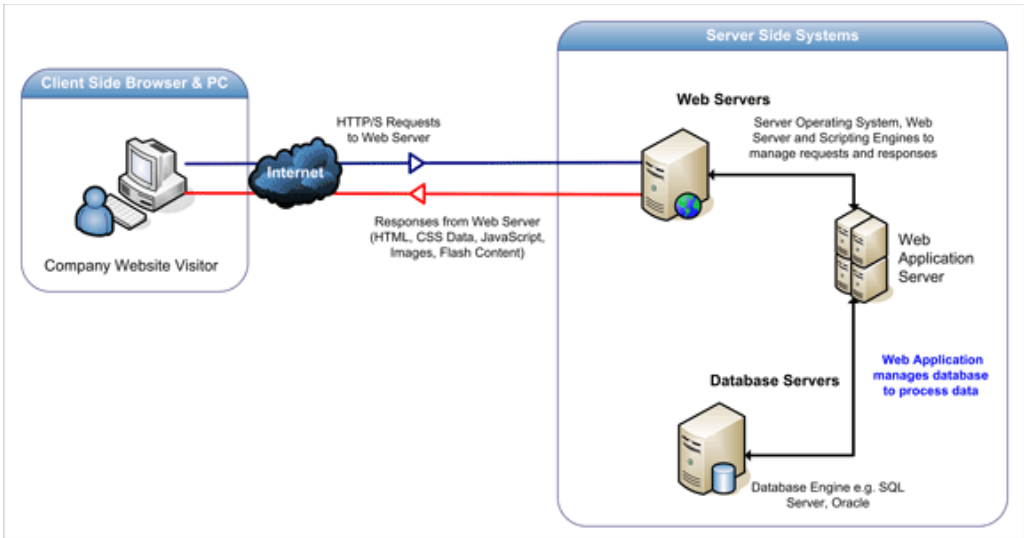


Figure 2.1 – *Source:*

https://www.acunetix.com/websitesecurity/web-application-attack/

Figure 2.2 – *Source:* *https://www.wikitude.com/geo-augmented-reality/*

Figure 2.3 – *Source*: *http://www.quotenet.nl/var/hearst/storage/images/quote/nieuws/layar-geen-onenigheid-met-investeerder-prime-28157/384317-1-dut-NL/Layar-geen-onenigheid-met-investeerder-Prime1_crop700x350.jpg*

Figure 2.4 – ***Taken from the application early in production***


```
▼ Object
  ▼ geometry: Object
    ▼ location: _.K
      ▶ lat: ()
      ▶ lng: ()
      ▶ __proto__: Object
    ▶ viewport: _.rc
    ▶ __proto__: Object
  ▼ html_attributions: Array[0]
      length: 0
    ▶ __proto__: Array[0]
    icon: "https://maps.gstatic.com/mapfiles/place_api/icons/geocode-71.png"
    id: "ae7f399e279596782024a875d4a4b3ae26937d5c"
    name: "Croagh Patrick"
  ▶ photos: Array[1]
    place_id: "ChIJb5tRRJ9-WUgRMnx98idVlNU"
    rating: 4.7
    reference: "CmRbAAAALX4uRwRvZfaoEqhqe_QcsyK4NI2roeS7hu9ARTEi0y6o17SjiohSC3OrZIEE_hlrUOhluDoOG_a-0sojouCXz
    scope: "GOOGLE"
  ▶ types: Array[2]
  ▶ __proto__: Object
```


Figure 2.5 – ***Taken from the application.***

```
elevator.getElevationAlongPath({'path':paths[i], 'samples':35}, (elevations, status) =>{
```

Figure 2.6 – ***Taken from the application***

```
▼ Array[35]                                                    croaghpatrick.html:208
  ▼ 0: Object
      elevation: 23.68660354614258
    ▼ location: _.K
      ▶ lat: ()
      ▶ lng: ()
      ▶ __proto__: Object
      resolution: 152.7032318115234
    ▶ __proto__: Object
  ▶ 1: Object
  ▶ 2: Object
  ▶ 3: Object
  ▶ 4: Object
  ▶ 5: Object
  ▶ 6: Object
  ▶ 7: Object
  ▶ 8: Object
  ▶ 9: Object
  ▶ 10: Object
  ▶ 11: Object
  ▶ 12: Object
  ▶ 13: Object
  ▶ 14: Object
  ▶ 15: Object
  ▶ 16: Object
  ▶ 17: Object
```

Figure 2.7 – ***Taken from the application***

Figure 2.8 – ***Taken from the application***

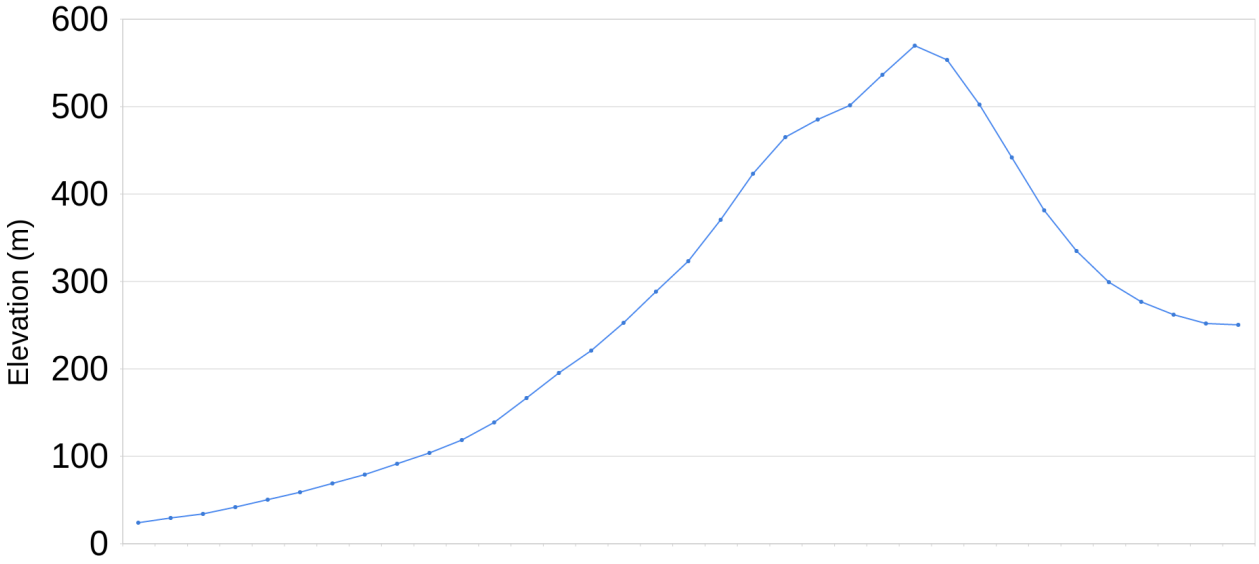


Figure 2.9 – ***Taken from the application early in production***

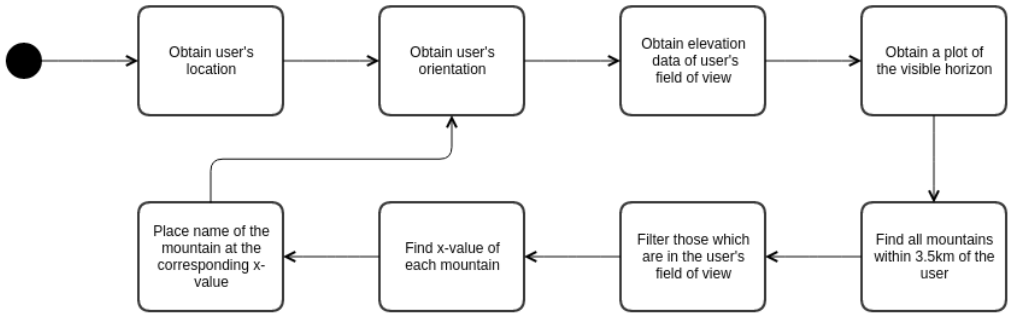


Figure 3.0 – ***Created on gliffy.com***

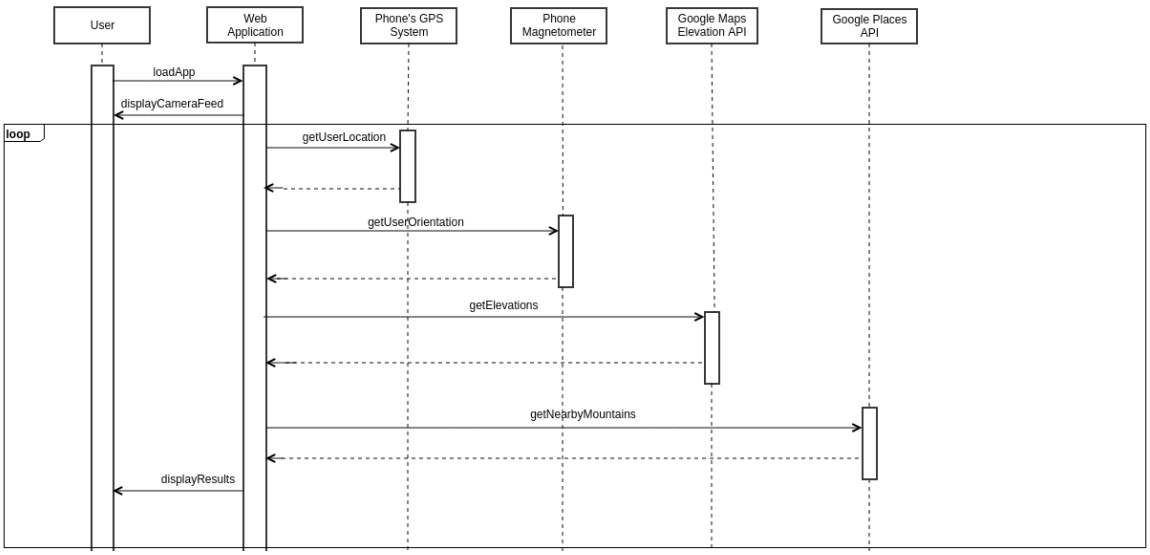


Figure 3.1 – ***Created on gliffy.com***

Figure 3.2 – ***Created with mockuphone.com***

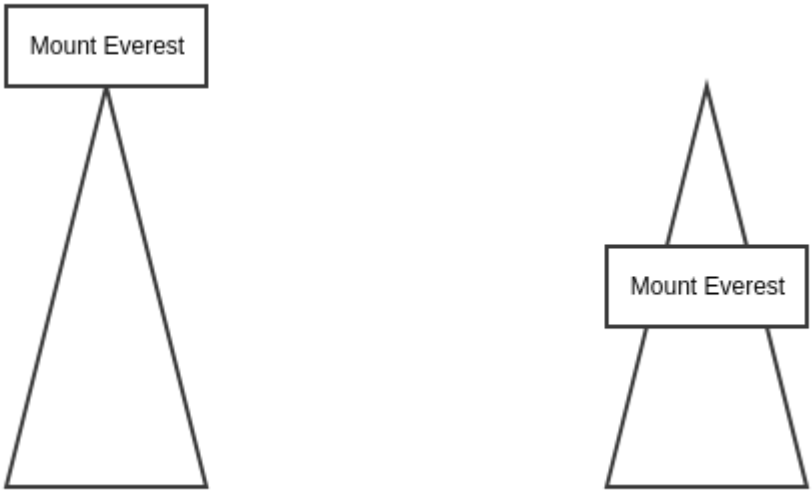


Figure 3.3 – ***Created with gliffy.com***

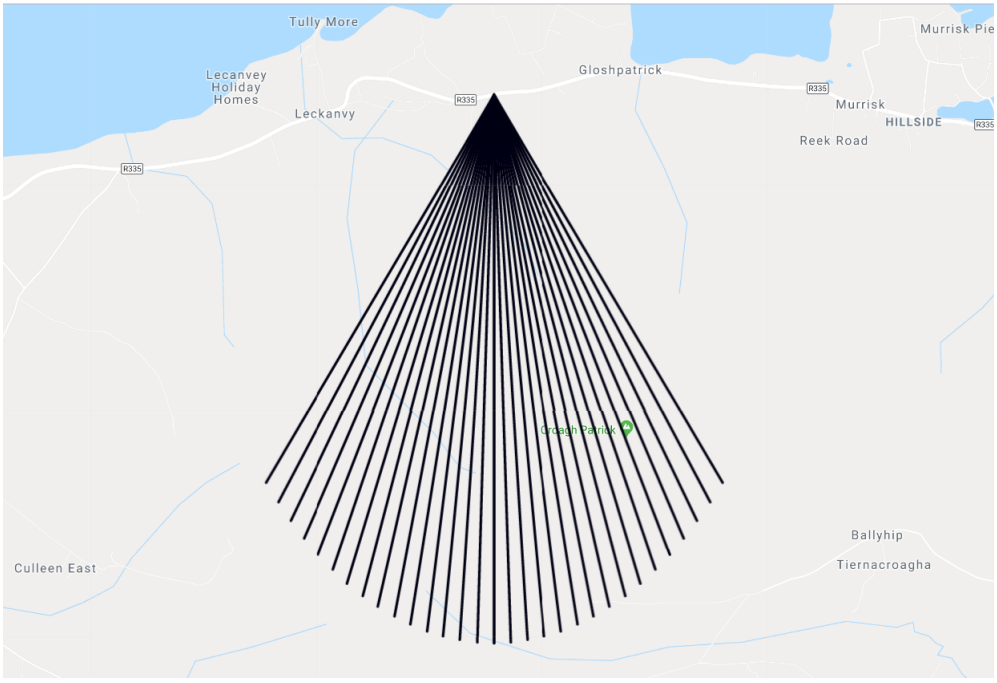


Figure 3.4 – ***Taken from application early in production***

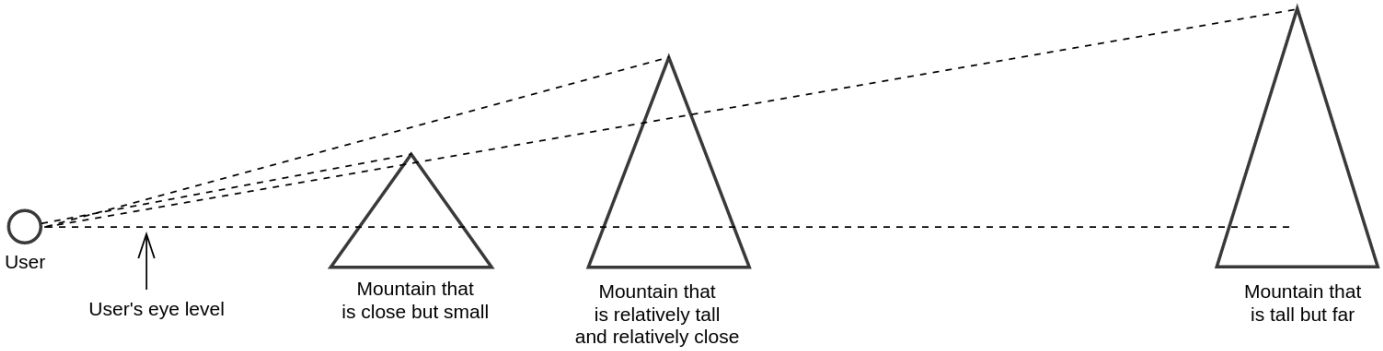


Figure 3.5 – ***Created with gliffy.com***

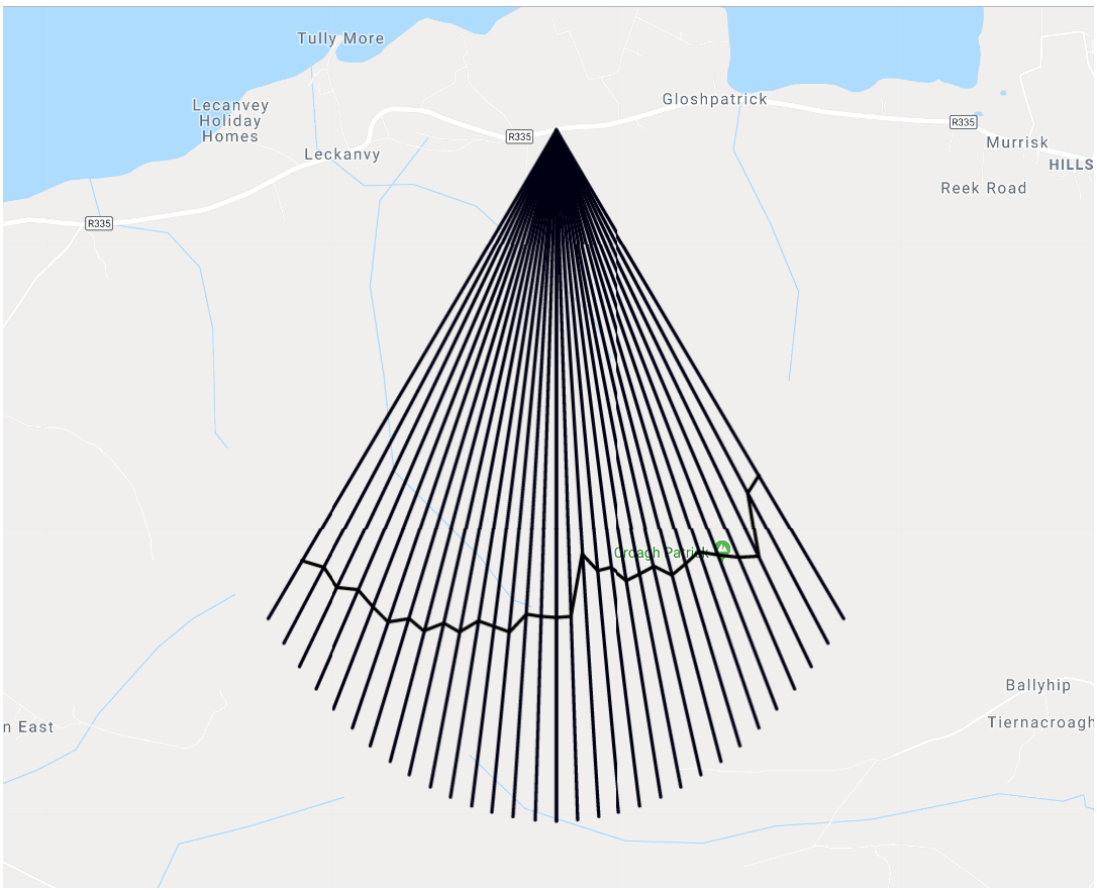


Figure 3.6 – ***Taken from application early in production***

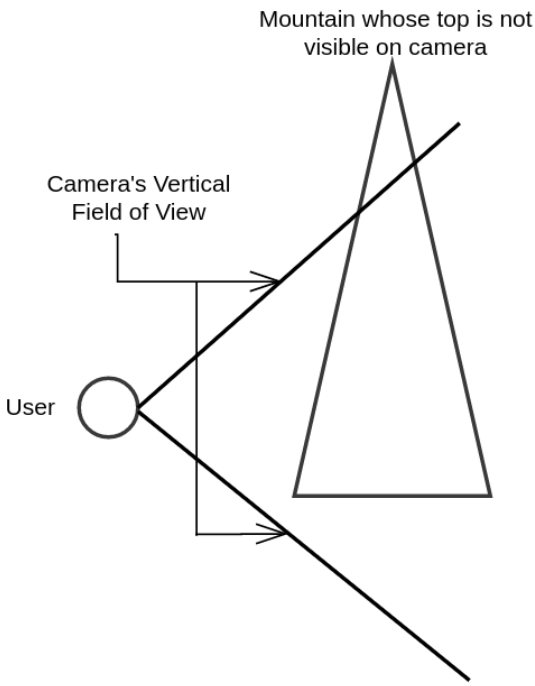


Figure 3.7 – ***Created on gliffy.com***

Figure 3.8 – ***Screenshot taken from desktop version of application***

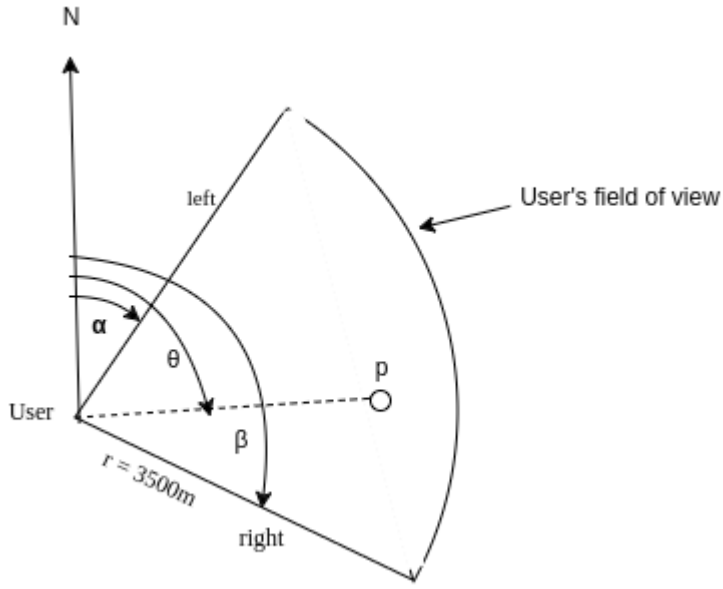


Figure 3.9 – ***Created with draw.io***

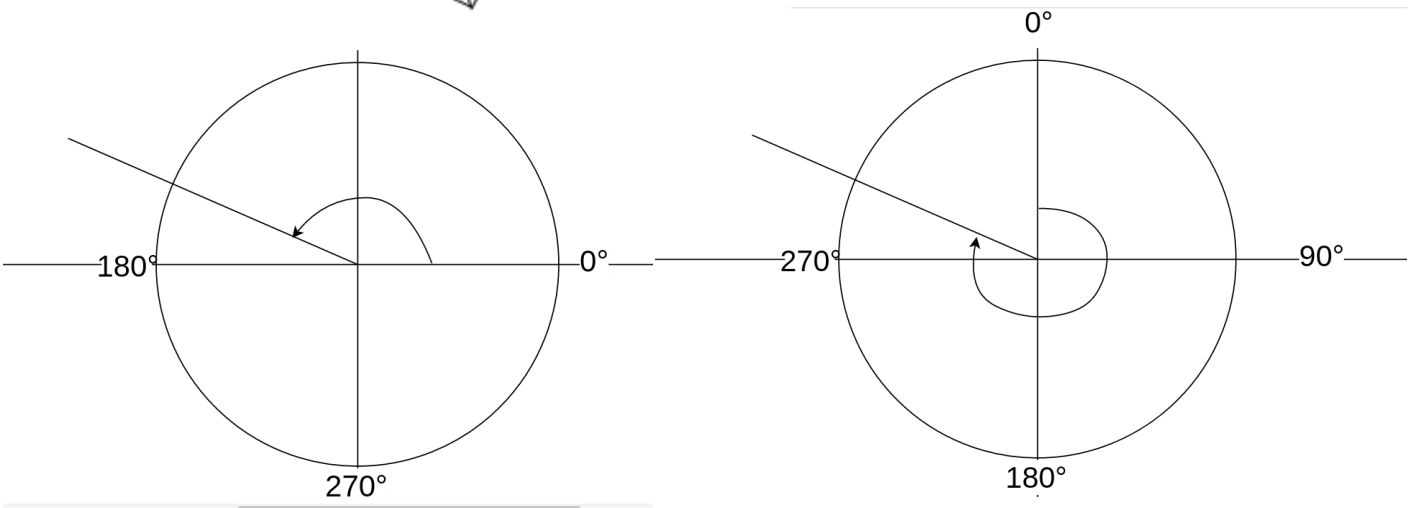


Figure 3.10 – ***Created with draw.io***

Created with LaTeX:

$$\frac{latitude\ of\ a\ -\ latitude\ of\ b}{longitude\ of\ a\ -\ longitude\ of\ b}$$

$$\frac{\theta\ -\ \alpha}{\beta\ -\ \alpha}$$

Source Code

In the attached CD are the index.html and functions.js files, which are all you need to run this application. The application is also hosted at **macneill.scss.tcd.ie/~concannh**