Navigating Across Piste Maps

Andrew James Marmion

September 2017

Dissertation submitted in partial fulfilment for the degree of
Master of Science in Software Engineering

Computing Science and Mathematics
University of Stirling

Abstract

DOGFI.SH Mobile Ltd (Dogfish) is a mobile applications development company based in Stirling, Scotland. They currently develop mobile applications for their clients using a variety of native and cross-platform solutions. One of their clients, Crystal Ski, is in the process of having their 2013 application re-written to take advantage of technological improvements and updates to their own systems. The current application was built using Titanium but due to the constraints of using a cross-platform solution the application did not run as expected, especially when interacting with native APIs. Dogfish proposed a complete re-write of the application. This meant that the existing location algorithm was to be re-written natively and if possible to update it so that it performed more accurately.

Initially an in-depth analysis was undertaken of the existing application and surrounding systems in order to understand how the location system worked. Although the application provided a way to show a user their location on the map, and it could show preconstructed recommended routes, it was unable to give a user directions across the piste map. It was discovered during this time that the recommended routes system could be upgraded so that it would be able to provide users with a way to find the shortest path across the piste map.

The objective of this project was to convert and improve the location finding algorithm, create a routing algorithm based on the recommended routes system, and construct two native applications to showcase the functionality so that it could be presented to Crystal Ski.

The project was completed using an Agile methodology. It was split into three epics: the location epic, the routing epic and finally the application epic. These epics allowed for clear and distinct points of completion during the development lifecycle. Testing was actively done, with Dogfish's quality assurance team checking completed work against the user stories and their assessment criteria.

At the end of the project two showcase applications were handed over to Dogfish, each contained the libraries: implemented in the native language for each application. The location algorithm was improved upon and now utilises an affine transformation: this is more mathematically correct than the similar triangles method. The routing algorithm was also extended so that it now works on any map type and not just piste maps. Dogfish was very happy with the result and these libraries are currently being implemented in the re-write of the Crystal Ski application. There are also other clients who are interested in including the libraries in their applications.

Attestation

I understand the nature of plagiarism, and I am aware of the University's policy on this.

I certify that this dissertation reports original work by me during my University project except for the following:

- The data about plots, nodes and routes was given to me by Dogfish and has been used with their express permission.
- The method for finding a user's location outside of the plots, and the similar triangle method was written by Paul Burrows (CTO Dogfish).
- User Stories were constructed from the user stories given to Dogfish.
- The Priority Queue [43] for Swift was written by David Kopec and has been used in accordance with its licence.
- JSON parsing was accomplished with Klaxon [41] in Android and SwiftyJSON [42] in Swift.
- The scrolling tileview in Android was accomplished by using TileView [44].
- The notification in Swift was accomplished by using Toast-Swift [49].
- Dependencies were loaded in Android using Gradle [47] and in Swift using Carthage [45].
- Dijkstra's Algorithm [12] was used to calculate the shortest path across the node and routes.
- The Affine Transformation calculation was checked by Dr Jessica Enright (University of Stirling).
- The statistical tests were recommended by Katie Howie (Teaching Fellow – University of Stirling).

Signature **Date 08/09/2017**

Acknowledgements

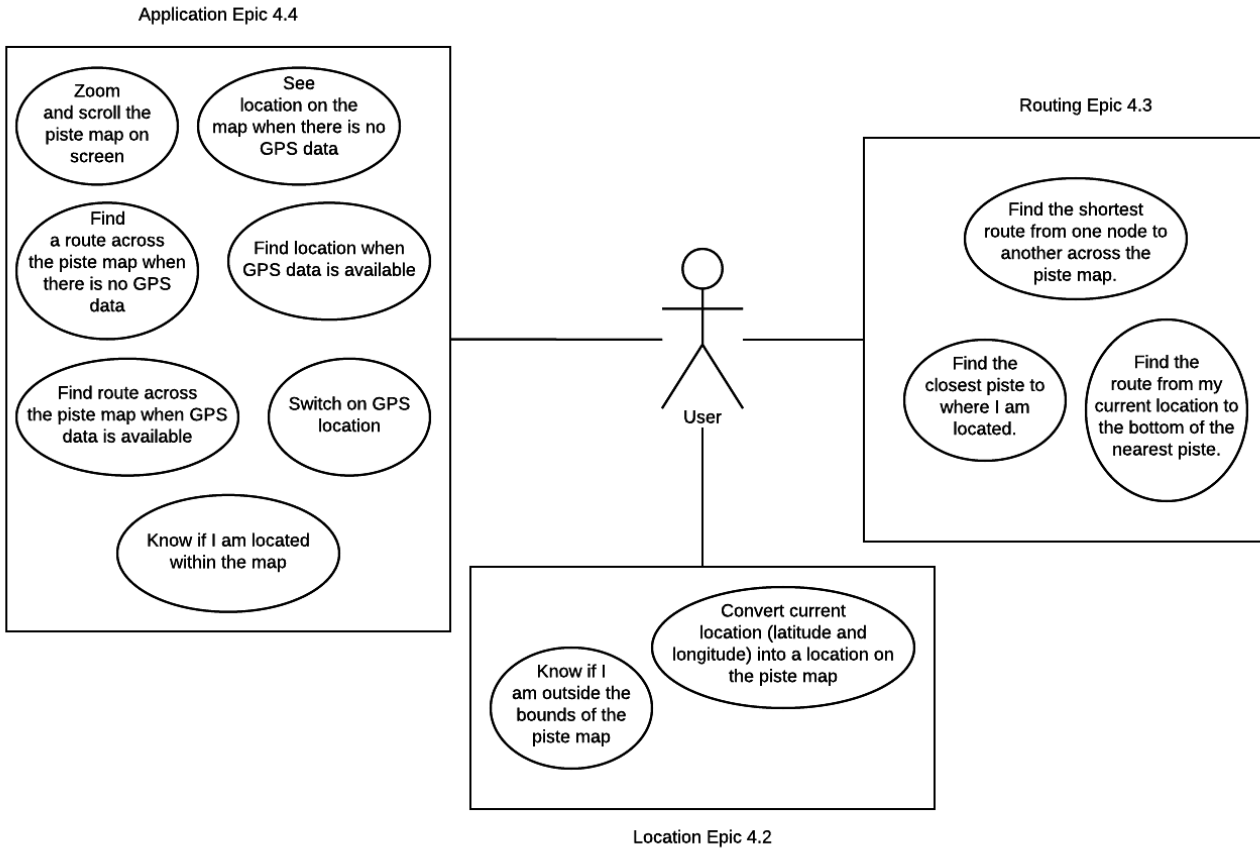
Firstly I would like to thank Dr Mario Kolberg for being my supervisor. His guidance, help, understanding, sense of humour and infinite patience with my self doubt, without these completing this dissertation would not have been possible. I would also like to thank Dr Jessica Enright for her help in checking the correctness of my affine transformation, and Kate Howie who helped me decide which statistical tests I could use to compare the results from the location algorithm.

Secondly I would like to thank DOGFI.SH Mobile Ltd as a whole and to thank all the employees who helped me over the course of this project. A special thanks to Paul, Mike, Colin, Klaudia, Jordan and Laura, their support helped make this project possible.

I would also like to thank Martin Hofhanzl for being a rock throughout the whole degree program. He has been a fantastic confidant and has helped push me to be a better developer. The collaboration and advice has been very much appreciated.

Finally, I would like to thank my family for listening attentively while I talked at them about problems that they couldn't understand. For reading the copious number of drafts and for supporting me when I felt overwhelmed. I would also like to thank my girlfriend Sharon, whose constant support and reminders to take a break helped me immensely.

Table of Contents



List of Figures

List of Equations

List of Tables

1 Introduction

1.1 Background and Context

DOGFI.SH Mobile Ltd (Dogfish) is a mobile applications development company based in Stirling, Scotland. They currently develop mobile applications for their clients using a variety of native and cross-platform solutions. One of their clients, Crystal Ski, is in the process of having their 2013 application re-written to take advantage of technological improvements and updates to their own systems.

The mobile application, originally written in JavaScript by Dogfish, used the cross-platform SDK Titanium. The application has gone through several versions over the years, these updates have been added to the existing application. This has meant that the application has suffered from being bloated, due to having old classes and methods that are no longer required or have been superseded. The user interface was designed and built 4 years ago and has started to look dated. The current application, version 3.0.5, has suffered from problems with regard to accessing native APIs on both Android and iOS, this has meant that the GPS functionality of the application has not provided the user with the experience that Crystal Ski was hoping for. A complete re-write of the application, by Dogfish, has been undertaken with a view to improving upon the existing systems.

The application contains an algorithm that converts a user's latitude and longitude into an x and y coordinate, allowing the user's location to be shown on the piste map. Due to the issues with accessing the native APIs the application's ability to correctly position a user on the piste map has become compromised. Although this feature was once lauded the users have now chastised it due to the fact that it is unreliable.

The current application is also missing a very obvious feature, one of navigation. Some ski resorts are large and complex and can be difficult to navigate. Although the application offers recommended routes, routes that have been selected by Crystal Ski, it offers no way for a user to know how to get from point A to point B. This recommended routes system could be leveraged to become an actual routing system, allowing users to navigate quickly and safely across a piste map. Currently there is no application on the market that offers this functionality, and by incorporating this into their application they would be making their application a world leader.

1.2 Scope and Objectives

There are three main objectives to the project. The first two objectives were to develop two libraries: one for calculating the user's location and the second for calculating routes across

the piste map. The third and final objective was to encase them in a showcase application ahead of being deployed in the upcoming release of Crystal Ski's new application.

As mentioned in section 1.1, the original application had the ability to show the user's location on the piste map. The first objective was to convert this algorithm, which was written in JavaScript into code that would run natively on both iOS and Android, as there should be a separate iOS and Android implementation. The resulting location library should take in a user's location as a latitude and longitude and convert it to a point on the piste map. The algorithm used should be efficient and accurate, being able to update the user's location on the map several times a second.

The second objective was the creation of an offline navigation library. The recommended routes system, which provides preconfigured piste runs for skiers: i.e. quarter day, half day, and full day skiing suggestions, could be enhanced so that user's could navigate on the fly from a chosen start point to a chosen finish point. This routing system should be able to work with or without a GPS signal, and it should allow a user to navigate from the closest node to where they are to the closest node to their destination. This algorithm had to be written natively for both iOS and Android.

The final objective of the project was to create two applications, written natively for both iOS and Android. The application should have a scrollable and zoomable map, it should interact with the device's GPS system to get the user's location. When the user is trying to find a route across the pistemap it should allow the user to select the start and/or end locations. When a route has been found across the piste map it should display this route on the map. These applications would be a vehicle to show the functionality of the location and routing libraries.

The project was undertaken as partial fulfilment for Masters of Science in Software Engineering at the University of Stirling. It was carried out with an agreement between the student, the University of Stirling and Dogfish. Dogfish offered the student the ability to gain valuable industry experience while freeing up their developers to work on other projects. The student undertook the project working as a full-time intern at Dogfish: working as part of a Agile development team using Dogfish's practices and methodologies, with a view to have a completed set of libraries by the end of the project.

1.3 Achievements

The libraries were delivered to Dogfish along with two showcase applications. The libraries each written natively in Swift and Kotlin: met and, in some cases, exceeded the objectives as

set out in section 1.2. The solution was built in line with the user stories, which had been written in conjunction with Dogfish.

The location algorithm is based upon a set of corresponding points. Points on a geographic map are paired with points on the piste map giving the ability to use triangles to find where the user is located. First the closest point to the user's location on the geographic map is found; then the smallest triangle that the user is in on the geographic map is found. This triangle will correspond to a triangle on the piste map; from here it is possible to find a user's location on the piste map by using one of two methods. The existing method by Dogfish uses similar triangles to find the location, however these triangles may not be similar and lead to erroneous results. A more mathematically sound method is to use an affine transformation, since all triangles are affine and there exists a transformation that will map a triangle to any other. Solving a system of equations, a transformation that maps the geographic triangle to the piste triangle can be calculated. Then using this transformation it is then possible to convert the user's current location to a location on the piste map. Dogfish's original method was converted from JavaScript to Swift and Kotlin, and the Affine transformation was written natively, first in Swift and then converted to Kotlin.

The routing algorithm was built from the foundations of the original applications recommended routes system. There existed an underlying graph that could be leveraged so that users could do point-to-point navigation across the piste map. The initial specification was to construct an algorithm that found the shortest path between closest node to the user's current location (either by tapping on the screen or by using the device's GPS system) and then to the closest node to the user's chosen destination. The algorithm uses an implementation of Dijkstra's algorithm with a min-heap priority queue to traverse the underlying graph and find the shortest path. Like the location algorithm, this was first implemented in Swift and then converted to Kotlin. The algorithm was further extended in two ways. Firstly the route from the user's current location to the nearest node is now calculated and added to the shortest path. This makes for a better user experience, as it does not assume that the user is at the node that they are starting from. Secondly, the algorithm was originally intended to only work on piste maps, maps that have single direction routes, but it was updated to work with bi-directional routes. It does this by checking several different possible routes and choosing the shortest one. The algorithm can now be used on many different types of maps. Finally the shortest path was required to be drawn on the piste map. This necessitated two different implementations, one for iOS and the other for Android as the scrolling piste maps in the applications operate differently. The route is shown in multiple colours, based on the type of run/lift that the user is to travel along.

The algorithms are separate and do not need each other to function. It is possible to create a point-to-point navigation system without using the GPS functionality. Similarly it is also possible to show a user's location on the map without providing the routing functionality. This separation of algorithms means that they can be used in a variety of different ways.

Two showcase applications were constructed. They were natively built for iOS and Android devices. They consist of a scrollable map view; this required different implementations for each. In iOS it was possible to use native user interface elements, however in Android it was required to use an external library, TileView, to achieve a similar result. The applications interact with the device's GPS systems allowing the user to see their location on the piste map in real-time. It also employs touch screen gestures so that users can pick a start/end point for the routing system.

The libraries were tested by Dogfish and will be used in the current re-write of the Crystal Ski application. The showcase applications have been shown to other clients and there is interest in using these libraries in their future applications.

1.4 Overview of Dissertation

The dissertation is structured into the following chapters.

Chapter 1: Contains the background and the context followed by a definition of the scope and objectives of the project. It also discusses what achievements were made in the project before going on to discuss the overview of the structure of the dissertation.

Chapter 2: Contains the State of the Art. This contains discussions on the current system that Crystal Ski has in place: it's CMS and the Titanium built application. It then goes on to discuss the different algorithms that could be used for both the location and routing, before finally looking at the different native and non-natives ways that the showcase applications could be built.

Chapter 3: Covers Agile Theory and how it is used in iterative and incremental development. It then goes on to discuss the hybrid version of Agile that Dogfish uses.

Chapter 4: Discusses the requirements of the project and how it was split into 3 epics, which were made up of user stories.

Chapter 5: Discusses the design of the data structures and the applications. How the application interacts with the CMS, the possibility of a single language library, and the differences that are to be expected between the different native implementations.

Chapter 6: Covers the implementation. The location algorithm: the original Dogfish implementation and the new affine transformation method. The routing algorithm: its

construction and the improvements that were made to it. It explains in detail the differences between the application implementations in the different native environments.

Chapter 7: This chapter discusses the testing, which was done on the algorithms and the applications that were subsequently created.

Chapter 8: The last chapter gives a summary of the project as a whole, with an evaluation of the completed work. It then goes on to discuss future work and improvements.

2 State-of-The-Art

This chapter will explore the existing application and its CMS, solutions that already exist for finding your location on a piste map, solutions that already exist for routing, and a detailed analysis of the algorithms that could be used to solve the problem in each case.

2.1 The Existing Application

DOGFI.SH Mobile Ltd (Dogfish) developed the existing Crystal Ski application 4 years ago, with initial publication on the 20 December 2013. It has since gone through several iterations with version 3.0.5 being the most current. The application was built with Titanium's Appcelerator, a cross platform mobile application development solution.

The application has a backend content management system (CMS) that allows Crystal Ski to update the information on the application without the user having to re-download the whole application. The CMS also allows for the creation of recommended routes. These routes are then overlaid on the piste map and the user can follow them for a fun day of skiing.

Reading reviews of the application on both the Apple App Store and the Google Play Store show that the application is highly lauded but at the same time users are not happy due to issues with GPS functionality. These problems mainly come from building the application in Titanium and the difficulties that it can have interacting with native APIs. The original choice for building in Titanium was due to the fact that it allows for quick prototyping and deployment of applications that have simple functionality and mirror what you would expect on a website. The second reason they chose to use Titanium, is that using a cross platform solution, instead of writing natively, means that one design can be built for both Android and iOS devices and it requires a smaller development team. The final reason for using a cross-platform solution is cost, it clearly is cheaper to build a cross-platform application as you require less developers Over the years new features have been added to the existing application causing it to become unwieldy and cumbersome, sacrificing performance for increased functionality.

2.1.1 Mobile Application

The mobile application was written in Titanium [57]. Titanium uses Alloy [50], which is a framework that allows the use of a Model-View-Controller [55] architecture in the application. Titanium uses JavaScript and XML to build applications. The XML defines the underlying look of the application. Each page or even element in the application has to be defined in XML. The functionality is provided by JavaScript written in specific files, usually the starting

point is an index.js file. JavaScript is a classless language, using objects and functions to mimic the functionality that would usually come with a class.

An interpreter sits between the application written in JavaScript and the devices' built in APIs. The use of an interpreter slows down the performance of the application giving the user a poorer experience than if the application had been coded natively. This is specifically true as applications become more complex, demanding more of the interpreter. It also makes working with the devices APIs more difficult and at times functionality can be lost. The application allows the user to manage their booking with Crystal Ski, find out information about the resort (including piste maps and recommended routes) and allows the user to see their location on the piste map and the location of their friends. Figure 1: in the left hand image shows a scrollable and zoomable piste map, and in the right hand image shows a recommended route being displayed.

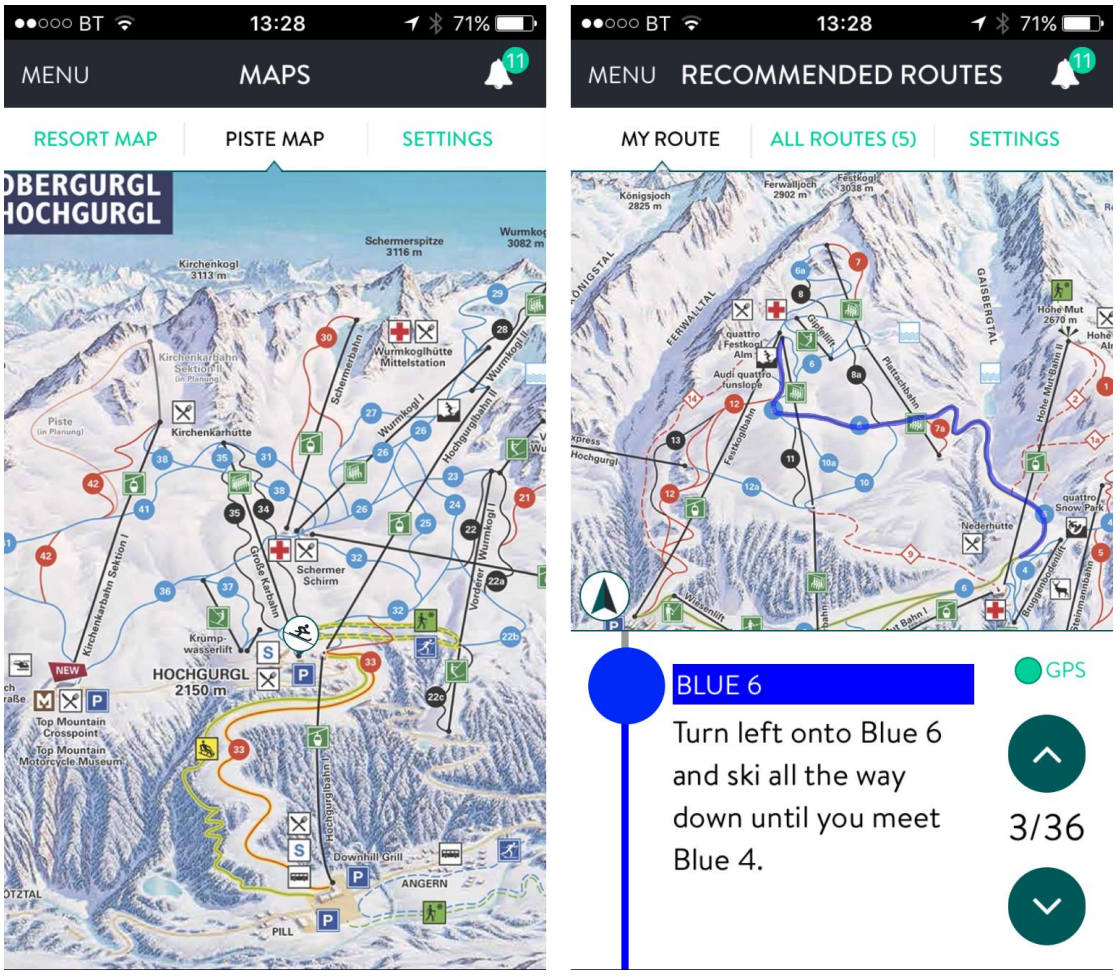


Figure 1: Crystal Ski: Piste Map and Recommended Routes

Since its launch the application has had close to 100,000 downloads on both Android and iOS devices. On any given day, during the ski season, the application can have upwards of 20,000

active users many of which are not actual Crystal Ski customers but users who are using the application to access the information that it provides about the given ski resorts.

Aside from the GPS issues, when it does work the general consensus is that the display of a user's location is quite accurate. The piste maps are displayed on a per resort basis allowing the user to see the map that is most relevant to them. The user can scroll, zoom and pan across the map, allowing them to easily see points of interest that have been marked. Although the application can show the user location, their friends' location (this is done by uploading, at regular intervals, the user's and their friends locations to a server that can then be accessed by the application), points of interest it does not show them how to get there. For skiers unfamiliar with the resort, they may have difficulty traversing it and trying to find the points of interest or their friends from where they are. The current recommended routes system in the CMS could be utilised and further extended to create a point-to-point route system. Taking the user's current location as the start point and allowing them to select a destination, a shortest path could be calculated and then displayed on the piste map.

2.1.2 The Content Management System

As mentioned in 2.1 there exists a content management system that allows an administrator to create resorts, upload piste maps, create recommended routes, and link coordinates on the piste map to actual latitude and longitude values.

The main page of the CMS, see Figure 2, allows the administrator to upload piste maps, setting the geographical boundaries for the map, and then add Plots and Directions.

When a piste map has been uploaded and its bounds have been set, an image of the geographical map is downloaded to the server from Openmaps.org. This is then combined with existing data to make a geographical map image that shows shading for mountains, and has overlays for known ski runs.

The administrator can now go to the Plots section and link points on the piste map with points on the geographical map, Figure 3. The administrator clicks points on the piste map and then clicks the corresponding points on the geographical map. The CMS allows the administrator to check if enough points have been added by testing the location conversion system to see if the result is accurate enough. Obviously this is a subjective result and it does not mean that the result is correct.

The other system that an administrator can use is the Directions page, Figure 4. This page allows routes to be put into the system. This is done by clicking a start point on the piste map and then clicking intermediary points so as to align the route along the correct run or lift. Once the data has been entered, runs and pistes can be selected to make up a recommended route.

The information that is created by both these systems is then served to the user via an API. When the user downloads a resort to their device, it downloads the piste map, a file containing the Plot information (used for calculating the user's location on the map, 2.3.1.1), and the recommended routes information (this will be used to calculate the shortest path across the piste, 2.3.2).

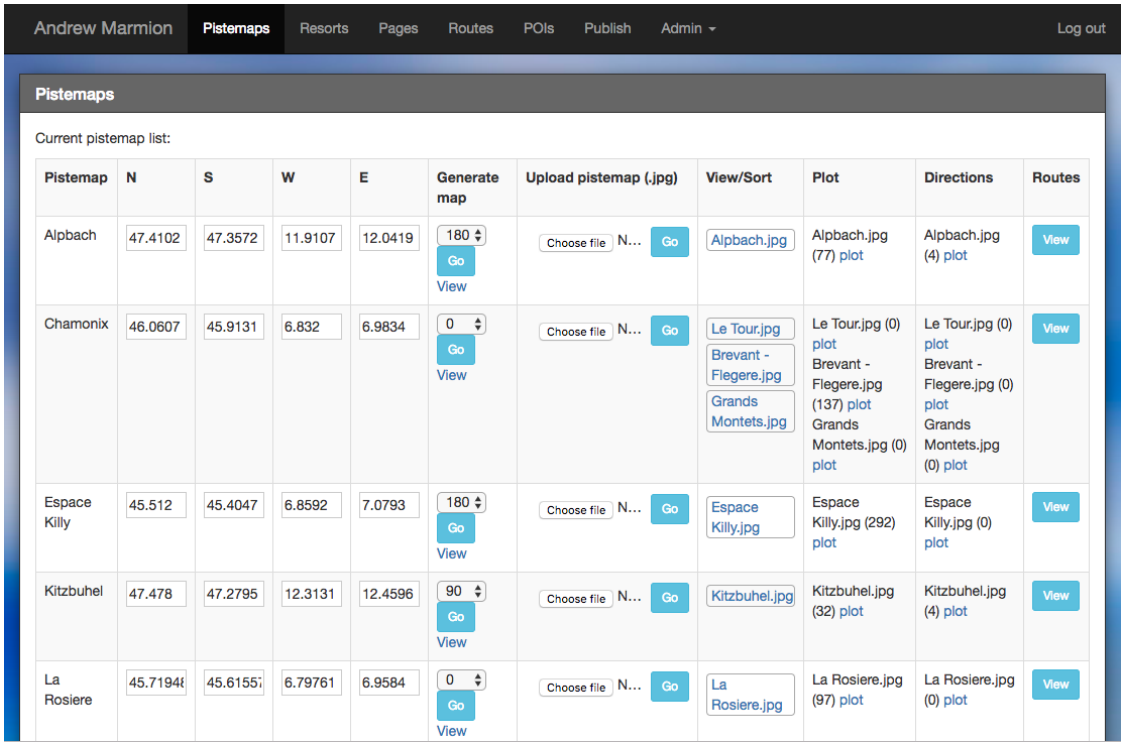


Figure 2: Main Page of the CMS: allows access to Plots and Directions

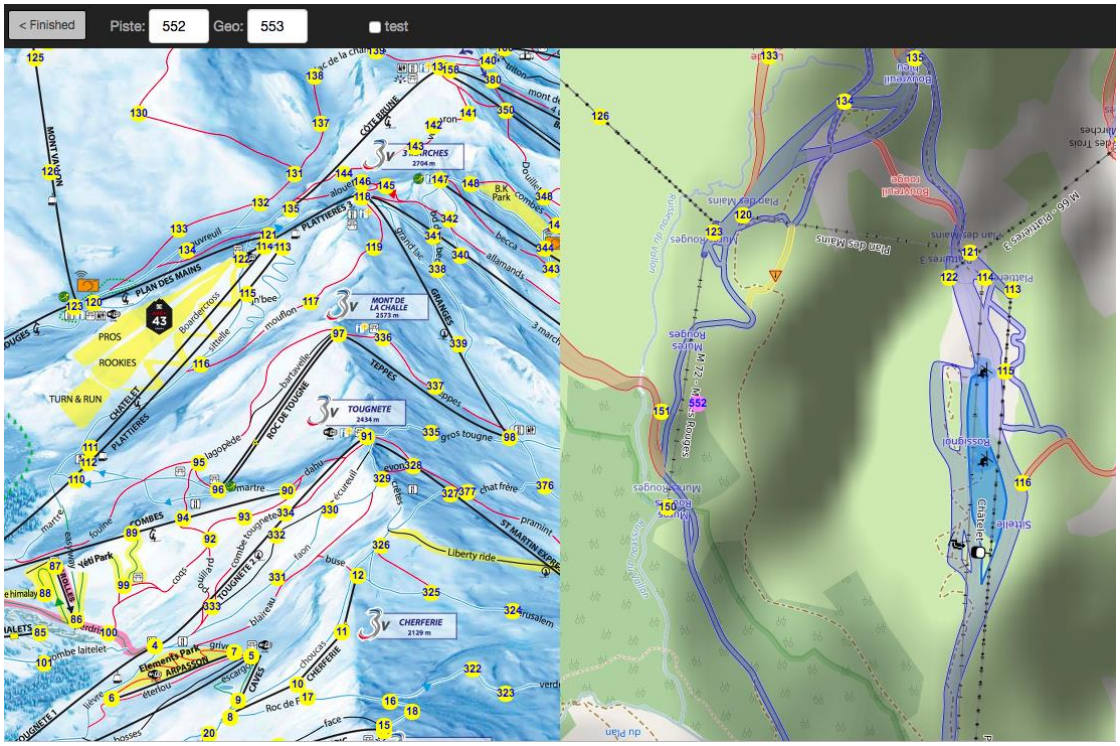


Figure 3: Plots: plotting sets of points on both the piste and geographical map

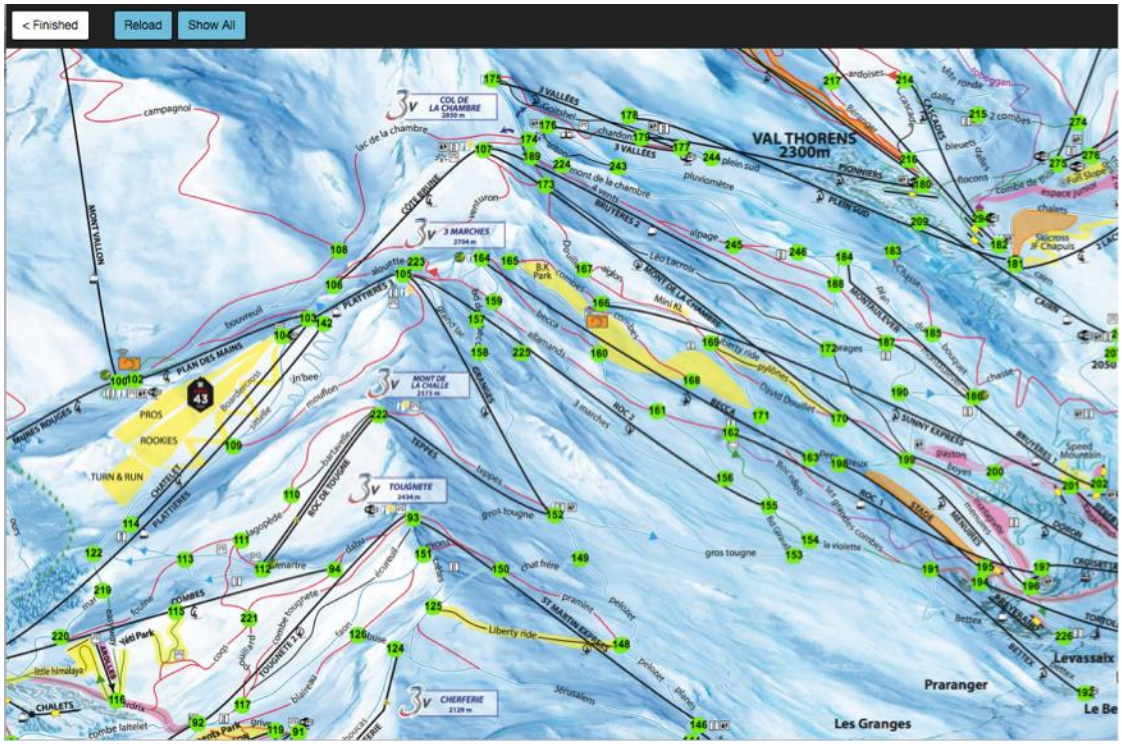


Figure 4: Directions: plotting routes across the piste map

2.1.3 Current Application Summary

The current application has many useful features that will be carried over into its next iteration. The application currently allows users to download resort information to the device.

This has the benefit that it keeps the application size small and means that Crystal Ski does not have to produce a new application every time they wish to update the resort information..However, there is a slight problem with the current system. Currently the only way for a user to get the latest information is to remove the resort from the application and then re-download it. This does not make for a good user experience. It would be better if the device would automatically check for new versions of the downloaded resorts and then prompt the user to update. The flip side to this would be to remove the interaction with the CMS and include every resort inside the application. This would lead to large file sizes and if any updates occurred to the underlying data a new application would have to be created and go through the app store approval process, which would take time and money. Having the CMS allows Crystal Ski to make updates to the resorts in real time and saves them money, as they do not need to pay a developer to make the changes to the application. As the application does not come with any resorts on it when it is first downloaded, if the user forgets to download the resort before they are out of range of a data connection, then the application is un usable as the user will not progress pass the initial login screen. However, once the user has downloaded the resort to the device it is usable without a data connection. This makes it very useful on ski-slopes, as data connectivity can be difficult to obtain.

2.2 Related Applications

2.2.1 Location

This section will deal with the different applications, which currently exist which map a user's location on to a piste map.

2.2.1.1 LAAX

One of the first to offer accurate geo-location on a piste map was the LAAX[1] ski resort in Switzerland. Their solution was to send a local mountaineer out on to the hills and record the coordinates of specific points on the piste map so that they made a grid; these are then used to locate a user on the piste map. This was done in 2012, however it has not been possible to find the webpage that shows the user's location on the map due to their website having been updated since then.

In December 2015 LAAX launched a mobile application. This application manages the users booking for the LAAX resort, much like the Crystal Ski application does for all the resorts that they manage. The application has gone through several versions and they are now on version 4.1.0. In February of this year they moved away from using their own mapping system to using Google Maps. The application is frequently updated and they release winter and summer versions of the application, meaning that it is only possible to see the summer version during

the summer months and winter version during the winter months. Figure 5 shows the mountain, with the ski lifts clearly visible, the highlighted route is a mountain bike trail.

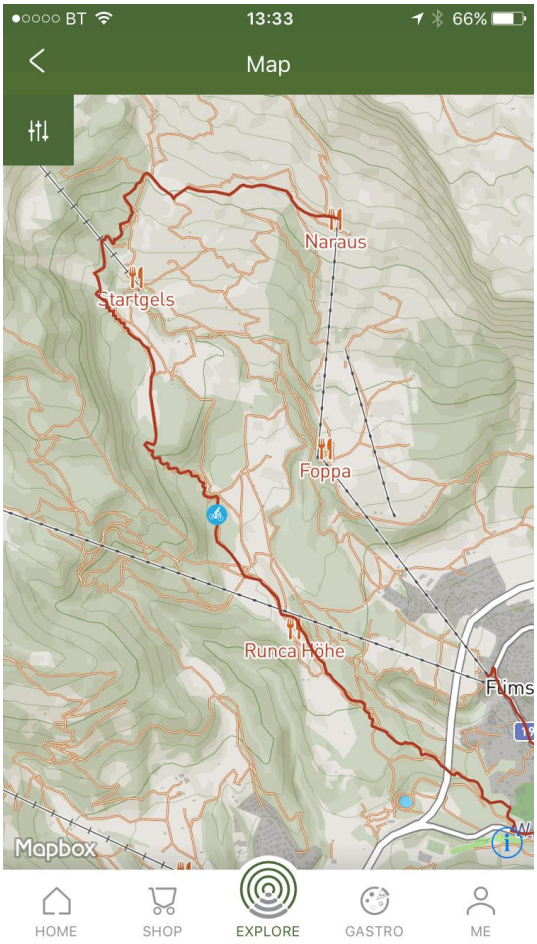


Figure 5: LAAX Inside: View of mountain bike trail

The LAAX website and application offers many interesting features. The idea of mapping the piste map to geographic points is very similar to how it was done in the original Crystal Ski application, however by being on location it would be possible to get a more accurate result. It would be interesting to see the differences between this approach and their now current approach using Google Maps but as it is not possible to download older versions from the app stores they cannot be compared. No reason was given for their move to Google Maps, but it can be surmised that the improvements in Google Maps' accuracy and imaging between then and now may have influenced their decision.

The LAAX app offers the ability to check the status of the lifts, the snow conditions, and even has access to webcams. These are all great features that could be added to the current Crystal Ski application, however as the Crystal Ski application is dealing with hundreds of different resorts, each with different levels of technology and cellular coverage, implementing these

features would be challenging, as only some resorts would have access to all three. The application also offers the ability to track your ski runs and these are then added to a leader board allowing the user to compare themselves to other skiers at the resort. Again, this is a nice feature that could be implemented in the Crystal Ski application, as it could be possible to track a user's location as they ski across the slopes. Though, it is worth noting, a leader board would have to be resort dependent and would require further infrastructure investment as a backend would be required to host the leader board.

2.2.1.2 Maprika

Maprika[2] is a mobile application that allows a user to create a map with location awareness. These maps are then shareable and can be downloaded by any other user. Though only the creator of the map can edit it. Browsing the online repository it can be seen that many different areas have been mapped, from mountain ranges to university campuses.

Using the application is quite intuitive: a user uploads a photo of the map, sets the geographic area that the map covers, they set points on both the photo and the geographical map. Figure 6 shows how a user inputs both sets of points.

Maprika's system is very similar to the implementation that was used in the original Crystal Ski application. Though because the map is being plotted on a small mobile device, although it has an intuitive interface, it can be tricky to use. Once a map has been created and uploaded only the creator can edit the map.

The ability of a user to create a map is a useful feature, though crowdsourcing the locations on a piste map would allow for greater accuracy. By asking users of the application to check if specific locations are in the correct place a more accurate and detailed piste map could be created. This could be built into the Crystal Ski application, however it would require some form of checking as users could enter inaccurate locations. It would also require a backend service to manage and administer the updates to the application.

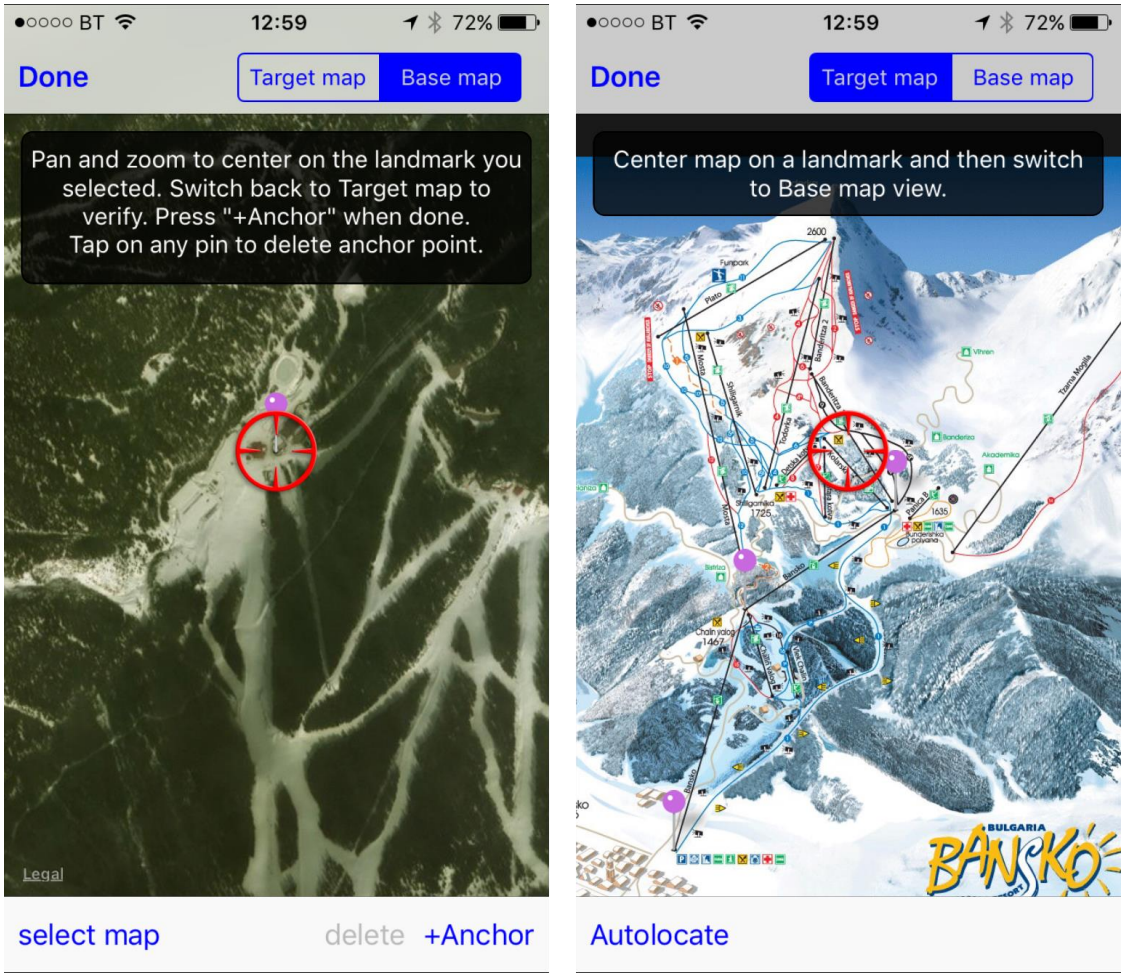


Figure 6: Maprika: entering points on both maps

2.2.1.3 FATMAP

FATMAP [4] is another application that you can use to find your location on a map. The application does not use a standard piste map but rather their system is built upon a 3D model of the area. They utilise images and data from Bluesky International Ltd and Getmapping PLC: companies that specialise in mapping terrain and aerial imagery. These companies use sophisticated techniques, such as LiDAR [5] to create accurate 3D mappings of an area. This solution does provide phenomenal results as can be seen in Figure 7.

FATMAP's solution uses 3D renderings of the area, which provides a very accurate map. The application allows the user to pan and zoom across the map, tapping on a marked run brings up information about that run, gradient information as well as avalanche zones are also available for viewing. Maps are available for use if there is a cellular signal but can be purchased if the skiing will be taking place in an area that does not have reception. Similar to LAAX (2.2.1.1) the application also offers the ability for the user to track their run.

Being able to get detailed information about the runs would be a useful feature to have, as it would allow users to make a more informed decision on whether they should or should not

attempt a particular run. A lot of the features that FATMAP have implemented require a large amount of data, which is currently not available to the general public. This makes implementing their solutions difficult without a significant budget.

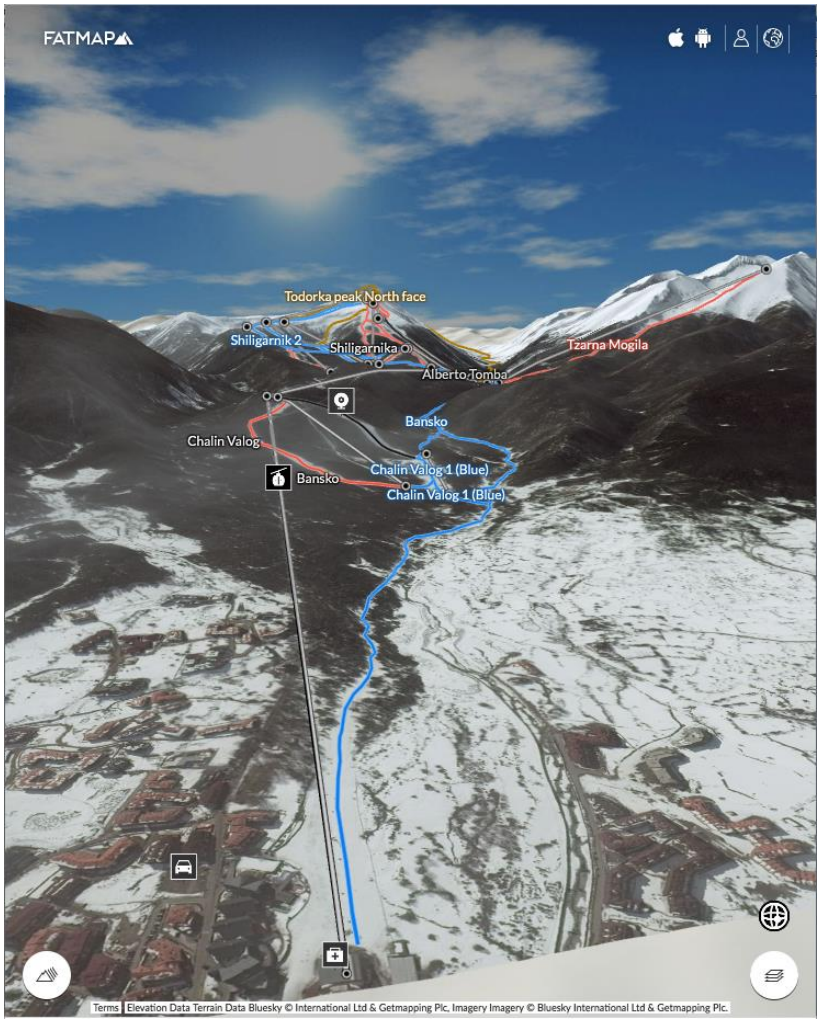


Figure 7: FATMAP: a hybrid piste map

2.2.1.4 SATSKI

An interesting piste location system to note is SATSKI [3], which the owner, Jean-Claude Baumgartner, was arrested and jailed for fraud in 2012. SATSKI, originally started as small car GPS devices that had a custom loader installed on them. Users could then add piste maps by way of an SD card. They then moved on to mobile applications. The location system in SATSKI was actually designed by Paul Burrows and the current location algorithm in the Crystal Ski application takes inspiration from this earlier work.

The SATSKI application also tracked a user's progress as they skied across the pistes. This progress could then be exported and viewed on Google Earth by way of using a KML[1] file. These exports could then be converted into interesting visual displays of the day's skiing.

As the device would be using the GPS signal to locate the user on the piste map it would not be too difficult to extend this functionality to record the data on the device for playback at a future date. The difficult part would be the playing back of this data, currently there exists no native way to view a KML file on an Android or iOS device. However, exporting and using the data with Google Earth would be a possible solution.

2.2.1.5 Location Application Summary

It would appear that the majority of applications use/used a system of linking points on the piste map to points on the geographical map. The applications boast many great features that skiers or mountain users would appreciate. Unfortunately the timescale and scope of the project does not allow for the implementation for many of them.

There has been much success with linking points on the two maps and using the underlying data set to find a user's location on the piste map, and it would seem that this is a good solution to the problem of plotting a user's location on a piste map.

2.2.2 Routes

Directions, getting from point A to point B, have become ubiquitous in our society. There are many options to choose from when it comes to routing. This section will deal with the different applications that provide routing solutions.

2.2.2.1 Microsoft AutoRoute

One of the first successful applications was Microsoft AutoRoute. This was a desktop application that allowed users to plan routes. In the days before mobiles and always on Internet connections this application really stood out from the crowd with its level of accuracy and detail.

Microsoft AutoRoute is no longer in production as routing applications have moved to the Internet rather than exist as a desktop application. Roads are always changing and having a desktop application would need constant updating, which would cause user frustration. By having the solution on the Internet companies that provide routing services can keep them updated in a timely manner.

[1] Keyhole Markup Language (KML) is a type of XML that is used to display geographic data.

Microsoft AutoRoute was a massive application and the complexity of the underlying data structure will be beyond anything that is required of a piste map. Though it would be able to find a shortest or fastest route between two towns, it would not be able to find the route across a piste map, as that particular data set was missing.

2.2.2.2 Standalone SatNav Systems

Next came SatNav systems for cars: such as TomTom and Garmin. These systems utilised maps that came preloaded on the device and using trilateration from GPS satellites were/are able to show your position on the device and use that as a starting point for route planning. Figure 8 shows a route between two points on a TomTom SatNav system, highlighting its dynamic nature showing road works and traffic congestion.

Figure 8: TomTom: displaying a dynamic route with road works and congestion.

The SatNav systems are very useful if you are planning to travel by road, unfortunately they do not have pistes and lifts in their system, thus it is not possible to plot a route with them across a piste map. Modern SatNav systems can receive updates that will show possible delays to the route, adjusting accordingly, while it would be beneficial to implement that feature in the application allowing users to know which lifts are busy or not functioning, to do this would require the user to have an internet connection (which is not always possible on a ski slope) and for the resorts to update an online database, which the device could then access. SatNav systems can also have points of interest located on the map, allowing the user to find out information about them from its built in database.

2.2.2.3 Mobile Devices

The logical extension of the SatNav system was to put it inside mobile devices; augmented with cellular tower information, WIFI, and bluetooth locations which can be accurately found and displayed on a map, allowing you to plan routes from that information. Figure 9 shows a

dynamic route being displayed in the Google Maps application. These routes are dynamic as they are calculated on current information, such as road closures and traffic jams.

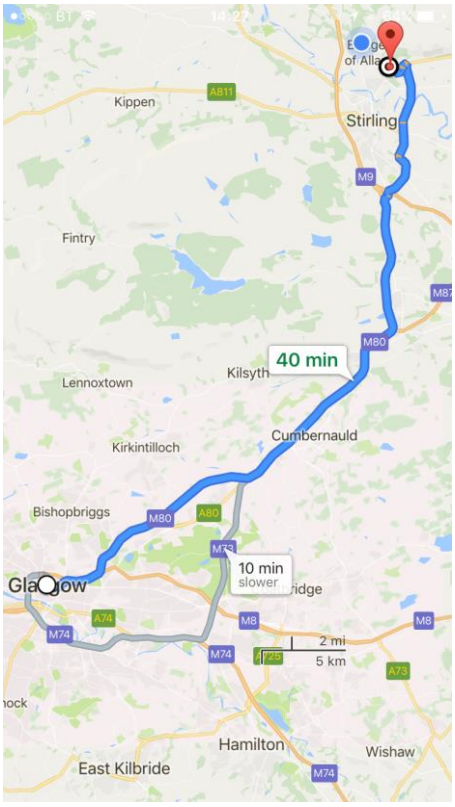


Figure 9: Google Maps: displaying a route from Stirling to Glasgow with alternative

2.2.2.3.1 Summary

The mapping application on mobile devices is very sophisticated; they contain dynamic routing and also show points of interest. One of their major drawbacks is that they require a constant Internet connection to work, though Google Maps now allows you to download areas of maps so that they can be used offline.

2.2.2.4 Routes Application Summary

Currently there appears to be no way of finding a route across a piste map. The current technologies have concentrated on roads and have not included piste maps.

All of these routing systems are based upon a data structure called a graph. A graph is made up of edges and vertices. The vertices are connected by the edges. These edges can be directional or bi-directional, they can also have weights and other attributes associated with them. In a road system, the roads are considered the edges and the intersections are the vertices (or nodes). Some roads are one way, they have speed limits, and different types: A roads, B roads, and motorways. Companies such as Google and TomTom construct graphs of these road networks [6]. Then, using a search algorithm [7][8] they can compute the route between the

start and the end. The algorithm that they use to find the route depends on the size of the graph that they have. The larger and more complicated the graph, the faster and more complex the algorithm becomes.

Bing, Microsoft's online search and mapping arm, previously used a custom version of Dijkstra's algorithm[7][8] but changed that to Customizable Route Planning (CRP) which had been developed by The Microsoft Research Team. This new algorithm is significantly faster than their previous one: *'we drastically improved query speedups relative to Dijkstra from less than 60 to more than 3000'*.

Traversing a graph to find the shortest path is definitely something that many agencies are interested in. Adding route planning to the application is a logical step as users have become accustomed to having that functionality in their day-to-day lives, clearly they would want it on the ski slope.

Adding points of interest to the piste map would also be a nice feature to have. For example, these could be locations of restaurants with their opening times; by tapping on them the information could be displayed in a popup.

2.3 Algorithm Review

This section will deal with the different algorithms in use for calculating location and routing.

2.3.1 Algorithms for location conversion

Piste maps are not flat 2D overhead maps but are a 3D representation of the actual location, usually viewed from the North. It is not a direct one-to-one mapping from one coordinate system to another. If we consider a grid of triangles, whose vertices are the plots from 0, then these triangles are stretched, skewed and rotated from one map to the other.

2.3.1.1 The Current Location Implementation

Paul Burrows, CTO of Dogfish, developed the current location system. This system is used in the original Crystal Ski application and in the CMS. It works on the basis of comparing triangles made by the latitude and longitude of points (geometric triangle) with triangles made by the corresponding x and y values for those points (piste triangle). These points from the plot map described in 0. A user's location is fed into the algorithm and it searches to find which geometric triangle is the best fit for the user's location. Once the best geometric triangle has been found, this is defined to be the smallest triangle; it is then compared with its piste triangle. Using similarity it maps the user's location in the geometric triangle to a point inside the piste triangle. This process involves comparing the location of the user inside the geometric triangle to its vertices and then mapping this comparison on to the same triangle on

the piste map. This resultant point is then used as the user's location. If the user is not inside a triangle a similar but slightly different method is used to calculate the user's location. This alternative method gives an approximate result, but it is not highly accurate and thus cannot be relied on.

Although the current method works well and users have lauded the functionality, similarity is better used on triangles that do not change shape. Similar triangles are defined as having the same angles and sides that are in the same ratio. Unfortunately points that are close together on the piste map may actually be far apart on the geographic map. From a mathematical standpoint similarity is perhaps not the best solution.

As the piste map and the geographic map use a similar coordinate system, the Cartesian plane, it would be prudent to look at algorithms that allow transformations on this plane.

2.3.1.2 Spatial Transformations

Spatial transformations deal with mapping from one coordinate system to another. This would be a good solution to use if all piste maps used the same coordinate system. Unfortunately they do not as many piste maps are hand drawn.

2.3.1.3 Euclidean Transformations

Euclidean transformations are taught to most people at secondary school. These transformations are:

- Rotations
- Translations
- Enlargements
- Reflections

Rotations deal with rotating the shape about a centre of rotation. Translations deal with moving the shape across the coordinate plane. Enlargements deal with enlarging the shape about a centre of enlargement. Reflections reflect the image across a line of reflection.

Although these four transformations can change the position, size and look of the shape after they have been performed they do not change the shape. The shape is still similar to the starting shape: i.e. the internal angles are the same and the lengths of the sides are still in the same ratio. Unfortunately a Euclidean transformation would not work in this case.

2.3.1.4 Affine Transformations

Affine transformations[10] are similar to Euclidean transformations in so far as they allow the same four transformations, but it adds an additional one of shear. For a transformation to be affine it must preserve the following properties:

- parallelism: lines that are parallel remain parallel
- collinearity: collinear points on a line remain collinear
- length ratios: the distance between points on a line remain in the same ratio

All triangles are considered to be affine [11]: '*Given two noncollinear triples of points, PQR and $P'Q'R'$ there is a unique affine transformation T such that $TP = P'$, $TQ = Q'$ and $TR = R'$.'* that means any triangle can be transformed into any other triangle. To perform an affine transformation a transformation matrix has to be calculated based on the coordinates of the points on the geographic map and the piste map. By taking each point in turn, a system of 6 linear equations is constructed and the solution is the transformation matrix.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Equation 1: The transformation matrix for calculating an affine transformation

This transformation matrix can then be used to convert any point within the original shape to a point within the new shape.

This method seems promising and it would be worth pursuing further.

2.3.2 Algorithms for finding the shortest path

Our graph of the piste map has some special criteria. Firstly, it must be directional as you cannot ski up a ski slope, you can only go down. There must be a way to differentiate between the different types of runs on the graph, there are runs for beginners to expert, not all skiers will want to go down a black run, thus the graph should default to choosing the easier run. The largest graph that search is to be performed across is quite sparse with a only 300 vertices and fewer than 700 edges, this gives a graph density of 0.0078[2].

There are many algorithms [23] that can find the shortest path across a graph. It would seem prudent to start with the least complex and build from there.

[2] Graph density [9] is defined as $e/v(v-1)$ where e is the number of edges and v is the number of vertices.

2.3.2.1 Breadth First Search

A breadth first search (BFS) approach will always return the shortest path across a graph. However, as a BFS marks vertices as visited, it will not return the shortest path in a weighted graph, as it will have marked vertices as visited and will not check to see if there is a better route.

2.3.2.2 Dijkstra

Dijkstra's algorithm [12] provides a solution for the Single Source Shortest Path problem. In Dijkstra's original implementation it did not use a priority queue but it has since been modified to use a priority queue (min-heap) based on a binary heap [15] and on Fibonacci heap [16]. Utilising these methods it is possible to speed up the search process from $O(|V|^2)$ to $O\big((|V| + |E|)\log|V|\big)$ [15] and $O(|E| + |V|\log|V|)$ [16] respectively, where V is the number of vertices and E is the number of edges in the graph. The priority queue is a min-heap allowing those vertices that are closest to rise to the top, meaning that they will be looked at first. This is what is termed a greedy approach. Dijkstra's will return a shortest path tree for the starting vertex but it can be stopped earlier once the destination vertex has been found.

Dijkstra's algorithm works in the following way:

1. For every vertex set the distance value to infinity. For the starting vertex set its distance to zero. Add all vertices to the priority queue.
2. The starting vertex is popped from the queue. This is the current vertex (A).
3. For each edge there is a destination (B) vertex. Check that the current vertex's distance + the weight to the destination vertex is less than the distance to the destination vertex (B) i.e. A.distance + B.weight <= B.distance. If this new distance is less than the distance to the destination vertex, update the destination vertex's distance and set its previous vertex to the current vertex i.e. B.distance = A.distance + B.weight and B.previous = A. By storing the previous this allows the path to be constructed.
4. Update the heap.
5. Repeat the above process for each edge on the current vertex.
6. Once all edges have been explored selected a new vertex from the priority queue, repeat steps 3 to 5.
7. Continue until the destination vertex is the target vertex, or until the priority queue is empty.

2.3.2.3 A*

The A* algorithm [13][14] is similar to Dijkstra's, 2.3.2.2, however it attempts to guide the search by way of heuristics. Instead of just adding the weight of the edge and the current distance together, it adds the result of a function that estimates the distance to the target vertex. The difficulty in utilising the A* algorithm is in the construction of this function. A poor function will not improve the solution but a good function will mean that fewer nodes are visited and a faster search is performed.

2.3.2.4 Bellman-Ford (Moore)

The Bellman [17], Ford [18] and Moore [19] algorithm, more commonly known as the Bellman-Ford algorithm also solves the Single Source Shortest Path problem. It is similar to the implementation of Dijkstra's algorithm previously described above, 2.3.2.3, however rather than using a priority queue, it iterates across all the vertices looking at the edges and updating the list of previous vertices. Although the Bellman-Ford algorithm is slower than Dijkstra's, $O(|V||E|)$, it is able to deal with graphs where some of the weights of the edges are negative and it can uncover negative cycles [23].

2.3.2.5 Floyd-Warshall

The Floyd-Warshall [20][21] algorithm solves the All Pairs Shortest Path problem. This problem finds the shortest path between all pairs of vertices. Although slower, running in $O(V^3)$ time, it does calculate all the shortest paths and would allow fast look up times if the graph was pre-processed.

2.3.2.6 Johnson

Johnson's algorithm [22] is another solutions to the All Pairs Shortest Path problem. It is able to deal with negative weights by first reweighting the graph using the Bellman-Ford algorithm, 2.3.2.4, and then using Dijkstra's algorithm to compute the distances between each pair of vertices on the reweighted graph. As Johnson's algorithm finds the All Pairs Shortest Paths then it could be suitable for pre-processing and creating a look-up table of all the shortest paths.

2.3.3 Algorithm Summary

2.3.3.1 Location

Looking at the different ways to convert one triangle into another it is clear that an affine transformation (see 2.3.1.4) is the most suitable solution to the problem. As all triangles are affine, and they can easily be transformed into each other it makes sense to look at this as a possible solution.

2.3.3.2 Shortest Path

Dijkstra's algorithm seems like the most likely candidate for solving our problem. As the graphs will not be significantly large and are sparse, no more than 300 vertices with less than 1000 edges, we can rule out pre-processing. Thus the Floyd-Warshall and Johnson algorithms should not be considered. The A* algorithm and its use of heuristic functions add extra complexity without much additional benefit. If the graph was more dense, then it would be prudent to consider this as an additional option. As the graph will not have negative weights the Bellman-Ford algorithm's ability to handle them would not be needed. Another reason to avoid pre-processing is that it would be possible to re-weight specific routes depending on user preference, having a pre-processed graph would mean that either all pre-processing would have to be done on the device (taking in to account user preferences) or the pre-processing could be done server side and not allow the user to input their preferences.

2.4 Mobile platforms

As the smartphone market has developed, so too have the solutions that can be used for developing applications. Initially developers had to develop natively but over the last 10 years other solutions have become available. Developers have a wide range of solutions to choose from.

2.4.1 Web, Cross Platform and Hybrid Solutions

2.4.1.1 Web

Websites today can utilise a user's location and perform a myriad of functions that applications can do. It is easy to make a website so that it is easily usable by different resolutions. It is also easy to update and code for, as the developer doesn't have to rely on the user having the latest version of the app to use the latest features. For the user to get the latest version, all the developer has to do is update the website and every one that has an Internet connection can access it, and with this lies one of Web's biggest problems. To access the Web the user must have an active Internet connection either from Wi-Fi or from a cellular network. Unfortunately, ski resorts tend to be in areas where Internet connectivity is a problem. Secondly, Web applications cannot access any of the phone's internal systems meaning that for all calculations and functions an active Internet connection is required.

2.4.1.2 Cross Platform

As noted in 2.1 the existing application was built in Titanium [25]. Titanium is one of several cross platform solutions that developers can utilise to build mobile applications. Using a cross platform solution can be both a time and money saver, as it allows developers to build one application and deploy it to different devices.

Titanium and ReactNative [24] are similar solutions in the fact that they both use JavaScript to build their applications. These applications use an interpreter to access the native SDKs[3] allowing greater functionality. Cross platform solutions allow for fast development and prototyping and this reduction in time can reduce overall development costs. However, there are some problems that occur when using these solutions. The user experience is not quite right, it's not quite the iOS experience and it's not quite the Android experience. This can leave users confused and unreceptive to the application. It can be difficult to marry up the code with the SDK. The Crystal Ski application suffered from issues with its location tracking as Titanium was unable to reliably access the underlying native components. One of the main problems with using a cross platform solution is that speed of execution is sacrificed. The interpreter can only run on one thread, meaning that processes can become blocked waiting for their turn to execute.

2.4.1.3 Hybrid

Hybrid solutions use web technologies such as JavaScript, HTML and CSS in combination with webviews inside a native application. PhoneGap [26] is the predominant hybrid solution. Applications are built with a combination of web technologies; through PhoneGap they are able to access the native APIs. Applications built using PhoneGap can easily be deployed as a website and across different devices with only minor tweaks. Like the Cross Platform solutions they are limited in the processing capabilities as everything is done through a webview. Although webviews have made significant progress since their first introduction they are not as capable as a full browser and lack the power and performance of a full native application.

2.4.2 Native

Native applications, although harder to construct, as it is not possible to code in one language for all applications, have significant advantages over other solutions. They have full access to the native SDK. This removes the limitations that other solutions have. They are heavily

[3] SDK or Software Development Kit is a set of tools that allows a developer to create application for a specific software platform.

documented: many developers are building applications natively that means there are lots of resources available from online courses [31], to forums on Reddit [30], to hundreds of thousands of posts on StackOverflow[28][29].

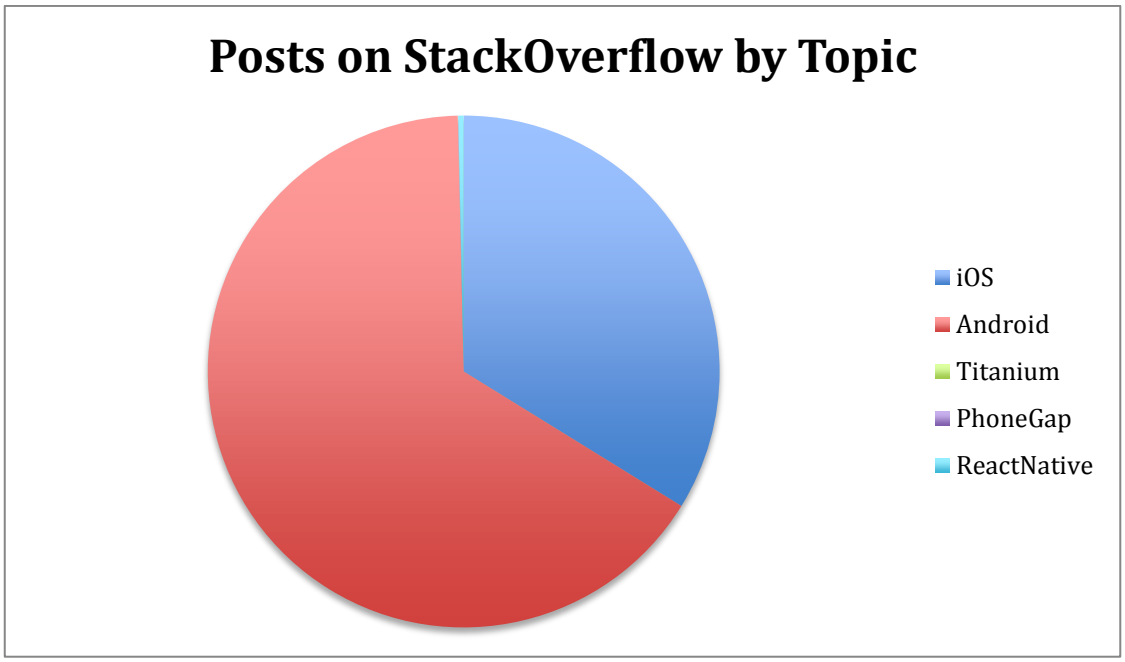


Figure 10: Posts on StackOverflow by Topic[4]

As you can see in Figure 10, Hybrid and Cross Platform technologies do not even make a dent when compared to native in the amount of posts on StackOverflow. The support available for native application development is significantly larger than any of the other solutions.

According to Gartner [27], a technology research company sales of Android and iOS devices in the last quarter of 2016 accounted for 99.6% of all devices. In the first 6 months of 2017 mobile devices used by operating system showed a staggering 64.43% using Android and 31.99% using iOS, making a total market share of 96.42% for these operating systems.

Android applications are natively written in two languages: Java and Kotlin. iOS applications are natively written in two languages as well: Objective-C and Swift. Swift and Kotlin are both relatively new languages. Kotlin [32] was developed by JetBrains in 2010 and then officially released in 2016. Swift [33][34] was developed by Apple, they also started development in 2010 but it was officially released in 2014. Kotlin and Swift have many similarities [35], and with minor changes one can easily convert one to the other.

[4] iOS - 517,272, Android - 1,007,074, ReactNative -120, Titanium - 5167, PhoneGap - 370. 11 July 2017

2.4.3 Mobile Platform Summary

Building an application that can find a user's location on a piste map and help them find the shortest path across said map, would best be done by building two native applications. As the majority of mobile devices use either Android or iOS (2.4.2) it would make sense to target both. Building the application natively would mitigate the problems that Web (2.4.1.1), Cross Platform (2.4.1.2), and Hybrid (2.4.1.3) solutions have. Although Android devices have a greater market share, they tend to be underpowered. Thus restricting their ability by building an application in anything other than native could cause performance problems later on. Though building two different applications will require working within the confines of two different SDKs.

With choosing to build natively, there comes the decision on which languages to use. As Java and Objective-C are very different, and Kotlin and Swift are very similar it would make sense to build the applications in Kotlin and Swift.

3 Iterative and Incremental Development

Dogfish adopts a Hybrid-Agile approach. This chapter discusses the Agile theory and how Dogfish's approach differs from the theory.

3.1 Agile Theory

Many software development companies have moved away from the more traditional approach of the Waterfall [39] model and have started using the Agile [37] model. The Agile model differs significantly from the Waterfall model in that instead of having all the planning and design being completed at the beginning it takes an incremental and iterative approach to the development. The Agile model works in Sprints (3.1.2) and at the end of each Sprint the Team Members (3.1.1.4) should have completed a new feature and have added value to their product. At the end of the Sprint a review takes place where the User Stories (3.1.6.1) are checked and if refinements are required they are updated and added to the upcoming Sprint. Thus creating the iterative and incremental element. The benefits of developing in an Agile way are that customers are able to see frequent updates and do not have to wait, as in the Waterfall method. This allows for feedback to be acted upon immediately and if changes have to be made they can be done early on in the product's development lifecycle.

3.1.1 Key Players

In Agile development there are four groups that each have a specific role and purpose.

1. Stakeholders
2. Product Owner
3. Scrum Master
4. Team Member

3.1.1.1 Stakeholders

A Stakeholder is any person or group that has an interest in the project. This is usually who the product is being built for. These people are usually invited to the end of Sprint review and asked for their feedback.

3.1.1.2 Product Owner

This person is usually a key stakeholder. In the case that the project is being built for an external client, this person would be their representative. They are the person that has the overall vision for the project. They create the User Stories and the Acceptance Criteria 3.1.6.1 on which they will be checked. Their role is very important as they decide which User Stories

will be put in the Sprint and are the point of contact if the Team Members (see 3.1.1.4) have questions.

3.1.1.3 Scrum Master

The Scrum Master is there to help liaise between the Product Owner and the Team Members. Their purpose is to help remove blocks, which are stopping the Team Members from being successful: i.e. by getting missing icons or APIs from the Product Owner. They also take on the role of a coach and help encourage the Team Members.

3.1.1.4 Team Member

Although the Product Owner decides what User Stories are put into a Sprint, Team Members are responsible for deciding how long it will take to complete these items and for choosing how they will complete them, i.e. they make the decisions on the coding solution.

3.1.2 Sprints

A sprint is a collection of user stories that are to be completed over a period of time. They are usually contained within an epic (see 3.1.3). Sprints usually last between one to four weeks, though many development teams opt for shorter one to two week-long sprints. A typical two-week sprint, starts as follows:

The first morning of the sprint a 2-hour meeting takes place where the Product Owner, Scrum Master, and Team Members meet to discuss what will be put into the upcoming Sprint. Once that has been decided the Task Board is then populated.

After the meeting the Team Members are then free to work on the User Stories. For the next eight working days, daily meetings lasting approximately 15 minutes take place (called Scrum meetings or Stand-ups) between the Product Owner, Scrum Master and Team Members. The purpose of these meetings is to state three things; it is not to solve problems. The three things that are stated are: what work has been completed, what work will be worked on today, and what things are preventing work from happening. By keeping on top of possible problems it helps to quell the snowball effect, where problems escalate and become too difficult to manage.

On the penultimate day of the Sprint, a code freeze occurs around midday. This is to give Team Members a chance to fix any bugs that the Quality Assurance Team has found before the Sprint Review on the final day.

The Sprint Review is where all Stakeholders are invited to a demonstration of what has been completed in the Sprint. They are able to ask questions, make suggestions, and see the progress of their product. These suggestions are incorporated into User Stories and will be

implemented in the upcoming Sprint. After the Sprint Review there is a Retrospective, which allows the Team Members to discuss which strategies worked well and how they could make improvements going into the next Sprint. Finally, the Team Members use the remainder of the day to size User Stories from the Backlog (see 3.1.6).

3.1.3 Epics

Epics are considered a large over-arching user story; effectively it is made up of smaller user stories (see 3.1.6.1). Epics can be made up of one or more sprints.

3.1.4 Task Board

The Task Board is a simple concept that can take many forms. Usually it is a physical board that has three columns: To-do, In Progress, and Done (see 3.1.5). User Stories (see 3.1.6.1) are initially placed in the To-do column. Team Members (see 3.1.1.4) who are working on a particular User Story move it to the In Progress column, and when they complete it, move it to the Done column.

3.1.5 Done

It is important to define the definition of Done so that all players understand what it means when a User Story is marked as such; whether it just means that the acceptance criteria has been met and there is testing still to be completed or if everything has been completed and it is ready to ship.

3.1.6 Backlog

The Backlog contains all the User Stories (see 3.1.6.1), in fact there are two Backlogs: The Product Backlog (see 3.1.6.2) and the Sprint Backlog (see 3.1.6.3).

3.1.6.1 User Stories and Acceptance Criteria

User Stories [38] are written in the following form:

> 'As a **<type of user>,** I want to **<do something>,** so that **<some value is created>.**'

Acceptance Criteria are items that must be met so that the User Story can be marked as Done. Only once all Acceptance Criteria are met can the Team Member mark the Story as Done.

User Stories are given a size, this is an estimate of how long they will take to be completed, and Team Members usually log how long it took to complete them. User Stories are traditionally ranked by order of importance, and depending on the project they may be ranked by difficulty. This means that once added to a Sprint, the most important/difficult User Stories are completed first.

3.1.6.2 The Product Backlog

The Product Backlog contains all User Stories: stories that haven't been finalised, stories that have been completed, and stories that are in the current Sprint. Stories that have not been finalised may be quite terse but will be updated by the Product Owner over time so that their Acceptance Criteria are succinct and easy to understand.

3.1.6.3 The Sprint Backlog

The Sprint Backlog contains only the User Stories that are relevant to the current Sprint. Team Members may ask for additional User Stories to be added to the Sprint if they have completed the ones that are there, or if they are blocked on the current User Stories and cannot work on them.

3.2 Dogfish's Implementation

Although Agile development processes are incremental and iterative, Dogfish's approach is not pure Agile. They borrow many of the facets of Agile development, Figure 11, by working in Sprints, using User Stories, having a Backlog, the Task Board and having daily Scrum meetings. However, as they are not building applications for themselves but for their customers there are other facets that they need to take into account. Their clients have a fixed timeline and budget. This means that their client wants to know how long it will take and how much it will actually cost: they are more concerned with having a working product rather than having the best product. This is where their approach mirrors the Waterfall [39] method, by setting the requirements before the project has begun it gives an overall picture of what has to be completed but it can make it difficult to adjust if the Product Owner decides to change the requirements or if the project becomes stalled for any reason.

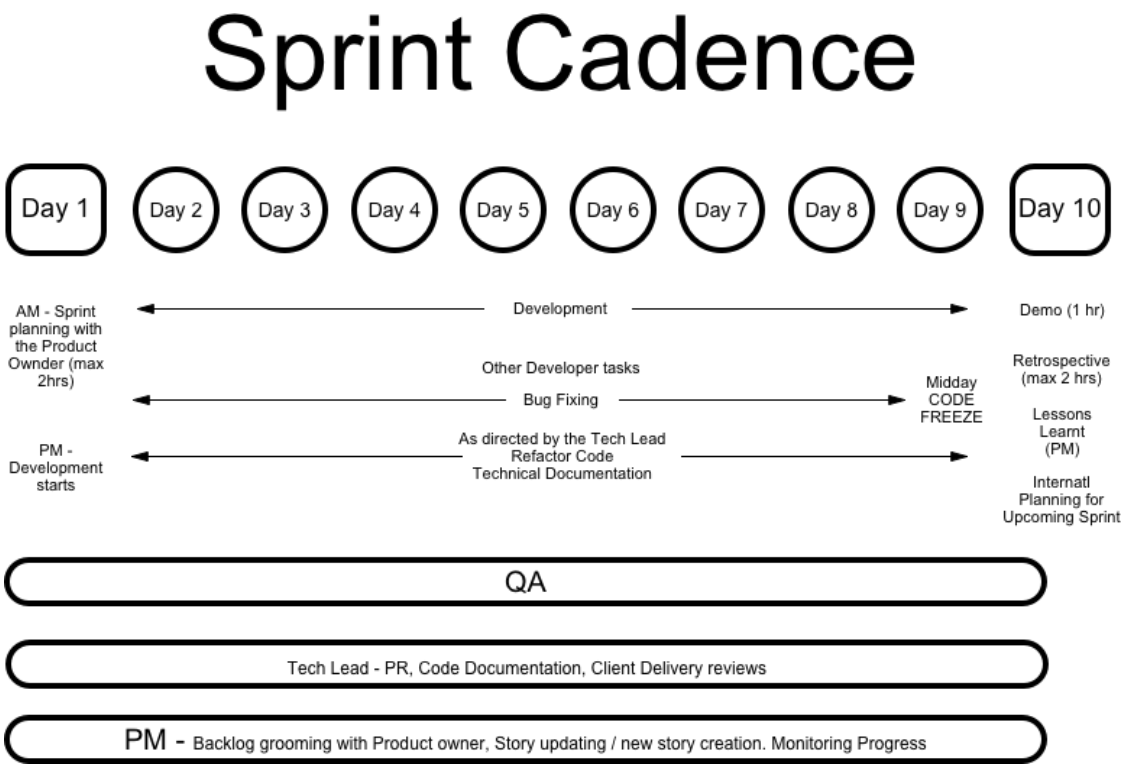


Figure 11: Example of Dogfish's two week Sprint

When it comes to writing User Stories, these are written in conjunction with the Product Owner, a Project Manager (re: Scrum Master) and a Technical Lead (an experienced Senior Developer). All of the User Stories and their Acceptance Criteria are completed before development begins, though they may be refined before they are placed into a Sprint. An estimation of the total time to create the application is given by the Technical Lead, if the Technical Lead gets this estimation wrong (usually by underestimating) it can have significant ramifications for the project. This disparity appears when the User Stories are sized by the Team Members. If the Team Members believe it will take longer to complete the work than is in the original estimate the project risks not meeting the final deadline.

The Task Board (3.1.4) at Dogfish is not a physical item, instead they use JIRA [40] a website that allows online tracking and project management for Agile teams. In Figure 12, it is quite easy to see the three columns that should appear on an Agile Task Board. With this utility it is easy for the Scrum Master and other Team Members to keep track of who is doing what, what is still to be done and what has been completed. This is very useful for those who are working remotely and who would not be able to have access to a physical Task Board.

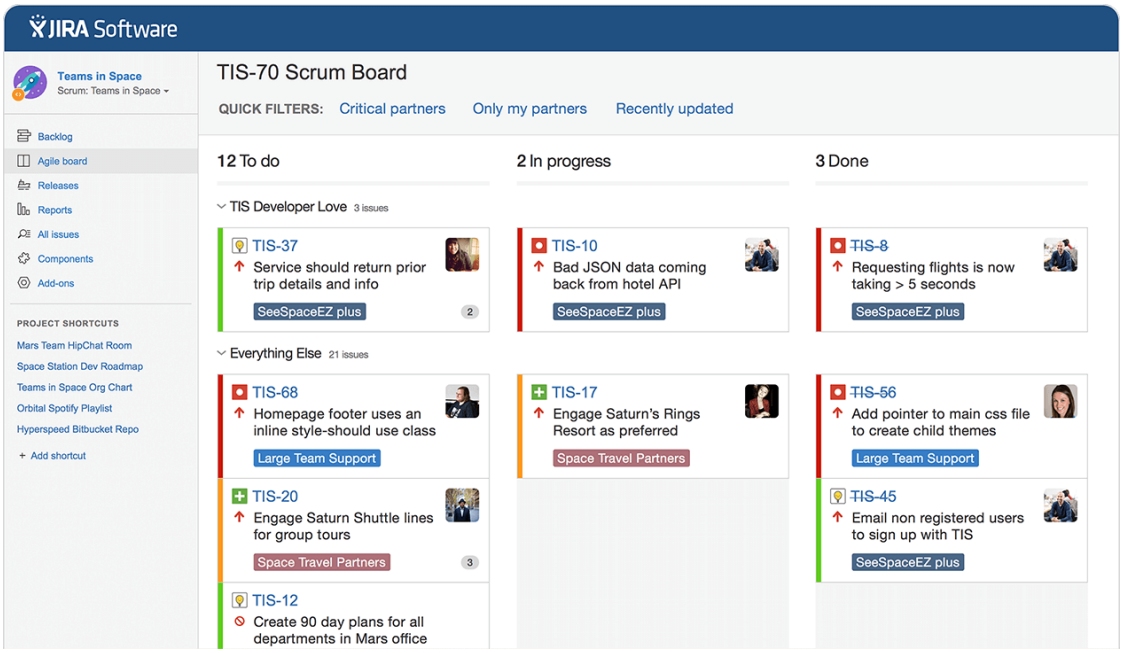


Figure 12: Example of a JIRA Task Board

Quality Assurance (QA) is an import facet of the work done at Dogfish. There is a dedicated team that tests the work that has been completed by the developers. Daily builds of the application are made and are tested against the User Stories' Acceptance Criteria. Once a User Story has been marked as Done the QA process starts. If a build does not match a User Story or a bug exists, then a new User Story is created and it is then added to the current Sprint's backlog with the highest priority.

Dogfish's use of a combination of both Agile and Waterfall methodologies is an attempt at taking the best features from both of these to come up with an overall more successful approach. Waterfall approaches do not allow for any change of direction once the project has started, this can be disastrous if initial designs are no longer tenable. Agile approaches allow for iteration and incremental development, and though Dogfish's implementation is not pure agile it does allow for changes once the requirements have been set. This makes it more flexible and easier to change direction, while maintaining a good overall estimate to the length of the project. However, one area where Dogfish is lacking is that test-driven-development (TDD) is not undertaken. TDD can slow the initial start of a project, as writing tests can be time consuming, it can make the QA process easier, giving more robust results as the underlying code has been tested significantly and thus reducing the number of bugs that may be found. It also stops new code being added breaking existing code, as the tests would still have to pass.

3.3 My implementation

As I am carrying out this project at Dogfish I will be following their work practices. This means that I will be working with the other Team Members in Sprints to complete User Stories based upon the requirements set out in chapter 4. Daily builds will be made and tested against the completed User Stories, with any unmet Acceptance Criteria or bugs being added to the current Sprint's backlog.

4 Requirements Engineering

Before any coding can be undertaken it is necessary to define what the software is required to do: this is what is known as creating the requirements [39]. Having the requirements defined allows for an unambiguous implementation where all stakeholders are aware of what is being created. By discussing the requirements with the client this allows all parties to have a clear understanding of what is to be done. It gives the client a chance to explain what they want and it gives the developers a chance to ask questions and clarify what the client want. Failure to fully understand the client's wants and needs, can lead to an application that does not meet with the client's expectations

4.1 Requirements Elicitation

Dogfish already had a set of User Stories that had been created in conjunction with Crystal Ski. These User Stories and their associated Acceptance Criteria define what they are required to build for their client.

The main requirements for the application that were discussed, and are as follows:

- Target both iOS and Android with separate native applications.
- Convert a user's latitude and longitude to a point on a piste map
- Allow a user to find a route from one location on the piste map to another

During the first meeting with Dogfish it was decided that a subset of User Stories would be created to clarify the exact requirements of the module that would be built. In subsequent meetings an in-depth discussion about these User Stories took place and they were split into three epics (3.1.3): the Location Epic (4.3), the Routing Epic (4.4), and the Application Epic (4.5). Splitting into three epics makes sense as it allowed for each sprint/epic to focus on one particular aspect of the implementation rather than trying to build a complete Android app and then a complete iOS app.

4.2 Use-cases

Use-cases make it very easy to see what a user would like to be able to do and breaking it down in to straightforward tasks. Taking the above epics it is possible to convert them into use-cases, these are shown below in Figure 13. Although the underlying functionality is quite complex, from a user's point of view the application is quite simplistic: and as a result the user cannot do many things in the application, thus keeping the amount of use cases low. As mentioned in the previous sections of this chapter, the User Stories are split into three epics.

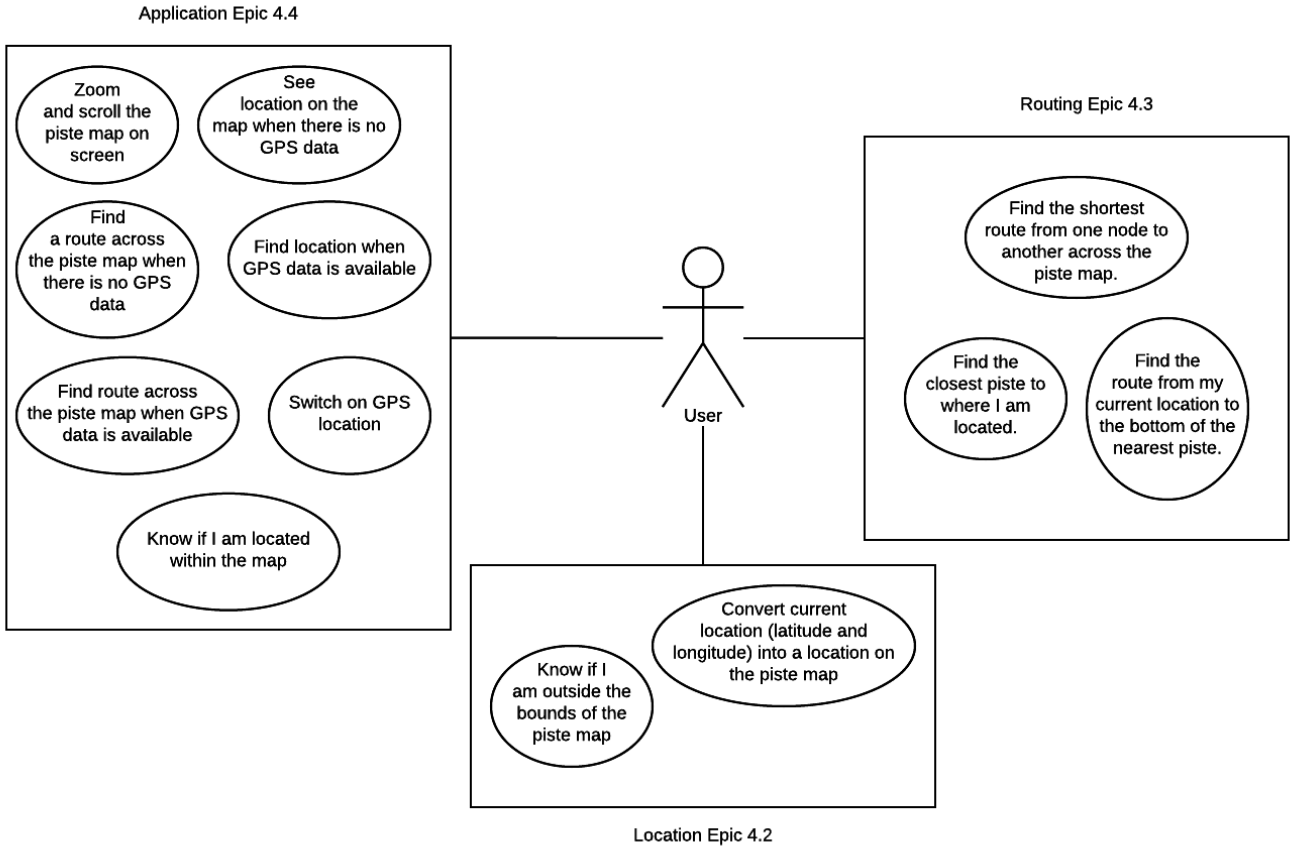


Figure 13: Use cases based upon the User Stories

The main interaction for the user comes from the work that will be done in the Application Epic (4.5) the Location (4.3) and Routing (4.4) Epics. This is the reason why the there is a significant overlap between the Location and Routing Epics and the Application Epic. The only use cases that are not involved in the overlap are the ones involving using the GPS systems of the device as the Location and Routing libraries won't actually require access to the GPS systems and could be used without them. However, the user's location from the GPS system will be fed in to the Location algorithm. The following chapters, 5 and 6, focus on each epic in turn, explaining in more detail the design and implementation.

4.3 The Location Epic

The Location Epic (LE) deals with the construction of the algorithm, which will convert a user's latitude and longitude into an x-y coordinate on the piste map. This epic contains the following user stories.

1. **As a user I want to take my latitude and longitude and convert it to x and y values on the piste map.**

AC1: Get user latitude and longitude from GPS system.

AC2: Convert user's latitude and longitude into pixel values

2. **As a user I want to know if I am outside of the bounds of the piste map**

   AC1: Get user latitude and longitude from GPS system.

   AC2: Confirm whether this is within the bounds of the piste map.

4.4 The Routing Epic

The Routing Epic (RE) deals with the construction of the algorithm that finds the shortest path across the piste map.

1. **As a user I want to be able to find the shortest route from one node to another node across the piste map.**

   AC1: Set starting and ending nodes

   AC2: Return shortest path (based on weights) between the nodes if it exists.

2. **As a user I want to be able to find the closest piste to where I am located.**

   AC1: Finds ordered list of closest intersections of routes.

   AC2: First entry is closest piste.

3. **As a user I want to be able to find the route from my current location to the bottom of the nearest piste.**

   AC1: Finds the piste that is closest to user's location, ignoring lifts.

   AC2: Calculate the closest line-segment within a piste.

   AC3: Calculate the route from the intersection with the line-segment to the bottom of the piste.

   AC4: Return the path from the user's location to the bottom of the nearest piste.

4.5 The Application Epic

The Application Epic (AE) is concerned with deploying and implementing the algorithms in native Android and iOS applications.

1. **As a user I want to be able to see a scrollable and zoomable piste map on the screen.**

   AC1: Scrollable view

   AC2: Zoomable view

AC3: Piste map is loaded and is showing on the screen.

2. **As a user I want to be able to see my location on the piste map when there is no GPS data available.**

   AC1: Scrollable piste map is visible

   AC2: Error message is displayed saying “Unable to use GPS, long press to set your start point”

   AC3: When user long presses location appears on map

3. **As a user I want to be able to find a route across the piste map when there is no GPS data available.**

   AC1: Scrollable piste map is visible

   AC2: Error message is displayed saying “Unable to use GPS, long press to set your start point”

   AC3: When user long presses location appears on map

   AC4: Notification to appear after long press was successful

   AC5: Route if available is displayed on the screen

4. **As a user I want to be able to see my location on the piste map when GPS data is available.**

   AC1: Scrollable piste map is visible

   AC2: User location is displayed on map

5. **As a user I want to be able to find a route across the piste map when GPS is available.**

   AC1: Scrollable piste map is visible

   AC2: User location is displayed on map

   AC3: Notification to appear after long press was successful

   AC4: Route if available is displayed on the screen

6. **As a user I want to be able to switch on GPS location.**

   AC1: At initial app launch display request for location access

7. **As a user I want to know whether I am located within the map**

   AC1: If user is not located within the bounds of the map display error message.

5 Design

This chapter will contain information about the design process.

There exists four main parts to the design of the application:

1. Interaction with the CMS
2. The location algorithm
3. The routing algorithm
4. The deployment in application

5.1 Interaction with the CMS

The showcase mobile applications will not interact directly with the CMS (2.1.2). In fact the showcase mobile applications will not have any knowledge that the CMS exists. The files that that CMS creates will be directly placed inside the showcase mobile application when the developer builds it; Figure 14 shows this interaction. First the files are downloaded from the CMS, next the developer places the files inside the Integrated Development Environment (IDE), in the case of building native applications the IDE for Android is Android Studio and for iOS it is Xcode. The files are then embedded in the application when it is compiled. The reason for this is that it removes the requirement that the mobile device would need an active Internet connection to download the data. Also by providing the files on the device a separate backend that could serve the files is not required to be constructed.

Figure 14: Interaction between CMS and Showcase Application

The CMS creates JSON (JavaScript Object Notation) files that store information about the plots and the routes across the piste map. JSON was chosen as the file format for several reasons. Firstly the CMS is written in JavaScript, this makes dealing with JSON trivial as it is natively supported. Secondly, as the current Crystal Ski application allows for the downloading of resort data to the device this is best served by a RESTful web API. This API is to be updated for the new application. This will allow the final application to be updated when changes occur and it will also lead to smaller application sizes by not having all of the files stored on the device. Thirdly, JSON files can be smaller than other formats, such as XML.

With the possibility of having many piste maps in the application, having a small file format is preferable. Finally parsing JSON will fail faster than parsing XML. This means that if there is an error in the JSON file we will find it faster than if there is an error in the XML file.

The JSON files will need to be accessed by the application so that the information contained within them can be used to place a user on the piste map or help a user find their way across the piste map. The plots and routes files will be added to the application at build time; see Figure 14.

The plots.json file, see Figure 15, will be used to calculate the location, each point inside the plots.json file has a latitude and longitude which is the location of the point on the geographic map, and an x and a y which is the location of the point on the piste map.

```
{
    "pistename.jpg": {
        "1": {
            "lat": 45.394312,
            "lon": 6.56549,
            "x": 1484,
            "y": 1320
        },
        "2": {
            "x": 1479,
            "y": 1309,
            "lat": 45.392049,
            "lon": 6.56828
        },
        "3": {
            "lat": 45.390994,
            "lon": 6.567893,
            "x": 1484,
            "y": 1304
        }
    }
}
```

Figure 15: Example of the plots.json file

The nodes.json file, see Figure 16, contains the nodes and routes that will be used for the underlying graph that will be used to calculate the shortest path across the piste map. Each

node has an x and a y which is it's location on the piste map. The paths array is a list of all the routes that can be reached from this node. Each path has the id of the destination node, the weight, the distance and the duration of the path and an array of points that can be used to draw the route on the piste map.


```
{
  "pistename.jpg": {
    "1": {
      "x": 271,
      "y": 401,
      "paths": [
                {
                "to": "2",
                "type": "",
                "subtype": "green",
                "name": "",
                "points": [
                          {
                          "x": 300,
                          "y": 394
                          },
                          {
                          "x": 335,
                          "y": 389
                          }
                          ],
                "weight": 0.238,
                "dist": 238,
                "duration": 1.48
                }
      ]
    }
  }
}
```


Figure 16: Example of the nodes.json file

Both Android and iOS have ways to serialise and read JSON files but they are not the developer-friendliest ways of dealing with it. For this reason I have opted to use external

libraries for the reading of the JSON files. For Android Klaxon [41] will be used and for iOS will use SwiftyJson [42].

5.2 Location

This section corresponds to the User Stories given in the Location Epic (see 4.3). The Location library will contain all the objects required for the Location algorithm to function. The Location algorithm will take a user's latitude and longitude and return a point on the piste map that corresponds to the user's location.

As each plot has a latitude and longitude coordinate (geo coordinate) and a corresponding x and y coordinate (piste coordinate) it is possible to find the closest geo coordinate to the user's current location. From this it is possible to find all the possible triangles that the user's location lies within. By comparing these triangles it is possible to find a best triangle, this is defined as the smallest triangle that the user is located within. Then by using the geo and piste coordinates calculate an affine transformation that maps the geo coordinates onto the piste coordinates. As all triangles are affine, it will be possible to map any point that is within the geo triangle to a point within the piste triangle. This will allow conversion of a user's geo coordinate to their piste coordinate.

There does exist one scenario that an affine transformation cannot cover, that is when a user is not within any triangle. Currently Dogfish has a solution for this and it was discussed that that creation of a solution for this was optional.

5.2.1 Structure

Figure 17 shows an overview of the structure of the location algorithm. This structure is based on the original that was used in the Titanium built Crystal Ski application. These objects are explained in more detail in section 5.2.3.

The location algorithm works in the following way. Firstly, a Location object (5.2.3.1) is created; this loads the plots.json file into memory and converts its contents into an array of TrianglePoints (5.2.3.3). When a request for a user's location on the piste map is made the TrianglePoints are then ordered by distance from the user. The closest 16 TrianglePoints are used to construct Triangles (5.2.3.4). These Triangles are then checked to see if the user is within them, if the user is contained within the Triangle this is then stored and the smallest Triangle is then chosen as the BestTriangle (5.2.3.5). From the BestTriangle, either the Similar Triangle Method (6.3.2) or the Affine Transformation Method (6.3.3) is used to calculate the location of the user on the piste map. Once a location has been found it is then finally checked to make sure that it is within the bounds of the Resort (5.2.3.2) and a RoutePoint (5.2.3.6) representing the user's location is returned.

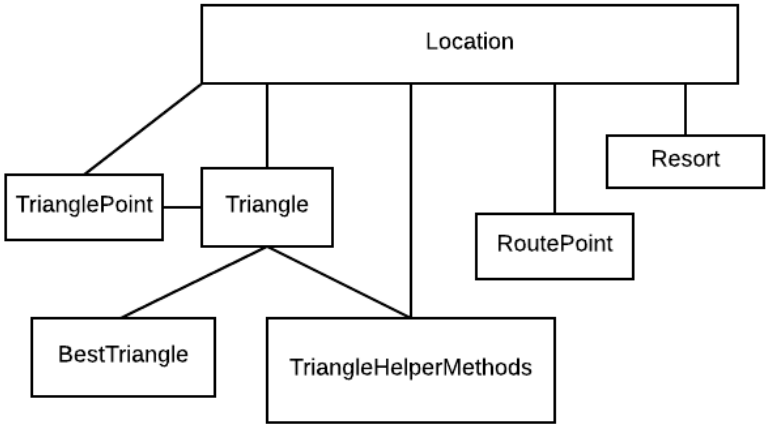


Figure 17: Overview of the main interactions of the Location Algorithm

5.2.2 Example of calculating the user's location using an Affine Transformation

Section 5.2.1 gave a brief overview of how the calculation will be performed and at which stages the different objects will be used; this section will walk through an actual calculation. Figure 18 shows how points on the geographic map (the points on the left denoted by a G) can compare with those on the piste map (the points on the right denoted by a P). Each point is correspondingly numbered: the point G1 on the geographic map is paired with the point P1 on the piste map.

Figure 18: Points on the geographic map (left) vs points on the piste map (right)

For Figure 18 we have the following coordinates, see

Table 1, from this we can construct an algorithm to calculate the user's location on the piste map. To begin we need the coordinates of the user's location. Let's take the user to be at latitude 56.115556 and longitude -3.937311. This location is close to the point G3 as can be seen clearly in Figure 19. To start we first calculate the distance to each point from the user's location and then order them by that distance. This gives the following result: G3, G4, G2, G5,

G6, G1. Taking the closest point to the user's location, this would be G3; we then construct triangles with the other points using G3 as the first point in the triangle. We then check to see if the user's location is within that triangle, comparing which side of the edge of the triangle the user's location is on does this. If all points are within the triangle, the triangle is compared against the other triangles that the user's location has been found to be within. By comparing the total distance from the user's location to the vertices of the triangle, allows us to choose the triangle that has the smallest area. This should give us the most accurate result.

Point	Latitude	Longitude	X	Y
1	56.115388	-3.932419	1577	1002
2	56.115604	-3.934522	1450	1007
3	56.115197	-3.93774	1318	1088
4	56.116584	-3.937011	1312	983
5	56.117685	-3.936882	1300	871
6	56.117015	-3.934393	1446	853

Table 1: Plots for location calculation

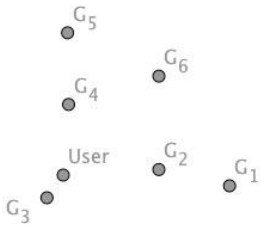


Figure 19: The user's location on the geographic map

For our example we find that the point is in the following triangles: (G3, G4, G2), (G3, G4, G6), (G3, G4, G1), (G3, G2, G5), (G3, G5, G6), and (G3, G5, G1). The smallest triangle that the user's location is in turns out to be (G3, G4, G2). It is quite easy to see this fact from Figure 19. Now that we have a triangle that surrounds the user's location we can calculate the transformation that will map the user's location in the geographic triangle to the piste triangle.

To perform an affine transformation on the user's location we first need to compute the transformation that converts the geographic triangle to the piste triangle. To do this we set up a system of equations to look like this:

$$x = lat\ t_{11} + lon\ t_{12} + t_{13}$$

$$y = lat\ t_{21} + lon\ t_{22} + t_{23}$$

Equation 2: The affine transformation equation

$$1318 = 56.115197t_{11} - 3.937740t_{12} + t_{13}$$

$$1088 = 56.115197t_{21} - 3.937740t_{22} + t_{23}$$

$$1450 = 56.115604t_{11} - 3.934522t_{12} + t_{13}$$

$$1007 = 56.115604t_{21} - 3.934522t_{22} + t_{23}$$

$$1312 = 56.116584t_{11} - 3.937011t_{12} + t_{13}$$

$$983 = 56.116584t_{21} + -3.937011t_{22} + t_{23}$$

Equation 3: The completed equations ready for solving

Solving these equations (Equation 3) is quite trivial and gives a final solution for the transformation that will map this specific geographic triangle to its corresponding piste triangle of:

$$x = -27728.7lat + 44526.3lon + 1732650.4$$

$$y = -66921.9lat - 16706.9lon + 3690636.1$$

Equation 4: The affine transformation

We can now substitute the users latitude and longitude into Equation 4 to give the position of the user's location on the piste map. In this example the user's location is at (1327.1, 1056.8).

5.2.3 Objects

The following objects and classes are used in the calculation of the user's location on the piste map.

5.2.3.1 Location

The Location class is the main workhorse of this algorithm. To create it, it requires a Resort object (5.2.3.2) and a JSON file containing the plots. The plots.json file contains an array of matched points: each x-y value on the piste map is paired with its corresponding latitude and longitude. When this file is loaded, its contents are parsed into an array of TrianglePoints (5.2.3.3). This process of loading the TrianglePoints only needs to occur once since they are then maintained in memory.

5.2.3.2 Resort

The Resort object contains information about geographic boundaries, the furthest points North, South, East and West. It also contains the height and the width of the geographic map.

5.2.3.3 TrianglePoint

The TrianglePoint object contains three coordinate pairs: a latitude and longitude, which relates to the point's position on the geographic map; an x and y, which relates to the location of the point on the piste map; and a gx and gy, this coordinate pair is an adjusted latitude and longitude. These latitude and longitude values are converted into pixel values on the geographic map, this allows for more accurate calculations to be performed such as the distance and internal angles. Finally the TrianglePoint contains a distance value, this value is the distance from the current user. It is used to order the TrianglePoints and when finding the BestTriangle.

5.2.3.4 Triangle

The Triangle contains three points, the length of each side, the sizes of the angles, and the maximum and minimum angles inside the triangle. The maximum angle is used to determine whether or not the triangle should be considered as a BestTriangle, Dogfish have found, through trial and error, that triangles with large obtuse angles produce inaccurate results in the similar triangle method. When a Triangle is created, all the internal angles and side lengths are computed, these values are then used in the Similar Triangle Method (see section 6.3.2).

5.2.3.5 BestTriangle

This is the wrapper object that contains the smallest triangle that has been found. It has three TrianglePoints, one for each of the BestTriangle's vertices, and it has the total distance from the user's location to the vertices allowing for comparisons of the triangle.

5.2.3.6 RoutePoint

The RoutePoint contains the x and y values of the user's location on the piste map, it also contains a Boolean value that states whether the point is contained within the bounds of the map.

5.2.3.7 TriangleHelperMethods

A class of static methods that help calculate angles and distances

5.3 Routing

This section corresponds to the User Stories given in the Routing Epic (see 4.4). The Routing library will contain all the objects required for the Routing algorithm to function. The Routing algorithm will find the closest node to the user's starting and ending locations and return the shortest path between them, if it exists.

As discussed in 2.3.2, Dijkstra's algorithm seems the most appropriate option to calculate the shortest path from one piste node to another. The differences in native development become apparent when looking at the design of Dijkstra's algorithm. The implementation of Dijkstra's algorithm requires a priority queue. This is a trivial matter for coding in Kotlin: as Java on Android comes with a built in min-heap priority queue: this can be used with no changes being made as Java and Kotlin can easily interact with each other. However, there is no default implementation of a priority queue in Swift or Objective-C, this means that a priority queue will need to be created. There exists a fantastic implementation of a priority queue written in Swift, by David Kopec [43] that with some minor modifications would be suitable for use in Dijkstra's algorithm.

5.3.1 Structure

Figure 20 shows the main structure of the routing algorithm. The routing algorithm is centred on the Piste object (5.3.3.1). This object is responsible for creating all the PisteNodes (5.3.3.2) and Routes (5.3.3.3), the PisteGraph (5.3.3.5), which is used to calculate the shortest path across the PisteNodes and for calculating the route from a user's location to the bottom of the closest piste.

The algorithm was structured in this way for several reasons. To calculate the shortest path the PisteGraph requires vertices and edges, the PisteNodes and the Routes represent these. It was initially thought that these could be contained in one object however, Routes have many different variables (from colour, weight, length) and they would be best served by having their own object and the PisteNode would hold a reference to all the Routes that started from it. The RoutePoint represents the coordinates on the piste map where the PisteNode is or Route passes

through. This RoutePoint is the same as the RoutePoint that is returned from the Location Algorthim. To calculate the route from a user's location to the bottom of a piste, it is necessary to find the location of where a line from the user's location intersects with the piste: these are called Intersections and can be ordered by the distance from the user so that the closest ones are dealt with first. Once an intersection has been found all the relevant information about the intersection is then stored in an Intersection object so that the route to the bottom of the piste can be drawn.

The routing algorithm works as follows. A Piste is created and the nodes.json file for the graph is loaded. The Piste then parses the nodes.json and creates PisteNodes and Routes, it then constructs a PisteGraph, which contains a PiorityQueue. When the user requests a route from the application the closest nodes to their start and end points are found by ordering the PisteNodes by distance. The closest PisteNodes to the start and end locations are passed to the PisteGraph, along with the array of all PisteNodes. The PisteGraph sets up the PisteNodes so that shortest path can be found. The PisteGraph then traverses the underlying graph and computes the shortest path; it does this by looking at the closest node and then all the routes that come from it. Each time it encounters a node it updates the distance to the node and then updates the priority queue. Once the destination node has been found, the PisteGraph then iterates backwards across the graph starting at the destination node, and checking the previous nodes until it reaches the original starting node. Once it has found the route, it returns it to the user as an array of PisteNodes. This array can then be used to draw a path across the piste map. A more detailed description of how Dijkstra's algorithm works can be seen in 5.3.2.

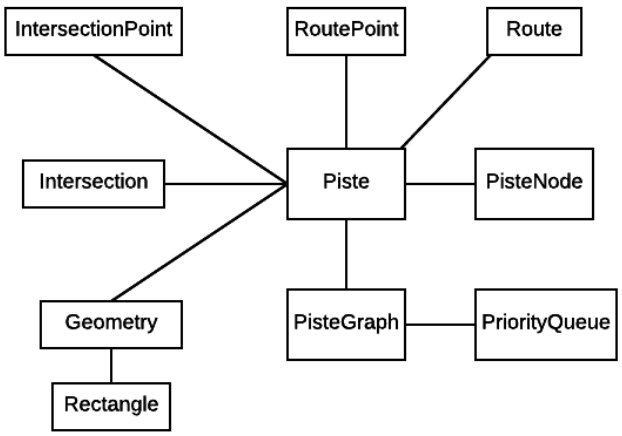


Figure 20: Overview of the main interactions of the Routing Algorithm

5.3.2 Example of Dijkstra's Algorithm

Section 5.3.1 described the interaction of the objects that are involved in the calculation of the shortest path. This section will describe how the calculation is actually performed.

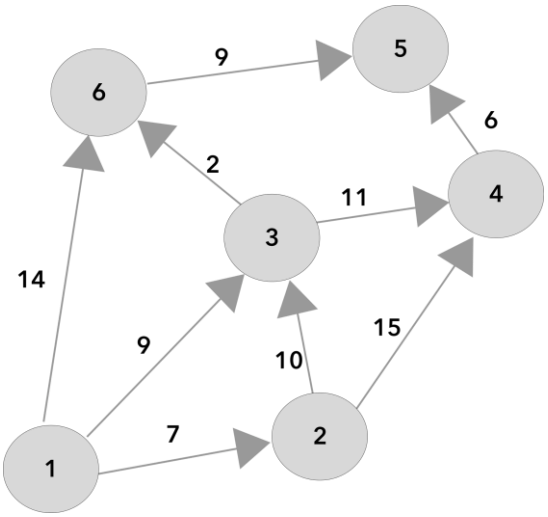


Figure 21: A graph with 6 nodes, showing the connections and their weights

Figure 21 shows a connected graph containing 6 nodes. Each node is connected either by a directional edge going to it or coming from it. An arrow marks the direction along the edge and the number beside the arrow marks the weight of the edge.

In this example we will traverse the graph looking for the shortest path between nodes 1 and 5. Each node has two variables: the **distance** that has been travelled to reach the node from the starting node, and a reference to the **previous** node. When the graph is first setup all the distances for every node are set to infinity and the previous nodes are set to null. The starting node, node 1, has its distance set to zero. This is illustrated in Table 2.

id	Distance from start	Previous node id
1	0	null
2	Infinity	null
3	Infinity	null
4	Infinity	null
5	Infinity	null
6	Infinity	null

Table 2: The initial setup of Dijksta's algorithm, starting node's distance set to 0

Dijkstra's algorithm greedily consumes the routes from each node, this means that it will search the closest nodes first, and then move outwards. The closest nodes to node 1 are in

order 2, 3, and 6. So the algorithm visits node 2 first, as it is closest, then node 3 and finally node 6. As the algorithm hasn't visited node 2 before it updates the distance from the start as it has travelled less than inifinity to get there. The distance from the start is the weight of the edge plus this distance already travelled, the weight is 7 and the distance already travelled is 0, so the distance is updated to 7. It then sets the previous node to node 1. It does the same for nodes 3 and 6. Table 3 shows the result after the first iteration.

id	Distance from start	Previous node id
1	0	null
2	7	1
3	9	1
4	Infinity	null
5	Infinity	null
6	14	1

Table 3: After the first iteration of Dijkstra's algorithm

Now Dijkstra's algorithm moves to the closest node. This is node 2. From node 2 it is possible to reach nodes 3 and 4. 3 is the closest node, so the algorithm checks to see if it can update node 3. It computes the distance from the start, which is 7 plus 10 (the weight of the edge) and this gives a total of 17. 17 is greater than the current distance from the start, which is 9, so the algorithm does not update node 3. The algorithm now checks node 4. The distance at node 4 is infinity so the algorithm updates node 4's distance to be the distance travelled so far plus the weight: 7 plus 15 giving 22, and it sets the previous node to be 2. Table 4 shows the updated nodes in the graph.

id	Distance from start	Previous node id
1	0	null
2	7	1
3	9	1
4	22	2
5	Infinity	null
6	14	1

Table 4 After the second iteration of Dijkstra's algorithm

Next the algorithm moves to node 3, from here it is possible to reach nodes 4 and 6. Node 6 is the closest so the algorithm, being greedy, looks there first. The current distance to node 6 is

14, the distance to 3 is 9. Adding the weight of the edge that connects node 3 to node 6 we get a distance from the start of 11. This distance is less than the current distance of 14 so the algorithm updates node 6 so that it has a distance of 11 and a previous node of 3. Now the algorithm checks node 4, the distance to node 4 is currently 22, the distance to 3 is 9. Adding the weight of the edge that connects node 3 and node 4 gives: 9 + 11 = 20. 20 is less than the current distance of 22 so the algorithm updates node 4 so that the distance is 20 and the previous node is 3. Table 5 shows the state of the graph after this third iteration.

id	Distance from start	Previous node id
1	0	null
2	7	1
3	9	1
4	20	3
5	Infinity	null
6	11	3

Table 5: After the third iteration of Dijkstra's algorithm

Next the algorithm moves to the closest node that it hasn't started from, which is node 6. From node 6 there is only one place that it can go, and that is node 5 (the destination node). Node 5 has not yet been visited so its distance is currently set to infinity. The weight of the edge that connects node 6 and 5 is 9 adding this to the distance to node 6, which is 11, gives a distance of 20. 20 is less than infinity so the algorithm updates node 5 so that its distance is 20 and its previous node is 6. We have now reached the destination node and because Dijkstra's algorithm greedily traverses the graph we can be sure that we have found the shortest path to the destination node. Table 6 shows the state of the graph after the fourth iteration of Dijkstra's algorithm.

id	Distance from start	Previous node id
1	0	null
2	7	1
3	9	1
4	20	3
5	20	6
6	11	3

Table 6 After the fourth iteration of Dijkstra's algorithm

To find the shortest path between the starting node and the destination node, all we have to do is iterate across the graph in reverse. Starting at node 5 we can see that its previous node is node 6. Node 6's previous node is node 3, and node 3's previous node is node 1. Node 1 has no previous so we must have made our way back to the start. Reversing this gives us the following path 1 -> 3 -> 6 -> 5.

5.3.3 Objects

The following objects and classes are used in the calculation of the shortest route across the piste map.

5.3.3.1 Piste

The Piste is the main object that manages and maintains the PisteGraph (5.3.3.5). It loads a JSON file of the nodes and their routes into memory; these are then used by the PisteGraph to compute the shortest path across them. The Piste is also able to find the closest PisteNode to a user's current location, the closest Route to a user's location, and combine this with the shortest path to build a route from the user's location to their destination.

5.3.3.2 PisteNodes

These are nodes on the graph. They have a location (RoutePoint) an id, and a list of Routes (edges) that they are connected to. These are built the first time the PisteGraph is created and are kept in memory for the duration that the application runs. For calculating the shortest path, they are required to have a previous variable (which is initially set to null) and a distance variable (which is initially set to infinity). They also have a function that allows them to add routes. PisteNodes implement the comparable interface allowing them to be easily sorted by the distance parameter.

5.3.3.3 Route

A route is basically a directional edge that connects one node to another. Initially it had a reference to the destination node, an array of RoutePoints that connected it from the starting node to the destination node, a colour, a weight, a name, and a subtype. Later on in the project a pixel length, route length and a route time was added, as this information become available from the CMS.

5.3.3.4 RoutePoints

These are a custom location point, storing an integer x and y value which correspond to a point on the piste map. These RoutePoints are shared between the location and routing algorithms, see 5.2.3.6.

5.3.3.5 PisteGraph

The PisteGraph contains the implementation of Dijkstra's algorithm. To compute the shortest path an array of PisteNodes is passed to the PisteGraph along with the starting node and the destination node. The algorithm then runs as described in 2.3.3.2. A function is used once the algorithm has finished, which looks at the previous nodes from the destination node and it is able to calculate the shortest route by traversing backwards across the graph.

5.3.3.6 Priority Queue

A Priority Queue was required holding the PisteNodes while the algorithm searched for the shortest path. In Kotlin, this was trivial as Android Studio comes with a min-heap priority queue. In iOS a priority queue was required, David Kopec's [43] implementation of a priority queue was used and modified by the addition of a function that would update the heap. The function worked by finding the index of the object that had been updated and then performing a swim [9] and then a sink [9] on that particular object in the priority queue: causing the queue to update correctly.

The way that the priority queues function in Kotlin and Swift differ. InKotlin it was necessary to remove the node from the queue and then re-add it to the queue once the node has been updated. Whereas in Swift the queue was updated by using the added updateHeap function on the node that had been updated.

5.3.3.7 IntersectionPoint

The IntersectionPoint was created so that the distance to the point of intersection and its coordinates could be returned from the helper function.

5.3.3.8 Intersection

An object that contains all the information about the Intersection: it is used to calculate the route from a user's location to the nearest node. The Intersection contains the Route that the intersection is on, the coordinates of the intersection, the start and end points of the Route, the distance it is away from the user and the node that the Route starts from. This was created so that all intersections could be compared and the relevant information to construct the route from the user's location to the bottom of the closest piste could be passed to the appropriate functions.

5.3.3.9 Geometry and Rectangle

A class of static methods that help calculate distances, intersections and rectangles that are then used by Piste object to help in the calculation of the shortest path.

5.4 Single Library for both code bases

Although the primary result of this project is to create native libraries the idea of creating a single library that could be used in both in applications was also explored. Android and iOS both have the facility to utilize libraries created in C++. Extending the native libraries to become a single library that could be used in both applications would require more extensive testing and learning a third language for this project. The idea was proffered to Dogfish, but it was rejected as they felt that it would add unneeded complexity to the application. Although not impossible, it can become more challenging to manage C++ objects in iOS and Android projects. This means that there would need to be two sets of objects maintained, those required for the C++ library and those required for the native application.

5.5 Showcase Applications

This section corresponds to the User Stories given in the Application Epic (see 4.5). The showcase applications will both be written natively for Android and iOS. They will both contain the libraries for calculating the location on the piste map and the shortest path across the piste map. The showcase applications are a vehicle to demonstrate the functionality of the libraries before they are implemented in the actual Crystal Ski application.

5.5.1 Displaying in the Showcase Application

As piste maps can be significantly larger than a phone's display it is important to have a scrollable and zoomable view that the user can use to see the full piste map. Figure 22 shows the relationship between the phone's display and a piste map. If a piste map were to be fully displayed on the screen then it would be too small to be of use, thus rendering its functionally useless. By placing the piste map on a scrollable and zoomable view a user can scroll to see more of the map and then can zoom in to see more of the details that it has.

Both Android and iOS have scrollviews that can be used to accomplish this, however as the piste maps are fairly large, file rendering issues can occur on lower powered Android devices. It was decided that an external dependency [44] that allows large images to be tiled should be used, thus reducing the overhead on the device.

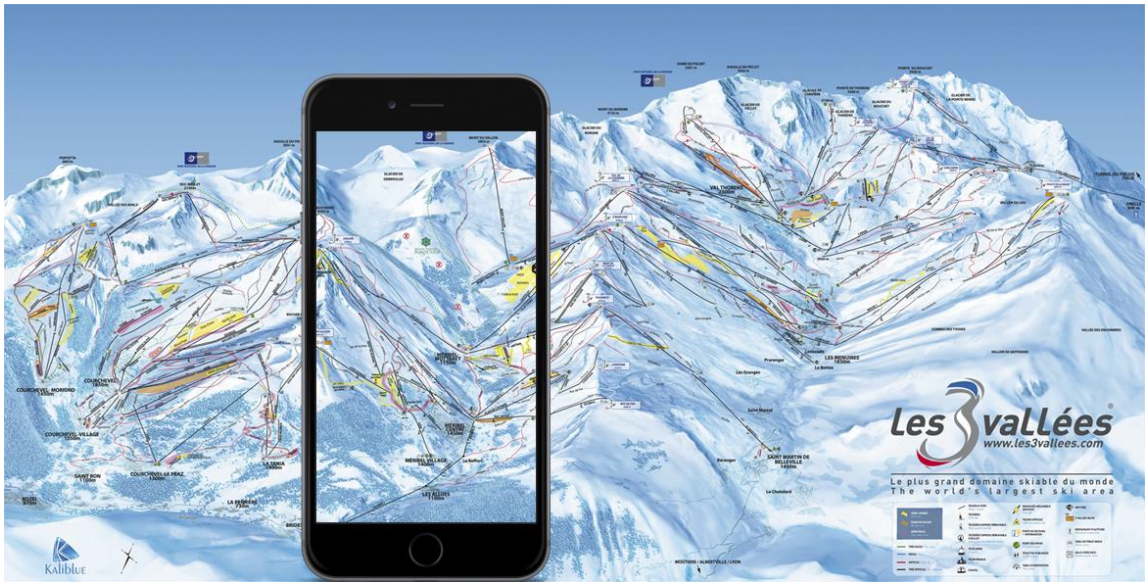


Figure 22: The piste map and how much is shown on a phone's display

5.5.2 Android and iOS Device Differences

Swift and Kotlin are both very similar languages and as such the differences in these languages are not expected to cause many differences between the algorithms for calculating the location and the routing, see sections 6.3 and 6.4 respectively for their implementation. The main differences are expected to come from the way that the underlying iOS and Android SDKs function with regard to the user experience and user interface. These differences will be explained in more detail in section 6.6 where a comparison of the implementations will be made.

6 Implementation

The initial implementation of the Location and Shortest Path algorithms was created in Swift first and then converted to run in Kotlin. Swift has Playgrounds which allow for code development without having to build an application every time you want to test code. This makes developing and prototyping in Swift very quick as it is possible to check one's results quickly.

6.1 Dealing with JSON

In Swift it was decided to use the dependency manager Carthage [45] to install SwiftyJson [42]. The reason for this is that it creates smaller projects than using CocoaPods [46] and gives faster build times. This is useful when testing an application to see if it produces the desired results. Although Swift Playgrounds cannot install dependencies it is possible to create a playground inside an existing Swift project, this then opens up the possibility to have access to any dependencies that the project has inside the Playground. Using this method meant that it was possible to utilise the best features of projects and Playgrounds.

SwiftyJSON allows quick and easy recovery of values from a JSON object. It removes the need for a lot of boilerplate code while still leaving the optionality of the results if required. Both the Plots and the Routes are JSON files and these need to be converted into their appropriate objects. It was considered that the objects may be stored in a database, however as the JSON files will be stored on the device it would add unnecessary complexity by storing it in a database.

Similarly in Android, Klaxon [41] was used to read the JSON from the files. It was loaded into Android Studio by using Gradle [47], the default dependency manager in Android Studio.

6.2 System Overview

The system is made up of three parts: section 6.3 deals with the implementation of the location algorithm, sections 6.4 and 6.5 deal with the routing implementation, and section 6.6 deals with the creation of two native mobile applications and the differences in the implementation. Finally, section 6.7 shows a walkthrough, shown side by side, of both applications.

6.3 Location Algorithm

The initial implementation used similar triangles to calculate the location of the user, taking their location in a triangle made up of latitude and longitude coordinates and converting it to a x and y coordinate within a triangle made up of x and y piste map coordinates. However, as every triangle is affine it would seem more appropriate to calculate the affine transformation

that maps the geographic triangle that the user is in to the piste map triangle, and then using this transformation convert their geographic location into a piste map location.

Although Dogfish authorised the creation of the affine transformation method, they also wanted their existing method converted to run natively on both Android and iOS, so the first step was to convert their JavaScript code into Swift and then Kotlin.

6.3.1 Calculating a User's Location

To calculate a user's location, the Location object is passed the user's latitude and longitude. The TrianglePoints are then ordered based on the distance that they are from the user's location. The first TrianglePoint in the array is chosen, this TrianglePoint is the closest point to the user and it will be in all the Triangles that are considered; this is done so that it helps reduce the number of Triangles. If we were look at the possible permutations[5] for an array containing 300 TrianglePoints there would be 26,730,600 Triangles to consider. We can reduce this number to 89,102 by always using the closest TrianglePoint as one of the vertices of our Triangles. This is still a lot of Triangles to consider with many of these Triangles not being usable due to the fact that the user's location is not inside them. Dogfish had decided to use only the first 16 entries in the array[6] leading to total possible Triangles to consider of 210. This is much more manageable[7]. There is no specific reason for using the number 16, however checking points that are further away would just lead to more triangles being discarded as being too big. Setting a limit allows the calculation to be speedier without a loss of accuracy. Dogfish had found that using Triangles with large obtuse angles did not work well for the similar triangle calculation, so any Triangle that had an angle larger than 175 degrees is ignored. Finally the sizes of the Triangles are compared, with the smallest one being selected for the location calculation: the BestTriangle. If a BestTriangle has been discovered then either the Similar Triangle Method (6.3.2) or the Affine Transformation Method (6.3.3) is used to calculate the user's location on the piste map. The differences in the results from these methods are discussed in section 7.2. If no BestTriangle can be found, meaning that the user is not inside any Triangle, then the method in section 6.3.4 explains how an approximation of the user's location can be found. The end result is a RoutePoint containing the user's location,

---

[5] Using permutations we find that that for a 300-node array: $P(300,3) = 26{,}730{,}600$, if we use the closest point in every triangle we find that the number of permutations drops to $P(299,2) = 89{,}102$.

[6] The first 16 entries include the closest TrianglePoint, excluding this we have $P(15,2) = 210$ possible Triangles

[7] The application was run over the full 89,102 possible Triangles, calculation time went from a few hundredths of a second to around 30 seconds, with the same result being produced.

before this is returned is it checked to see if it is within the bounds of the map. If it is within the bounds of the map, then its isValid variable is updated to true.

6.3.2 Similar Triangle Method

The Similar Triangle method was conceived by Dogfish and is implemented in their CMS. Initially two triangles (based on the geographic coordinates) are created see Figure 23. The first, ABC, is made with the coordinates of the BestTriangle, the second, ABD, is made with the two points closest to the user's location and the user's location, D. Next we extend the line from the closest point A, through the user's location at D until it intersects with the line BC at the point E, Figure 24. This gives the distance AE and it can be compared with the distance AD to give a distance ratio. Then taking a triangle made up of the coordinates of the BestTriangle from the piste map (piste triangle) it is possible to work out the length of the line AE on the piste triangle. Using the distance ratio we can now calculate how long the line AD should be on the piste triangle and then the expected coordinates of the user's location on the piste map.

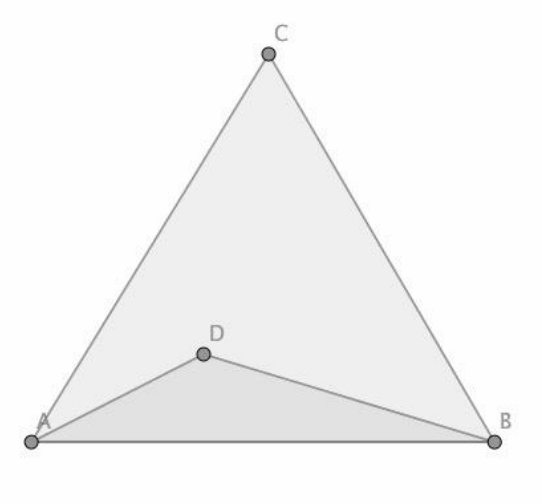


Figure 23: Two triangles are created on the geographic map

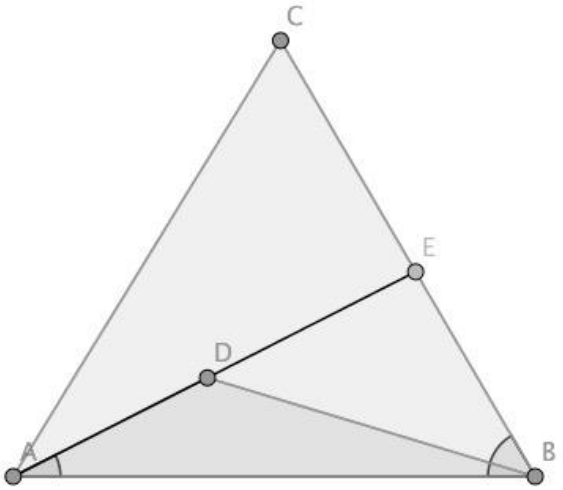


Figure 24: Creating a third triangle ABE

6.3.3 Affine Transformation Method

Although the method Similar Triangle method produces a good approximation of the result, it is flawed in its implementation as it makes the assumption that the triangles on the geographic map and the piste map are similar. This is in fact not the case as the piste map is not a 1-to-1 mapping of the geographic map.

As all triangles are affine [11], and because of this there exists a transformation that will transform one triangle into any other triangle it was decided to use this approach, as it should produce a more accurate solution.

Initially it was thought a solution by using the properties of matrices could be created, however it transpired it was not viable. This was due to the fact that it is not possible to calculate the inverse of 2x3 matrix. After a consultation with Dr Jess Enright, a mathematics lecturer at the University of Stirling, it was suggested that the best way to create the transformation matrix would be to solve a system of six linear equations. For each pair of coordinates in the BestTriangle we use the following to calculate the affine transformation matrix

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Each pair of coordinates gives two equations in the form

$$x' = xt_{11} + yt_{12} + t_{13}$$

$$y' = xt_{21} + yt_{22} + t_{23}$$

The resultant six equations can be solved algebraically to find the values of $t_{11}, t_{12}, t_{13}, t_{21}, t_{22}$ and t_{23}. These then can be used to convert the user's location on the geographic map to the their location on the piste map.

6.3.4 Outside of the Triangles

Unfortunately, the methods described in 6.3.2 and 6.3.3 will not work for a point that is outside of the triangles. Dogfish had stated that calculating the location outside of the triangles was an optional requirement, however it was still necessary to translate their solution from JavaScript to Swift and Kotlin.

Figure 25 shows the situation of when a user is outside of all the triangles. The user is located at point D, the furthest point from the user is at point C, and the points A and B are the two closest points to the user.

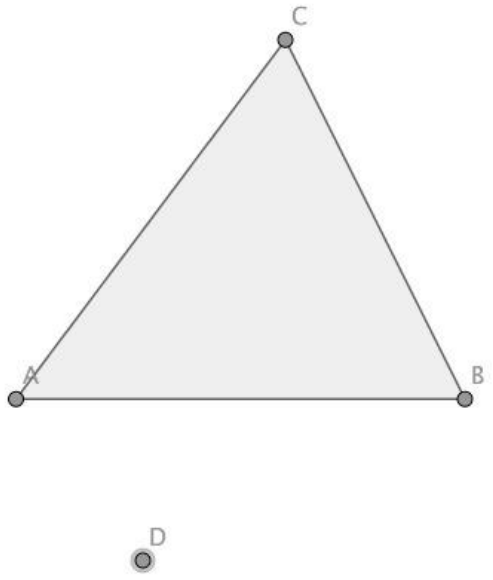


Figure 25: The situation when a user is outside of all of the triangles

Calculating the angles and the lengths AC and DC shown in Figure 26 for the geographic triangle can then be used in conjunction with the angle and A and the length AC in the piste triangle to create an approximation for the user's location.

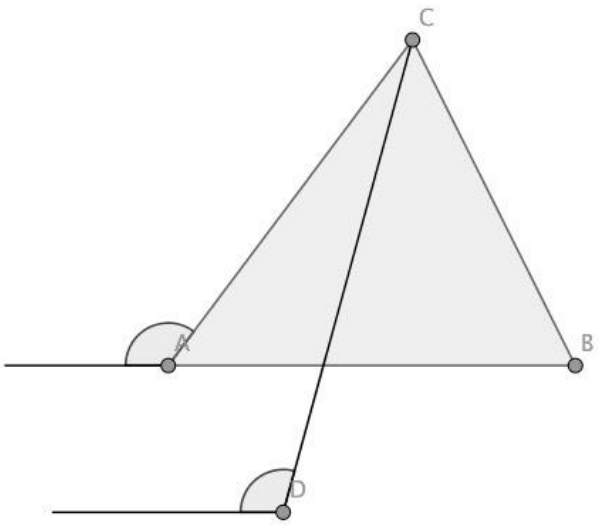


Figure 26: The angles that the lines AC and DC make with the horizontal

Although this method gives a result, it however is far from satisfactory. As the triangles on the geographic map are not similar to the triangles on the piste map this method will never yield a truly accurate location. Looking to improve upon this method, different solutions were postulated but none were any significantly better. Using the same affine transformation as in 6.3.3 produced encouraging results but as the affine transformation would only produce accurate results for a point located within the bounds of the triangle the method does not extend well to the calculation of a point outside of the triangle. Looking to extend the method that Dogfish had created and combine it with an affine transformation, the point of intersection between the lines AB and CD was looked at. However, it must be noted that not all user positions will cause the line CD to intersect with the line AB, thus rendering this method useless. Though if it was possible to find a triangle that did have an intersection between AB and CD, the ratio of the line CD to the line segment from the intersection to the point C on the geographic triangle may not be maintained on the piste triangle. Again, the fact that the triangles lack similarity causes a problem and a better solution to this has still to be found.

6.3.5 Location Summary

If the triangles were similar the affine method and the similar triangle method should produce the same result. However, the triangles are not 100% similar, therefore the Similar Triangle Method will not produce the most accurate result. All triangles are affine [11] and that means that there exists an affine transformation that will transform one triangle into another. It is possible to find this transformation by solving a system of equations. This result is surely the most accurate out of the two, mathematically.

As it is not possible to travel the globe and test the location algorithm on the actual piste, the CMS has the ability to display a test for the user's location. The affine transformation was converted from Swift to JavaScript so that a comparison could be made with the similar triangles method. By clicking on the geographic map a circle appears on the piste map showing the user's expected location. The CMS was updated to show two circles, a red circle showed the expected location from the Similar Triangles Method, and a pink circle showed the expected location from the Affine Transformation. This allowed for a visual comparison to be made. It should be noted that it is difficult to ascertain which method is superior in accuracy; however, one would expect the Affine Transformation to be superior due to it being more mathematically correct.

The implementation of the algorithm in both Swift and Kotlin has no significant differences; for all intents and purposes they function exactly the same.

6.4 Route Finding

The previous iteration of the Crystal Ski application showed recommended routes. Administrators using the CMS created these routes and they were then used in the original application and displayed as an overlay on the piste map. This information was then taken and made into a graph.

6.4.1 Calculating the Shortest Route

The shortest route is computed by using an implementation of Dijkstra's algorithm. The PisteGraph is passed the starting and ending nodes of the route, and the graph that they are on. Setting the graph up for performing a search, each PisteNode has its previous node set to null and the distance travelled to infinity. The starting PisteNode's distance is set to zero. The nodes are then added to the priority queue. The priority queue is ordered by the distance variable in the PisteNode. The first node in the priority queue is the origin node, this node is popped from the queue and its Routes are checked following the method described in 2.3.2.2. It would be possible to exclude Routes at this point, meaning that if a skier did not wish to ski more difficult runs then they could be excluded from the shortest path calculation. The problem with excluding routes is that it may result in no valid route being returned. Rather than excluding them, it may be better to change the weighting of the routes by some factor, thus making certain route types more or less desirable. Once the shortest path tree has been computed the algorithm then traverses backwards from the destination node to find the path to the starting node: looking at the destinations node's previous node and then that node's previous node until the starting node is found.

6.4.2 Additional Routing Options

Once the shortest path had been found the next task was to find the closest piste and the route from that piste to the next node. Looking at a route between two nodes A and B, see Figure 27, we can see that it is made up of several segments.

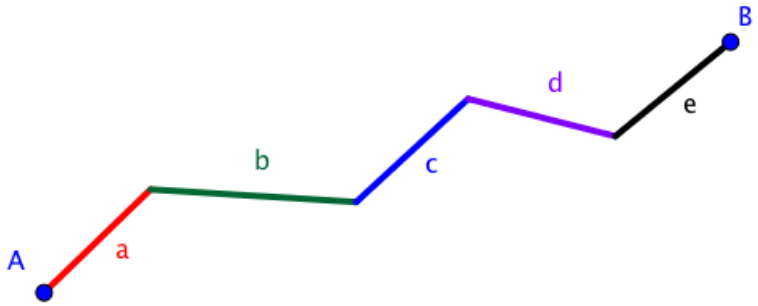


Figure 27: A route is made up of several segments

To find which route is closest to the user's location it is accomplished in the following way. First, all the nodes are ordered by distance from the user's location. Then from the ordered set of nodes, the segments of each route (edge) are checked to see how far away they are from the user. At this point any route that is considered a lift is ignored, as it is not possible to join a lift while it is half way up. These intersections are stored in an array and are then sorted by distance from the user. The closest intersection is then used and a new route from the user's location to the bottom of the piste is calculated. Once this new route is calculated it can then be added to the shortest path by computing the shortest path from the node that the new route ends at.

When finding the intersection for each segment inside a route, one of three situations could occur. Figure 28 illustrates these scenarios. We are hoping that the user's location is at User2, however, it cannot be assumed to be there, so we have to perform a series of checks. Firstly, the line segment is initially assumed to be infinite in length, this is done so that we can calculate the perpendicular distance to the line. Once that is done we have to check that the user is not at User1 or User3. We do this by calculating the dot product of the triangle, looking to see if the triangle has obtuse angles at either A or B. If the user is at User1 or User3 the distance to points A or B is calculated. This is then returned as an Intersection, which can then be sorted, by distance to the user.

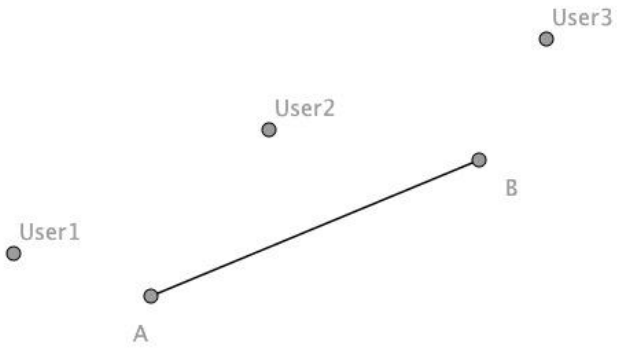


Figure 28: A segment of a route with the three positions a user could be in

Figure 29 shows a route from the user's location, it takes them directly to the closest piste, then to the bottom of that piste. From the bottom of the piste, that node is used as the starting node for the shortest path. The shortest path is then appending to the bottom of the piste and then displayed.

Figure 29: The shortest path with the user's location being off piste

6.4.3 Routing Summary

The routing algorithm works as expected; it finds the shortest path between any two nodes on the graph. Extending the algorithm so that it calculates the route from the user's location to the node at the bottom of the closest piste also works as expected. This allows users to have a better understanding of how to proceed with their route.

Similarly to the Location algorithm (see 6.3) there were no significant differences in the implementations in Swift and Kotlin. The only main difference was the need to use a custom priority queue in Swift as one does not come as standard in the iOS SDK.

6.5 Universal Map Routing

Although the original implementation was designed to work with piste maps, that have a specific set of rules[8], it was thought that it would be possible to extend the functionality to work on other types of maps. A map of the University of Stirling campus was chosen and using the CMS, routes were plotted across it. Doing this uncovered some issues that would not make for a good user experience. When connecting two nodes in the CMS the admin draws the route between them, this means that if the nodes are to be connected in the opposite direction then, as the admin has to draw that route as well, this reciprocal route may not match the first exactly and this can cause unexpected behaviour: such as being routed in the wrong direction.

6.5.1 Implementing Universal Map Routing

If we consider two nodes A and B where they have had the reciprocal route drawn between them (see Figure 30). We can clearly see that the two routes do not match exactly. The blue line represents the route connecting the nodes in the direction A to B, and the green line represents the route connecting the nodes in the direction B to A. User1 is closest to the green line, and User2 is closest to the blue line. This isn't a problem when User1 wants to travel in the direction of the green line (from B to A), and similarly when User2 wants to travel in the direction of the blue line (from A to B). However, if User1 wants to travel to node B, the route would take the user from their location to the intersection with the green line, then on the green line to node A, and finally along the blue line until the reached node B. This situation is illustrated in Figure 31, and we can see that it is far from ideal, as the user would expect the route to go directly to node B without it passing through node A, Figure 33.

[8] You can only go down a piste and you can join a piste at any point, you can only go up a lift but you must join the lift at the bottom.

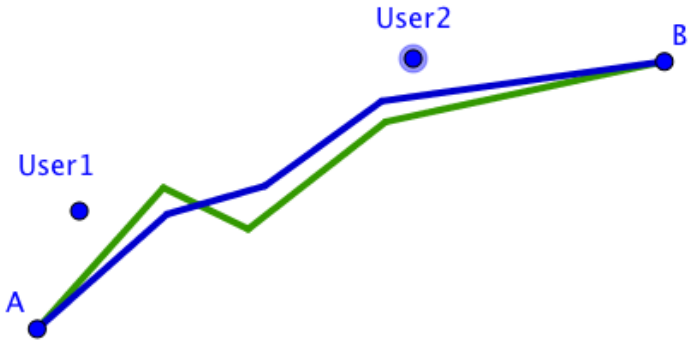


Figure 30: Connecting nodes A and B in the original system

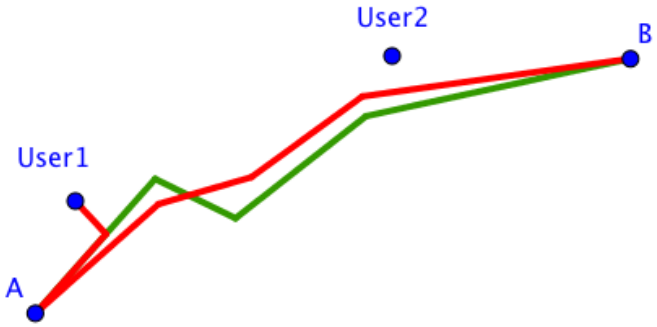


Figure 31: User1 travelling to node B from their current location

It would be preferable if there was only one route connecting the nodes that was bi-directional. By reversing the route that connects A to B we can have one route and reduce the problem down to have one intersection instead of multiple possible intersections (see Figure 32). However, it does not eliminate the problem of the route going via node A when it should go directly to node B. To decide whether the route should go to node A or not, the distance for both options is compared and the shorter of the two is selected and returned to the user. The desired result is shown in Figure 33.

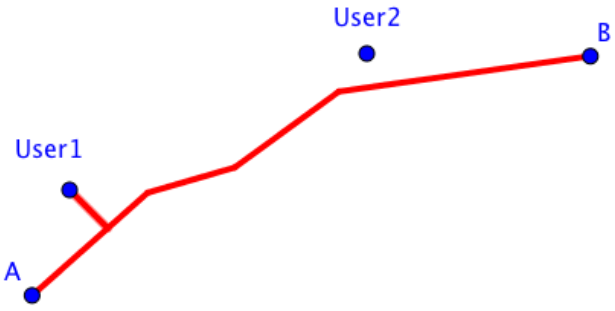


Figure 32: Having one bi-direction route between A and B

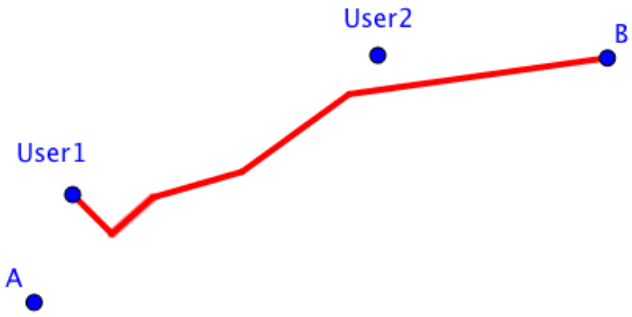


Figure 33: The desired result is achieved by comparing overall lengths

6.5.2 Universal Map Routing Summary

This addition now makes the routing algorithm more versatile, working with both piste maps (maps which have routes that can only be travelled in one direction or routes that can only be joined from one point – lifts) and maps that has bi-directional routes (such as maps that we would expect to find in everyday life – university campus maps and town plan maps).

6.6 Deployment in application

Displaying a zoomable and scrollable image in an application requires different approaches for iOS and Android. Both implementations use a RouteDrawer class, it returns what is required to display the shortest path in each application. Both iOS and Android applications have two ways to calculate the route across the map. Firstly, and the most easily implemented is the double-long-press method, this requires the user to long press the start position and then long press the end position. Once these coordinates are stored the Routing can be performed. The second method is to use the mobile device's GPS functionality to get the user's latitude and longitude. This is then converted into an x and y position on the piste map giving the starting position; by long pressing at the desired destination a route can be calculated. A long press to set the start/end position was chosen because it is difficult to accidentally perform a long press so a user would not accidentally set or change the start/end position while they are just browsing the map.

6.6.1 Application Architecture

It was decided that the applications would be built using MVVM [56] (Model-View-ViewModel) instead of MVC [55] (Model-View-Contoller). The reason for this was to

enhance readability and reusability of the code. Having never built an application in the MVVM design it seemed like a good opportunity to try it out.

6.6.2 The iOS application

This section discusses how the showcase iOS application was built.

6.6.2.1 Scrollable and zoomable views in iOS

UIKit is Apple's framework, which allows a developer to design visual components that users can interact with. UIKit comes with many predefined components that can be sub-classed to add additional features. All UIKit components inherit from UIView and have a view property that can be added to. UIKit has a UIScrollView class that allows you to create a scrollable and zoomable view; other views can be added making it very versatile.

To setup a scrollable and zoomable image, the image is loaded from the application's internal directory into a UIImageView. UIImageViews allow you to display images. The UIScrollView then has its contentSize set to the size of the UIImageView's bounds, then the UIImageView containing our image is added to the its view. To make sure that we can scroll and zoom the image that is on the UIScrollView we need to implement the UIScrollViewDelegate methods, specifically the viewForZooming method so that it returns the UIImageView that we are using. To control the amount of zoom that we want, we need to set the UIScrollView's minimumZoomScale, its maximumZoomScale and the zoomScale that we wish the map to be presented at when it is first loaded.

6.6.2.2 Long press in iOS

Once the UIScrollView has been created and it is now possible to zoom and scroll across the UIImageView that has been loaded into it, we need to add a UILongPressGestureRecognizer. This is so that we can enable the long press functionality required to plot a route across the piste map. Using the gesture recognizer allows the location of where the user long pressed to be found, in fact, we do not actually want to find the location in the UIScrollView as that will only give the coordinates in relation to the screen of the device. Thus a calculation would then have to be performed to determine where on the UIImageView the user had pressed. Instead we can ask the gesture recognizer to return the location within the UIImageView so that these extra calculations are not required and the correct x and y position is returned. Once a start and end position have been given, they are passed to the Routing algorithm (6.4) and an array of PisteNodes representing the route that the user should takem are returned. This array is then passed to the RouteDrawer (6.6.2.4) class which will then return a UIImageView containing the route.

6.6.2.3 GPS in iOS

To implement GPS functionality in iOS requires importing the CoreLocation library. This library is part of the iOS SDK and allows an application to access the GPS functionality of the device. Importing the library is not enough to get access to the functionality. Permissions must be requested in the application's Info.plist[9]. To get updates from the GPS the application must use a LocationManager, this interfaces with the location systems allowing the developer to perform several required tasks. Firstly, the user must be asked whether or not they consent to the application using their location, if a user chooses not to allow GPS location the routing system will default to the double long press system. Next we have to state the desired accuracy that we require, and set up the delegate, see 6.7.5. CLLocationManagerDelegate sends callbacks so that when the location is updated we are notified from this we can get a user's latitude and longitude. This can be used to display their location on the map or to set the start point for a route across the map.

6.6.2.4 Drawing routes and location in iOS

The iOS implementation has a RouteDrawer class that is responsible for drawing on top of the UIImageView that is contained within the UIScrollView. The location and routing drawings both work in the same way. They start by creating a transparent UIImage that is added to a UIImageView (this is the same size as the UIImageView that is being used to display the map), the route or location is then added to the transparent UIImageView and returned from the RouteDrawer where it can then be displayed.

Both the location and route are constructed by using the UIGraphics functionality that comes within the iOS SDK. A graphics context is created and calling the beginPath method starts the path. It is possible to set stroke width, and stroke and fill colours. These can be changed throughout the drawing process as long as the path is written to the context before the changes occur. A simple marker is used to represent a user's location. For the route across the map, a Bezier path is used. The reason for this is that it gives a nice curve between the points rather than just having straight lines connecting them.

As routes and locations will change, it is important to be able to remove them from the view. There are two choices that could be made: remove everything and rebuild it, or just remove the location/route from the view. It would seem overkill to remove everything and rebuild it every time a user changed location or built a route, so it makes more sense to just remove them from

[9] Info.plist is an XML file that stores the essential configuration for the application. Without it an application would not run. If an application requires specific permissions to access sensitive areas of the device, such as the GPS functionality, then they must be requested here before they can be used.

the map's UIImageView when needed. Giving the location/route's UIImageView a tag can help to do this: the tag can then be searched for and if present in the view hierarchy it can then be removed.

6.6.2.5 Notifications in iOS

Several notifications are displayed to the user while they are using the application. There are notifications to enable the GPS system, when a user long presses, when a user is outside of the bounds of the map, when the GPS is disabled, and when the GPS coordinates are updated the accuracy of the GPS is displayed.

The notification asking the user to enable the GPS system is provided by the iOS SDK. When permissions are requested by the LocationManager (6.6.2.3) it asks the user to agree or disagree to them. This is done once, and if they are not enabled then a user would have to go to the application's settings on the device to re-enable them.

An external library called Toast-Swift [49] provides the notifications when a user long presses. This library mimics the same notification functionality that comes with the Android SDK. This library was chosen over the native iOS alert system due to the fact that it is less intrusive. When using native alerts the user has to dismiss them and this is disruptive to the user experience. A vibration was also added to further notify to the user that they had long pressed.

The other notifications are provided by UITextViews that have their content and visibility changed depending on the situation. The UITextView warning a user that they are outside of the bounds of the map is also used to notify the user if the GPS functionality has been disabled. This UITextView is displayed at the top of the screen. The GPS accuracy UITextView is displayed at the bottom of the screen. The UITextView has its background colour changed depending on how accurate the result is, it also displays how many meters it is accurate to.

6.6.3 Android

This section discusses how the showcase Android application was built.

6.6.3.1 Scrollable and zoomable views in Android

Although the Android SDK has an implementation of a ScrollView it is distinctly lacking in its functionality. Large images tend to slow down its functionality, especially on older devices. The methods for adding and removing subviews are more convoluted and require a lot of boilerplate code. For this reason it was decided to use an external dependency called TileView [44]. This is a fully functional scrollable and zoomable view that takes an image that has been broken into tiles and displays it on the device. What makes TileView superior to an Android

ScrollView is the fact that it uses reusable tiles to render the image, rather than trying to render the image in one go. It also has methods for adding routes, placing markers and for capturing long presses.

To make TileView more suited to the application's needs, a subclass was created called TileViewWithPiste. This class takes in several parameters: a Piste object (to help with calculating routes), the width and height of the image that is to be used with the TileView, and the filename of the image to be loaded. By subclassing it meant that code could be used on any piste map could easily be reused and not have to be re-written in different Activities/Fragments.

6.6.3.2 Long press in Android

By subclassing TileView and creating TileViewWithPiste it was now possible to override the onLongPress methods of the TileView class and implement the required functionality. When a long press event occurs it returns an event object, from this event object it is possible to get the location of where the long press occurred on the screen of the device. However, this location needs to be converted to a position on the underlying piste map. The same method to do this is used for both the horizontal and vertical directions. Taking the calculation of the horizontal coordinate as an example the method is as follows: take the event's x position add this to how far the TileView has scrolled in the x-direction (there are methods built into TileView that give you these values), then take this total and divide it by the scale of the TileView. This then gives the correct coordinate on the underlying image. This can then be used with showing the user's location or a route across the map.

6.6.3.3 GPS in Android

To access the GPS functionality permissions must be requested by the application. Firstly, the permissions to be used must be placed in the AndroidManifest.xml file; these tell the device that the application may request these permissions. Next the user must be asked whether they consent to the application using these permissions, this is done by using the PermissionsRequester class. This class checks to see if the user has consented to the permissions and if they have not been asked or they have not consented presents the user with a dialog asking for their consent. If consent is not given then the application works by allowing the user to long press to set start and end points.

Once permission has been given to the application to access these services then the GPSLocation class creates a LocationReceiver that receives updates from the LocationManager. When the LocationReceiver receives an update it sends an intent to the

LocalBroadcastManager which in turn sends out a broadcast so that any Activities or Fragments that are listening for these specific broadcasts are then updated.

The Fragment or Activity that created the TileView can then register for updates from the LocalBroadcastManager and when it receives these updates it can then update the user's location on the TileView accordingly.

6.6.3.4 Drawing routes and location in Android

Once a start and end location have been created the Android application uses the functionality of the TileView to add them to the view.

For adding a user's location TileView provides an addMarker method that allows a marker that is an image to be placed at any point on or off of the map. Initally the marker is created and placed when the TileView is created, however the marker is placed off of the TileView so that it is not visible to the user. When the application receives an updated response from the GPS system the moveMarker method is called on the TileView and the marker's location is then updated.

To draw a route on the TileView there are convenience methods called drawPath and removePath. These are used with DrawablePath objects to draw or remove a path from the TileView. The RouteDrawer class in the Android application takes in an array of PisteNodes and returns an array of DrawablePaths. These DrawablePaths can have individual properties set, such as the colour and stroke width. These paths are then be added to the TileView by iterating across the array and adding each path to the TileView. To remove the route from the TileView we just perform the reverse operation, we iterate across the array removing each path from the TileView.

6.6.3.5 Notifications in Android

Several notifications are displayed to the user while they are using the application. There are notifications to enable the GPS system, when a user long presses, when a user is outside of the bounds of the map, when the GPS is disabled, and when the GPS coordinates are updated the accuracy of the GPS is displayed.

As mentioned in 6.6.3.3 when the application asks the user for permission to access the GPS functionality a native Dialog is displayed. If the user declines the permissions they will be asked again the next time they launch the application and they will have the option to decline any future permission requests.

When a user long presses they are alerted to this fact by a Toast notification, this notification comes as part of the Android SDK and is not as intrusive as using a Dialog. It displays a small

black box at the bottom of the screen with text alerting the user to the fact that they have long pressed.

The other notifications are provided by TextViews that have their content and visibility changed depending on the situation. The TextView warning a user that they are outside of the bounds of the map is also used to notify the user if the GPS functionality has been disabled. This TextView is displayed at the top of the screen. The GPS accuracy TextView is displayed at the bottom of the screen. The TextView has its background colour changed depending on how accurate the result is, it also displays how many metres it is accurate to.

6.6.4 Summary

Implementing the same functionality on the two platforms posed differing problems. For the most part implementation was quicker and seemed easier in iOS compared to Android. It was felt that Android required a significant amount of boilerplate code to be added to the application to get the same functionality as in iOS.

6.6.4.1 MVVM

The choice to use MVVM could be seen as overkill when constructing a small application as MVVM has a tendency to utilise more code just to achieve the them same thing as you can do in MVC. However, MVC (Model-View-Controller) is sometimes known as Massive-ViewController as they can cause UIViewControllers and Activities/Fragments to balloon in size. Using MVVM allowed the code base to remain small in these classes, and to make it easier to follow the code. As the code is separated into ViewModel and View it is easy to switch the underlying model that the View is using. This means that if we wished to change the algorithm associated with calculating the route/location, it could easily be done.

6.6.4.2 ScrollViews

The scrollview in iOS is much more functional than the scrollview in Android. It handles large images easily and it is fairly straightforward to add and remove views to it. Though this is possible in Android using the methods within the SDK, it is not as fluid as it is in iOS. For this reason TileView was chosen to be used in the Android application. Implementing TileView also created a separation in how the routes and user locations were displayed on the map.

6.6.4.3 Drawing routes and marking locations

The RouteDrawer class is responsible for taking an array of PisteNodes and converting it to a route that will be displayed on the map; in iOS it is also responsible for marking the user location on the map.

How the RouteDrawer works in both implementations is fundamentally different. In iOS it creates a transparent UIImageView that is then placed on top of the UIImageView that contains the map. In Android it utilises the TileView's drawPath method and creates an array of DrawablePaths that can then be added. Both produce effective results. However, the iOS implementation allows the use of Bezier curves for the path and this creates a much smoother and nicer looking curve. It also looks better when the map is zoomed out (at the minimum zoom scale), as the TileView keeps the path at a constant size, which isn't a problem when it is zoomed in, however, it is not very pretty to look at when it is zoomed out.

6.6.4.4 Location

Enabling GPS on Android requires the developer to manage permissions, create broadcast listeners and receivers and to manage the response from the GPS systems. It seems, as location is a feature that is often used by application developers that the Android SDK would make it easier for them to do so. The CoreLocation functionality that the iOS SDK has is vastly easier to implement than its Android equivalent and requires less than a dozen lines of code. Getting the GPS to work seamlessly in both applications was a challenge, primarily due to the problems that Android presented. There is an unfortunate bug in the Android application, which has been impossible to solve. If the application is loaded with the GPS functionality switched off it has not been possible to register a listener to discern when it has been switched back on, however the opposite case is not true. If the GPS functionality has been switched on when the application is loaded, the listener registers correctly and can determine whether it is switched off and switched back on again.

6.7 Walkthrough

This section deals with how the two applications look and shows a typical user flow through the application. The Android application is on the right and the iOS application is on the left. Although every effort was made to make the two applications look identical, there are some instances where they are unable due to the constraints by the operating systems and the individual implementations.

6.7.1 Asking for permissions

To access the GPS functionality both applications must request the permission to do so from the user. Figure 34 shows the native requests for permissions. iOS requires the reason for the request to be displayed when the permission is asked for, in Android this is given if the user denies the permission.

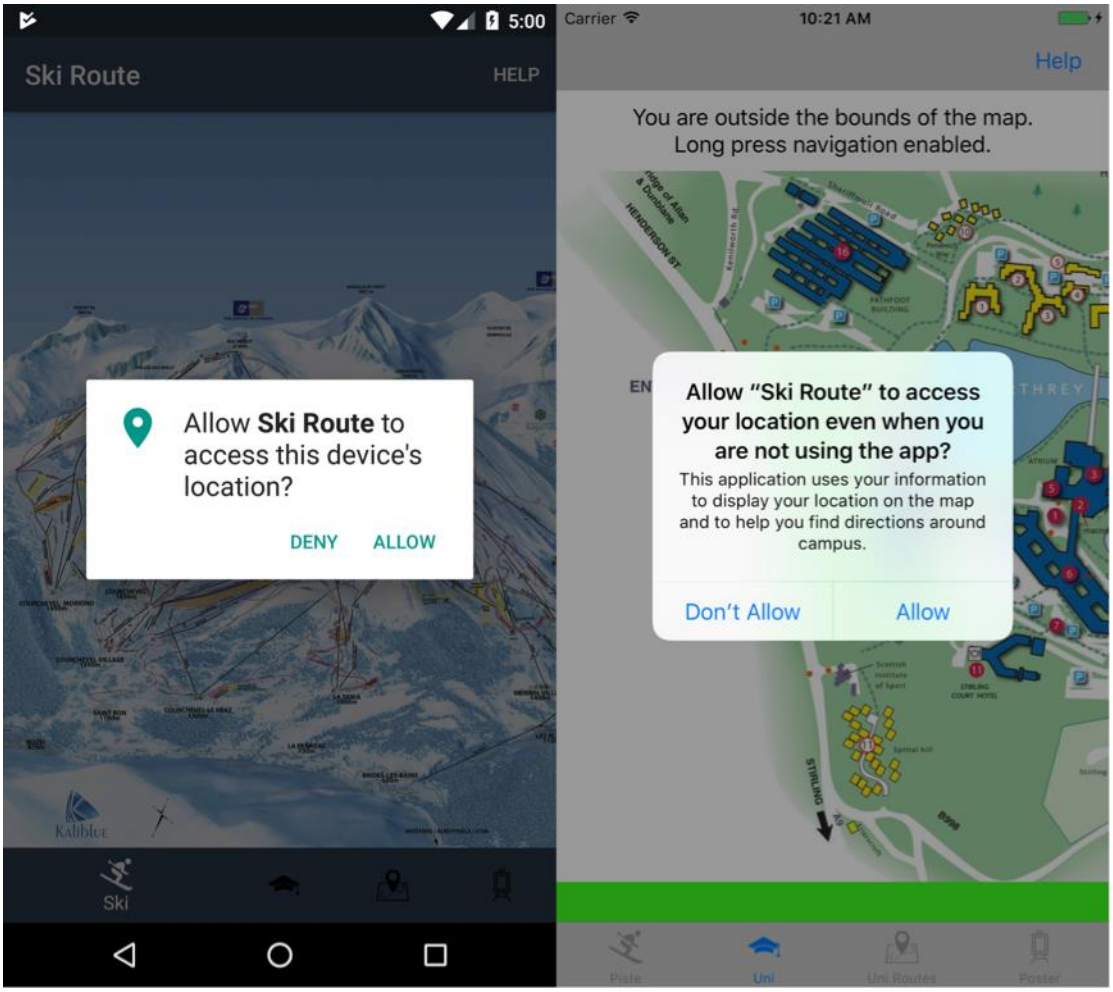


Figure 34: Asking for permissions

6.7.2 No GPS access

If a user denies the access to the GPS functionality or the GPS is not available, the user is then presented a warning stating this, it also prompts the user that long press navigation is enabled. This is shown in Figure 35.

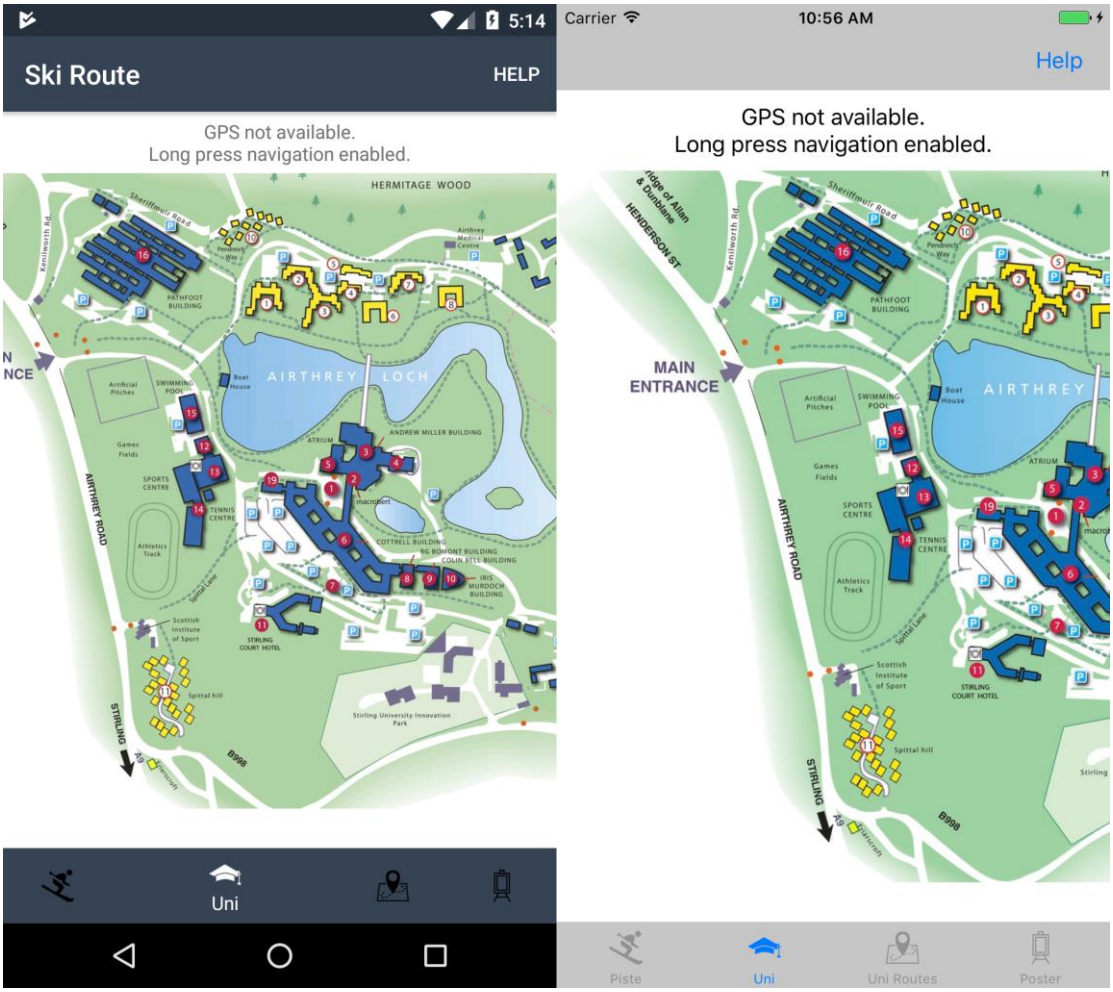


Figure 35: No GPS access

6.7.3 GPS available but outside of the map

In Figure 36 we can see what happens when a user is outside of the boundary of the map but has GPS enabled. The GPS accuracy label updates and shows how accurate the signal is, the message at the top of the device is then updated to reflect that. Similarly, when there is no GPS access, see 0, the user is able to user two long presses to navigate across the map.

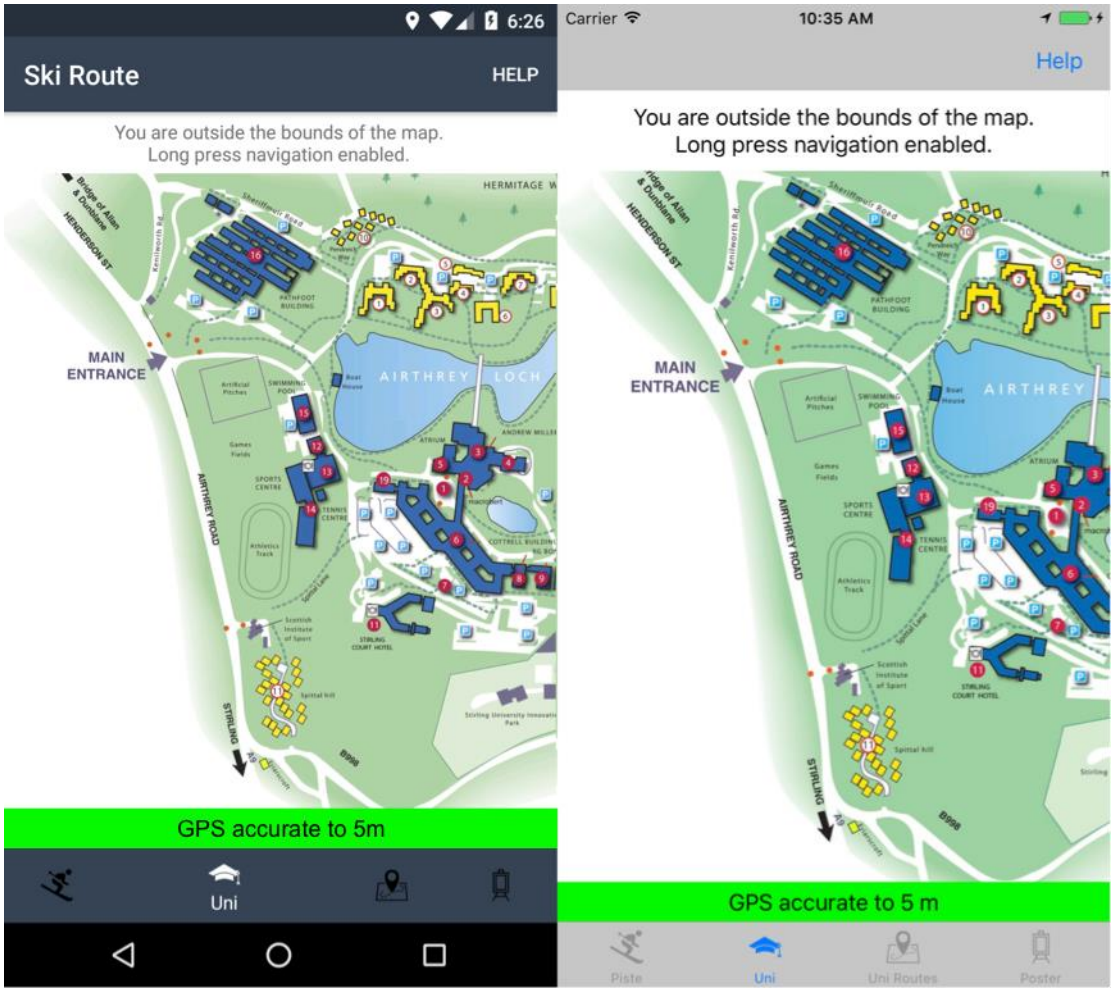


Figure 36: Outside the bounds of the map

6.7.4 GPS available but inside the map

When a user moves with the bounds of the map, see Figure 37, the display updates so that the user's location is shown. However, the message is now removed from the top of the device, this alerts the user to the fact that they no longer need to set a start location as their current location will be used.

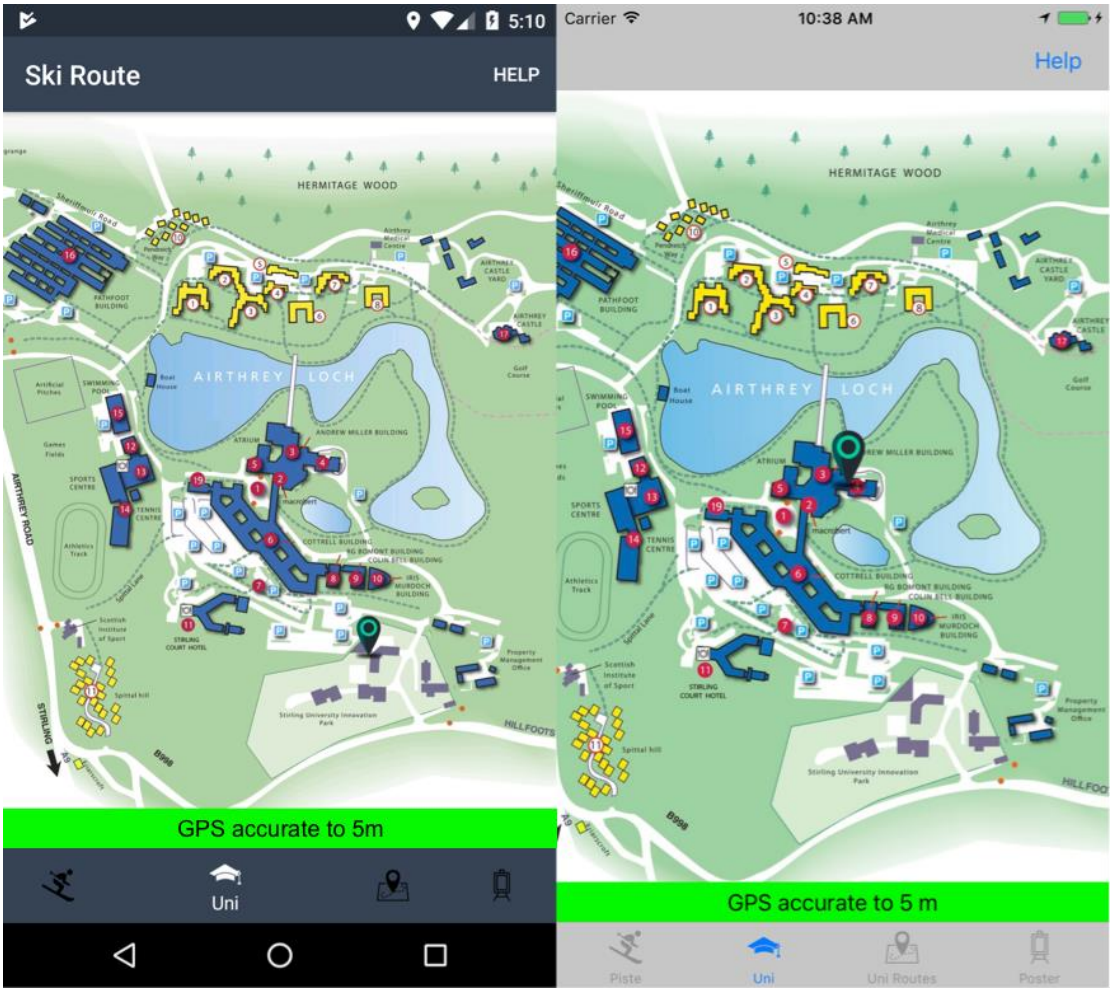


Figure 37: Inside the bounds of the map

6.7.5 Placing a start/end point.

When a user wishes to select a start/end point to their route (start points can only be chosen if the GPS is not available or the user is outside the confines of the map) they are notified when they have made a long press. Figure 38 shows the notification that is given to a user when they complete a long press to set the start point (the starting point is also marked with the use of a custom marker); a similar notification is presented when a user selects an end point. The notification in the Android device is provided by the Android SDK, due to its suitability and usability, a similar system was sought out for the iOS device. Toasty-Swift was found and added to the project. This helps to provide a consistent theme between the applications on the devices.

Figure 38: Starting point placed

6.7.6 Routes on the map

When a user has selected the start and end points on the map a route between the points is automatically calculated and displayed. If no route can be found, then no route is displayed. Both devices will show the same route for the same start and end points.

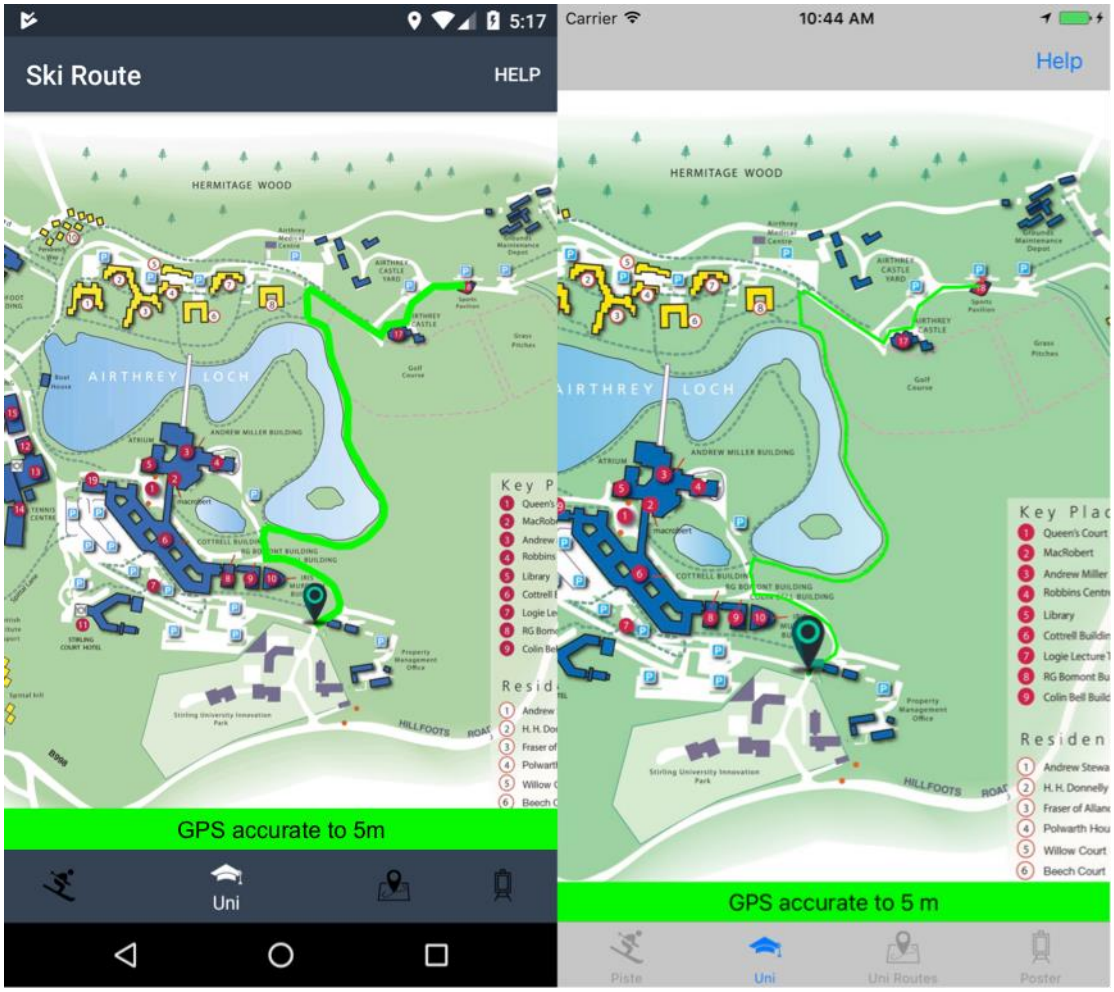


Figure 39: Route on the university map

Figure 39 and Figure 40 show examples of routes. In Figure 39 we can see a route on the university map, and in Figure 40 we see an example of a route on an actual piste map. Between each map and each device there are some differences that should be noted.

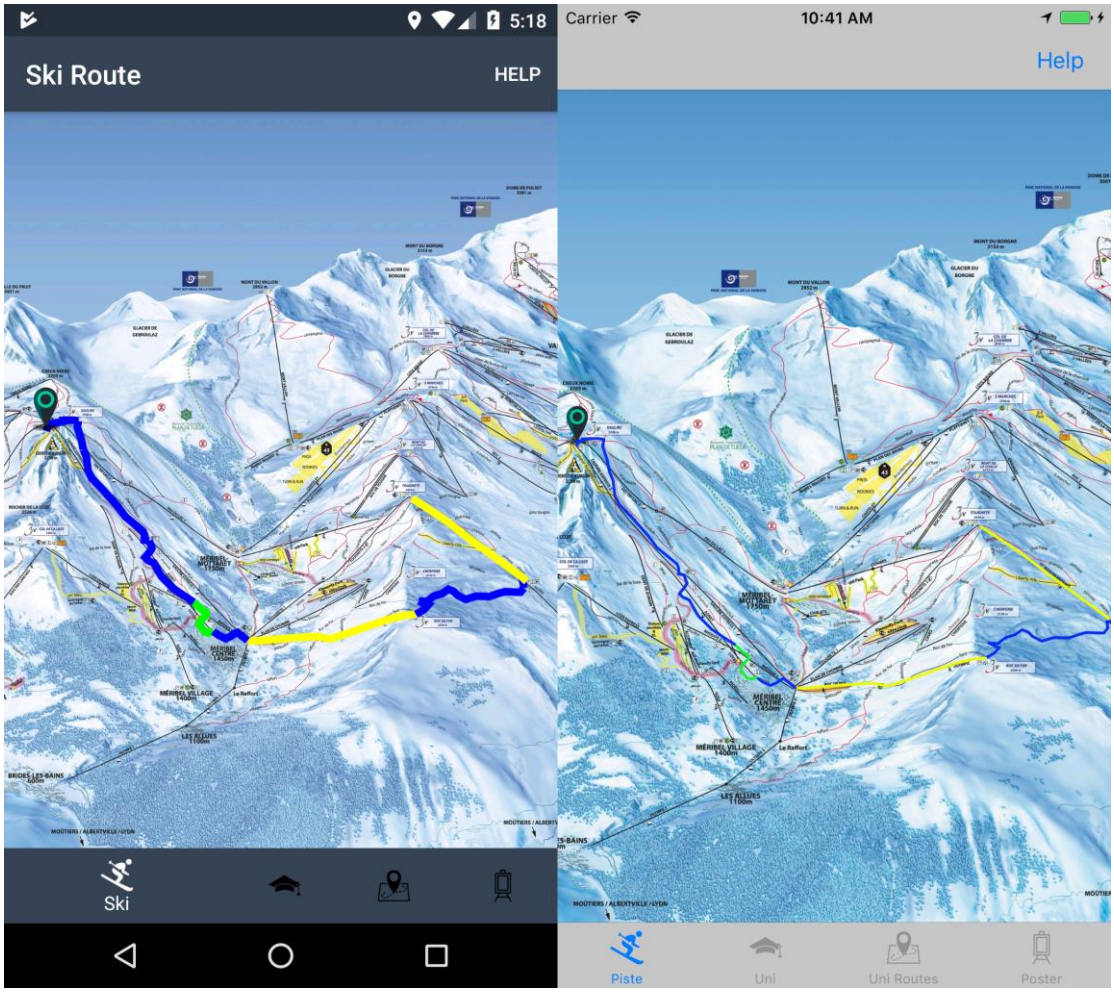


Figure 40: A route on a piste map

6.7.6.1 Device differences

The routes are displayed differently on the different devices. On an Android device, when the map is zoomed out, the route appears thicker than it does on the iOS device. This is due to how the TileView handles the route that is displayed on it: it displays the route at the size that it should be displayed at when it is fully zoomed in and scales the route accordingly so that the size is not changed. In iOS the route is also displayed at the size it should be when fully zoomed in but it does not scale it, so it appears thinner when the map is fully zoomed out in comparison to the Android version.

The scroll view implementation in both applications is the other main difference. As TileView recycles the tiles, so that memory usage is managed for the lower performing Android devices, it can sometimes take a moment to render the image properly. This is not a major issue on the latest Android phones but it does cause some lag on the older models. As the scroll view is implemented differently in iOS it does not suffer from this problem.

6.7.6.2 Map differences

There is a difference between the routes on the university map and the piste map. This difference is actually in the underlying graph that connects the nodes. The edges that connect the nodes in the piste graph have different colours and these are shown when the route is displayed: the different colours relate to the different levels of difficulty of the run (green, blue, red and black, and yellow for lifts). All the edges on the university graph were marked as green so there are no changes of colour. The routes between nodes on the university graph are inverses of each other meaning that the route from node A to node B is the same as the route from node B to node A, but in reverse. On the piste graph routes going up the mountain are given by lifts, and routes going down the mountain are given by runs, there is no requirement for bi-directional routes in the piste graph as there is in the university graph.

6.8 Summary

The two applications meet all the requirements of the user stories. There was a small issue with regards to the GPS functionality in Android that means that if GPS tracking is originally disabled when the application loads, it was not possible to register a listener to notice when the GPS is subsequently switched back on: restarting the application with the GPS turned on currently is the only way to register the listener.

The routing was also updated to work on bi-directional routes. Implementations such as the university map are now possible, allowing the routing to be used on a variety of different maps and thus extending its functionality beyond the original premise.

7 Testing

Testing is an important part of the software development process. It helps to find bugs, and to ensure that all requirements have been met. This chapter discusses the testing that took place.

7.1 Testing at DOGFI.SH Mobile Ltd

As seen in Chapter 3 the application was built using Agile principles and that Dogfish subscribes to a daily build with continual testing, see 3.2, against completed User Stories. A large portion of the testing was carried out by Dogfish's quality assurance (QA) team. If something that had been marked as completed turned out to not meet the acceptance criteria or if any bugs had been found, a report was sent back to the developer via the JIRA board and these issues were tackled as soon as possible.

The QA process at Dogfish subscribes to the following format. Once a user story has been marked as done that story is then tested against the current build. It is tested across all supported devices. In the case of iOS it was tested on the following devices: iPhone 5S, iPhone 6S and iPhone 7 plus. As the majority of iPhone users keep their phones up-to-date [53], 92% are on iOS 10 (the latest version) it would not be worth testing below this level. However, Android has a much more fragmented ecosystem. Where 92% of iPhone users are on iOS 10 or above, 91% of Android [54] users are on 4.3 or above, with less than 25% on version 7 or higher. This means that Android developers have to support a wider range of devices and operating systems. Testing was carried out on a variety of Android makes and models. If the user story passes all tests, then it is finally marked as complete. Otherwise if it is to fail, then a bug report is issued and the developer is tasked with fixing the bug as soon as possible.

The showcase applications were also given to other developers at Dogfish to see if they were able to break them. After several days of intense effort on the part of the developers no application crashes were reported and both Android and iOS applications functioned as expected.

7.2 Location

Testing the accuracy of the location algorithm was completed in several stages.

7.2.1 Comparison with original

The first stage was to compare it with the existing algorithm. This was performed on the CMS by showing two different circles when a test was performed. From this visual test it was found

that the Similar Triangles Method and the Affine Transformation Method produced similar results: at times one being better than the other.

To see if the results from the algorithms were significantly different from each other, the differences between the x-coordinates, the differences between the y-coordinates, and the distances between the coordinate pairs were tested to see if they followed a normal distribution. Figure 41, Figure 42, and Figure 43 show the probability plots for these values, and it is quite clear that none of the three data samples are normally distributed. To see if the differences between the coordinate values and the distances between the coordinate pairs were statistically significant, a Wilcoxon signed-rank test was performed on each data of the samples. The Wilcoxon signed-rank test is an alternative to a t-test and is used when a data sample is not normally distributed. For both the differences in the x-coordinates and the differences in the y-coordinates, it was found that there are no statistical differences at the 5% significance level. However, the distances between the coordinate pairs had a statistically significant median distance, though this is not practically important as the median distance is less than half length of the diameter of the of the location marker. This means that the user's location will likely be contained within, or be very close to, the location marker. A full breakdown of the statistical results can be found in Appendix 1.

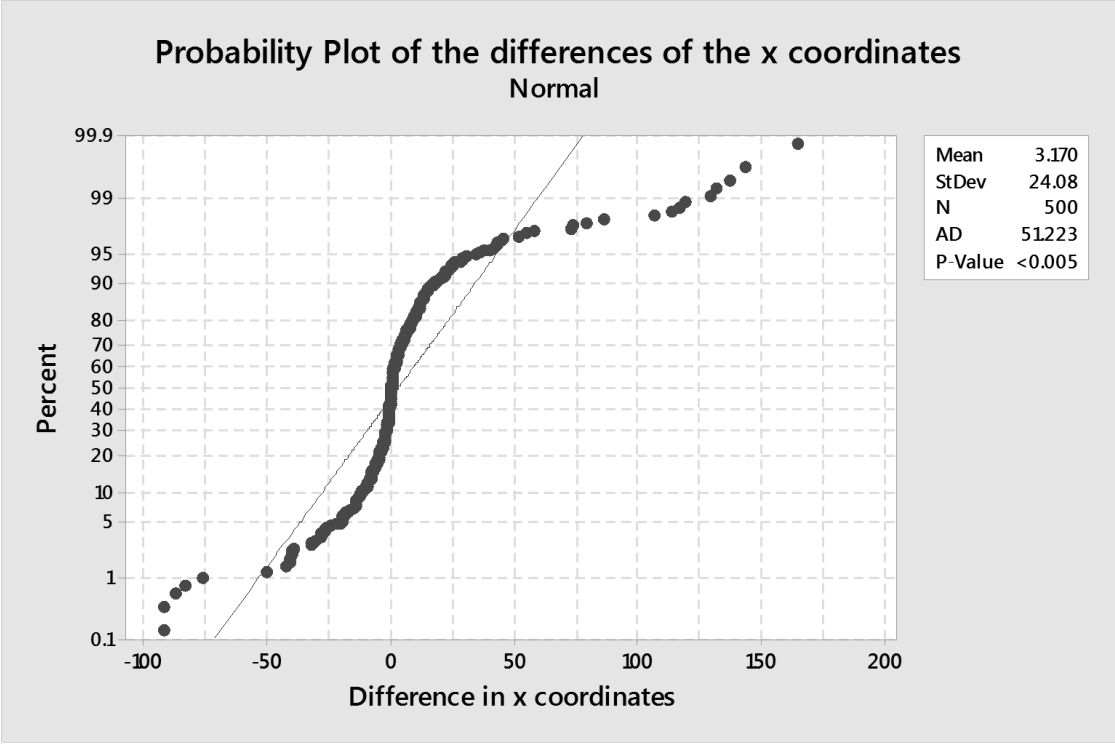


Figure 41: Probability Plot of the differences of the x coordinates

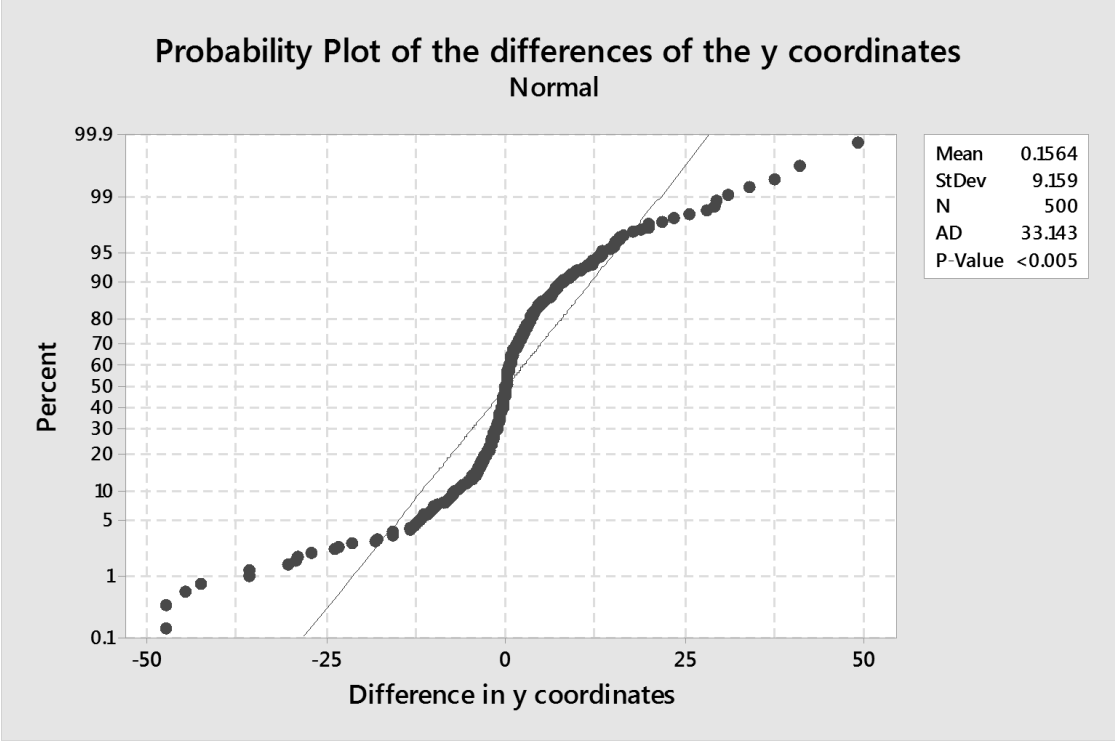


Figure 42: Probability Plot of the differences of the y coordinates

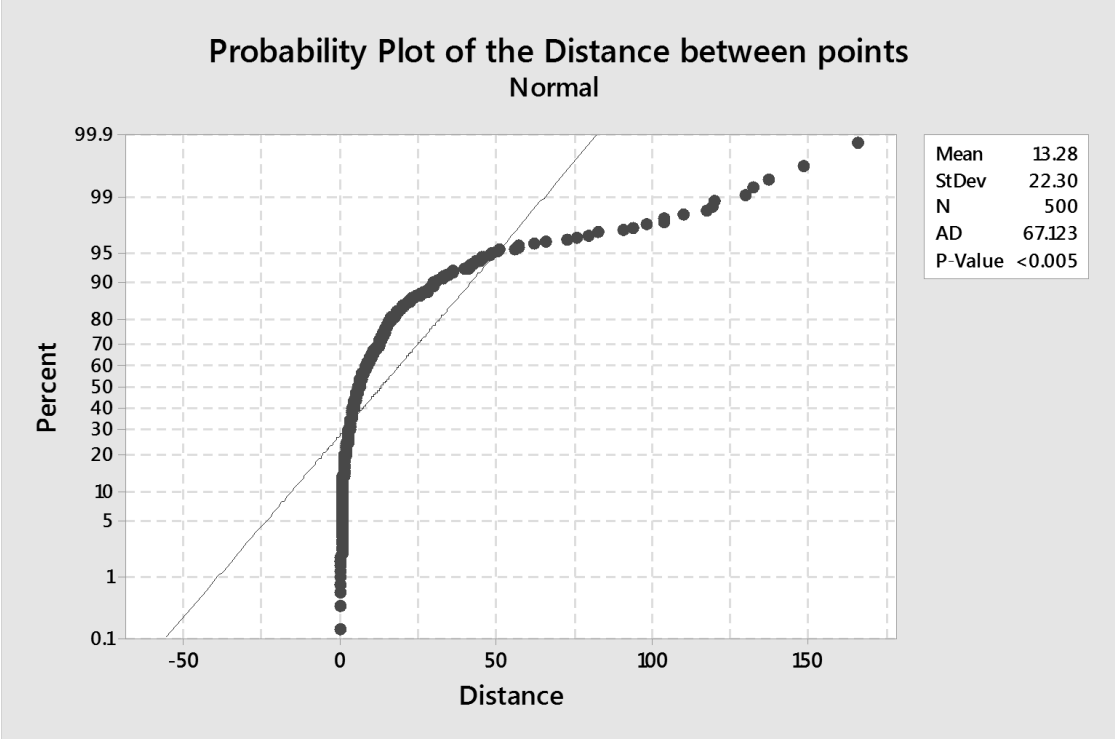


Figure 43: Probability Plot of the distance between coordinate pairs

7.2.2 User Testing

A group of fellow Masters students were invited to test the location library in the real world and to give their feedback on how well it performed.

The students were given an application that tracked the user's location and exported the results to a CSV file, which could then be used for later analysis. The application used a map of Stirling University, an area that the students were familiar with. They were then instructed to walk around the campus and collect data. They were to walk the same route twice. The application would randomly select either the Similar Triangles Algorithm or the Affine Transformation Method for the first route, and then it would use the other algorithm for the second route. The testers would be unaware which algorithm had been used on each route. Though this information would be recorded by the application and added to the exported CSV. The CSV file contained a timestamp, the user's GPS location, and the results from both algorithms. These were recorded at 5-second intervals.

The application would display the user's location on the map as they walked their route. Once the tester had completed their route the application saved the data to a CSV. The tester could then view the route, which they had walked on the device's screen. At this point they were asked to send the CSV along with the answer to the following question: **Is the tracked route shown on the device an accurate representation of the route you walked? Yes or No.** They would then repeat the route they had just walked for the second algorithm. At the end they would answer the same question and export the CSV, but they would also answer another question: **Out of the two tracked routes you walked, which is more accurate on your device, the first route, the second route, both are the same?**

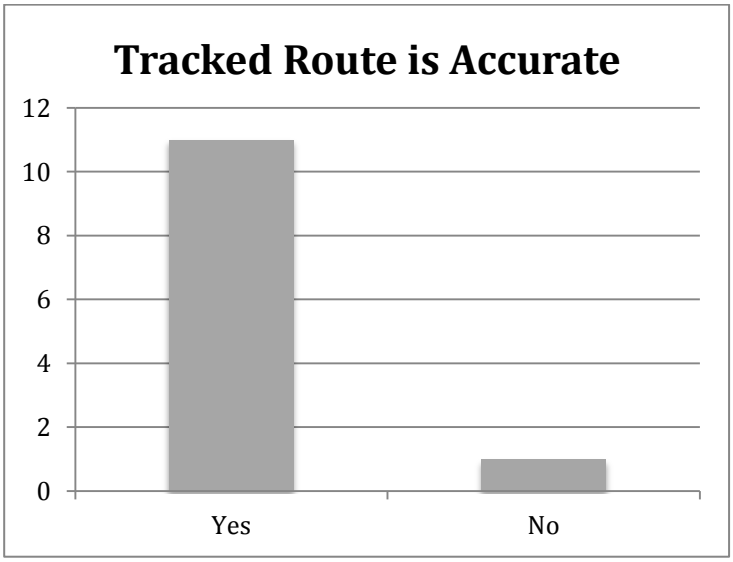


Figure 44: Is the tracked route shown accurate

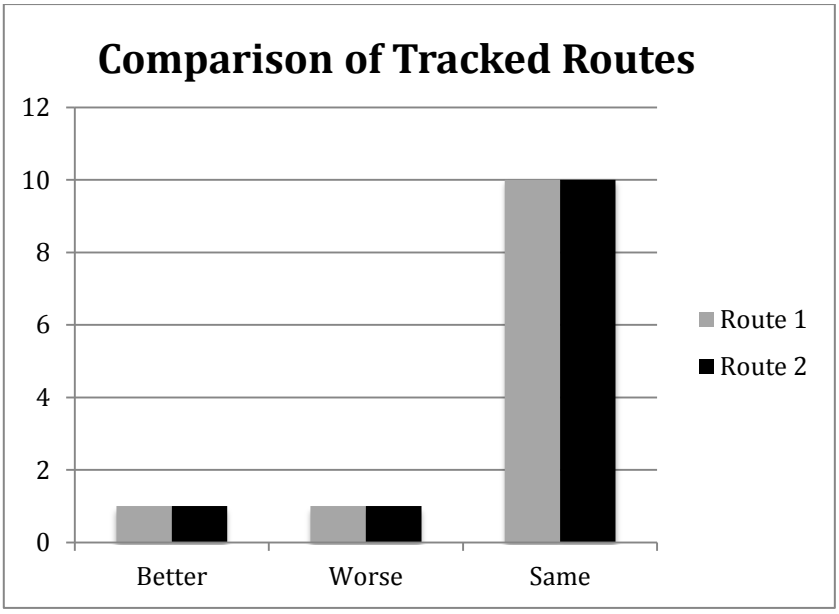


Figure 45: Comparison of Tracked Routes

Figure 44 and Figure 45 show the results from the testers. Figure 44 shows the average result for both routes. 11 out of the 12 users felt that the tracked routes shown were an accurate representation of the route that they had walked. An analysis of the data of the one user who felt that it was not an accurate representation showed that the level of GPS accuracy for their routes had decreased to 200m. That meant that they could be over 200m away from where they actually were.

This result mirrors what was found through the student's own testing and from the statistical analysis in section 7.2.1.

7.2.2.1 Examples of the User Test

In Figure 46 a fairly accurate path was reported on this particular walk, it was only when the user went inside the building that accuracy started to decrease from 5m to 30m. The routes of users were found to be quite accurate, regardless of which algorithm was used, and the comments on the response forms were encouraging that it showed an accurate location.

It was noted that when the accuracy of the GPS dropped, then inaccurate results were found. This it not due to the algorithm calculating the wrong coordinates, in fact, it calculated the correct coordinates, which were correct for the provided data. Rather it is due to the user's actual GPS location being wrong. In Figure 47, it can be seen that the user's location jumps into the Pathfoot building, the building close to the number 16 on the map. When the data for this particular walk was analysed, it was found that accuracy has dropped down to 160m at this point. This explains why the route has made a massive deviation than what is expected.

The goal of the application is not to map routes taken but rather to show the user's location on the map. By having a bar that states the accuracy of the GPS signal, allows users to understand

that their location on the map may not be correct. Though as the device is continually searching for a GPS signal the user's location will be updated frequently. Thus inaccurate locations will only be shown for a short time as the GPS accuracy increases.

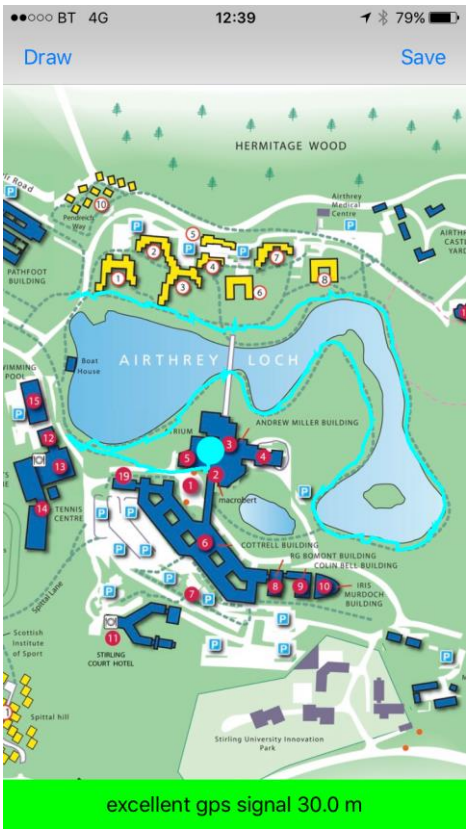


Figure 46: A test route walked around the university campus

Figure 47: Another test route walked around the university campus

7.2.3 Comparison Testing

As both the iOS and Android are to use the same location algorithm, and to check that they had been implemented correctly, a series of location conversion tests were performed. They both produced the same results, so it was concluded that both implementations were working correctly.

7.3 Shortest Path

Testing the shortest path across the map was undertaken in three stages.

The first stage involved using a small graph containing 8 nodes. As this graph was sufficiently small it was possible to determine by hand the shortest path. The hand-calculated shortest paths were then compared with the application's algorithm's shortest paths. They were found to be the same, so it was decided to proceed to the next stage.

The second stage involved comparing it with the shortest path that the CMS could find. A random selection of routes were chosen across the piste map and the CMS produced its result and this was then compared with the algorithm's result for the same routes. This was also found to be the same.

The third stage was to give the showcase mobile application to users with a printed copy of the possible routes. The user would then plan a route by selecting their start point and then selecting their end point. The user could then compare the resultant route across the map with the printed copy of all possible routes. The majority of users stated that the application calculated the route that they expected.

As the algorithm had passed all three stages it was concluded that the algorithm was working correctly.

The speed at which the algorithm performed was also tested. It was found that on an iPhone 6s the algorithm calculated the shortest path in approximately 0.002 seconds, while on a Samsung S3 mini it took 0.03 seconds. It was felt that the algorithm was operating quickly enough to not warrant any further optimisations.

7.4 User Interface

Both the Android and iOS applications were built with similar user interfaces. It was not possible to have testers travel to the actual location of the pistes so a “piste map” of the University of Stirling was made. Testers were asked to try out the application both on and off the university campus. On campus, it would allow the user to use the GPS functionality of their device to create a route around the campus, using a long press to set the destination. Off campus, the user could long press to set the starting point, and then long press a second time to set the destination.

When testing the shortest path on the device some users complained that the algorithm did not appear to return the shortest path. This was due to the fact that not all routes had been created on the map: i.e. not all nodes were connected, and if they were connected, they were not connected in a way that the user expected. To rectify this, a copy of the possible routes was given to the users for testing, this clarified to them, which routes were allowed and the new feedback from users was that indeed the application did produce the correct route.

8 Conclusion

8.1 Summary

DOGFI.SH Mobile Ltd (Dogfish) is a mobile application development company based in Stirling, Scotland. They build native and cross-platform mobile applications using a variety of technologies. One of their clients, Crystal Ski, wanted to update their existing application, which had been originally released 4 years ago, to take advantage of new technologies. The originally application had been coded in Titanium, a cross-platform SDK that allows for one application to be built, however due to limitations of the JavaScript interpreter and the fact that it did not allow consistent access to native APIs in both Android and iOS: a complete re-write was suggested by Dogfish to Crystal Ski.

Through extensive discussions with Dogfish it was decided that a native solution would be best, as it removed the problems associated with cross-platform solutions (see 2.4.1). As the whole application re-write was deemed too large for any one person to take on, a subset of the user stories was created and the problem of improving the location algorithm was proffered. Upon further analysis of the application, it was noted that the functionality to search for routes across the piste map was missing though the underlying information to implement such a system existed. It was suggested that this could be created and added to the application re-write.

It was decided that two showcase native applications would be built: one in Android and the other in iOS. They would contain native libraries that would allow a user to find their location on a map, and from that location find their way to another point on the map. These applications would then be presented to Crystal Ski and if successful would see the libraries implemented and used in the application re-write. The algorithms were written first in Swift and were then converted to Kotlin; it was considered that a single library in C++ could be written so that a single solution would only be required. However, on further investigation and discussion with Dogfish, it was decided that this would lead to a solution that would be too complex to maintain and did not fit the needs that they had at this time.

As Dogfish works in a hybrid Agile method (see 3.2) the project was completed in a similar manner. The project was split into three epics that allowed for iteration, continuous development and testing. The epics were broken down into individual sprints and managed by using a software task board called JIRA (see 3.1.4). The epics were completed in the order that they were given in chapter 4, and as each epic was undertaken, Dogfish's quality assurance team conducted daily black box testing on completed user stories. This is where the user story was tested against its assessment criteria. If a user story did not meet its assessment criteria the

developer was notified and the user story marked as having a bug. Continuous white box testing was conducted to make sure that the data structures and the internal workings of the algorithms functioned as expected.

The showcase applications both utilised their native SDKs to create the UI elements required to display a piste map, though not all the functionality could be provided by the native SDKs. In Android, Klaxon[41] was used to parse and serialize the JSON files, TileView [44] was used to create a scrollable and zoomable map view, and Google Play Services [51] was used for location functionality. In iOS, SwiftyJSON [42] was used to parse and serialize the JSON files, and Toast-Swift [49] was used to create notifications that matched the style provided by the Android SDK.

During the development phase the feedback from Dogfish has been resoundingly positive. They have been impressed by the speed of development and the professionalism in which it was undertaken. The presentation to the Crystal Ski was similarly well received and plans have now begun to integrate the libraries into the current re-write of their application.

8.2 Evaluation

Evaluation at the end of project gives the opportunity to reflect on the processes used, what went well, what did not, and what could be improved. The project's objective as described in section 1.2 was to create two native showcase applications that provided the user the functionality to locate themselves on the piste map using the device's GPS functionality and to be able to plan a route from that location to another point on the piste map.

8.2.1 Choice of language

When developing natively there are only several choices of languages that can be used in each SDK. For Android the main language used to develop natively was Java. That was until May 2017 when Kotlin [52] became a first class language. Having used Java and having developed Android applications in Java as part of the Masters course, it would seem like that it is a logical conclusion to develop the Android application in Java. However, Kotlin brings to the table a cleaner and easier to read syntax, and it provides compatibility with Java meaning that classes written in Kotlin will work with classes written in Java and vice versa. Thus Kotlin was chosen as the primary development language for Android.

Developing natively for iOS there are only two choices: Objective-C and Swift. Both languages work well together and can be used in projects composed of the other. However, the syntax for Objective-C has a steep learning curve and is even less readable than languages like Java. Swift, is similar syntactically to Kotlin and for this reason it was chosen: two thirds of

the development would be devoted to constructing the location and routing algorithms so it would make sense to have them written in a similar way. ,

Due to the syntactically similar nature of Kotlin and Swift it was very easy to code in one and then convert into the other. This meant that the algorithm development taking place could be done on one system and then converted. The solutions on both systems could then be compared with the expected outcomes and with each other.

8.2.2 Location Algorithm

The location algorithm built upon the existing implementation by Dogfish. It used the plots.json file to load an array of paired points (latitude and longitude with the corresponding x and y), which could then used to create a grid of triangles across both the piste map and the geographic map. Dogfish's original method, explained in detail in section 6.3.2, relied upon the fact that the triangles in both the piste map and the geographic map were similar. Unfortunately this is not the case as the triangles may include skew, meaning that their similarity is not maintained. Researching into the problem revealed that a possible solution could exist by using an affine transformation. As all triangles are considered affine, that means it is possible to find a transformation that converts one triangle into another. This transformation can then be used to convert a point inside one triangle into a point inside the other. The transformation is found by solving a system of equations and is significantly less complex than the original solution proposed by Dogfish.

Locating a user outside of the defined points in the plots.json file is difficult; although there may exist a GPS signal, it is not able to convert the location on the geographic map into a corresponding x and y on the piste map accurately. The reasons for this were explained in detail in section 6.3.4, but it is important to note that this is something that cannot currently be reliably done. Adding additional points to the plots.json file will mitigate this problem to a degree. However, the places where a user may be outside of the existing points is usually where it is difficult to pick and define points on the piste map due to the way that it has been constructed. Even though this is an edge case it should be addressed and a better solution discovered.

The location algorithm also relies on an external uncontrollable factor: a GPS signal. If the GPS signal becomes unreliable, the user's position on the map becomes unreliable too. Unfortunately there is currently no way to mitigate an unreliable GPS signal, however, the applications currently shows how accurate the GPS signal is to the nearest metre and this will notify users if the GPS signal worsens or improves.

8.2.3 Routing Algorithm

The routing algorithm uses Dijkstra's algorithm to find the solution to the single source shortest path problem. The nodes and routes used by the algorithm are contained in a nodes.json file. The algorithm finds the shortest path from the closest node to a user's current location (this node is located at the bottom of the nearest piste, this was chosen because users can only go down pistes), and then finds the closest node to the user's chosen destination. Once the path has been calculated, a route from the user's location to their closest node is added to the path.

During the testing it was discovered that the original implementation would only work on maps which have routes going in one direction and it did not lend itself well to working in other map situations, such as a university campus. To rectify this problem several changes were implemented. Firstly the nodes.json file was updated so that the routes connecting the points were reversed. This was to remove the problems that occurred when having different points on routes going in opposite directions (see 6.5.1). Next an algorithm that calculates the route from the user's location to the bottom of the nearest piste was updated so that it compares the different possible solutions and returns the one that is the shortest.

Although the location algorithm takes a user from their current location to the closest node to their destination, ultimately it does not take them to the exact location they press on. This could be implemented by following a similar process that takes the user to the bottom of their closest piste. However, it was thought best to leave that functionality off of this version due to the fact that users of piste maps would be looking to go to specific points like the start of pistes, lifts or restaurants.

When deciding on the type of algorithm to be developed, it was considered that a lookup table of the shortest paths could be created, but this idea was not pursued. The reason for this is two fold: firstly the graphs are relatively small (less than 1000 nodes) and adding routes and nodes to the graph does not require a lookup table to be updated. Secondly, having a lookup table of shortest paths may produce quick results, but it is static. It is possible with dynamic route calculation to adjust various parameters when the route is calculated. This means that it could be possible to change the weight of different types of runs: i.e. make black runs more likely to be selected; or even remove a specific type of run (though that may lead to routes not being found across the piste).

8.2.4 Brute Force Approach

When finding the closest node in both the location and routing algorithm a brute force approach was chosen. This was done because the number of nodes involved is quite small and

iterating across them does not take a large amount of time. By using a brute force approach, we reduce the complexity of the algorithms. However, the algorithms are designed in such a way the brute force approach could easily be updated to use a more complicated but faster approach such as an r-tree or kd-tree.

8.2.5 GPS Functionality

In Android the GPS functionality is not as seamless as it should be. There is a small issue that occurs, where the listener that tracks whether the GPS system on the device is switched on or off, does not register if the application is loaded with the GPS switched off. This issue only presents itself on some devices and not in the Android emulator. There is a work around; by restarting the application once the GPS functionality has been switched back on the listener registers correctly and notifies the user when the GPS is switched on or off. This is not an ideal solution to the problem, and it needs to be further investigated.

8.3 Future Work

8.3.1 Location

As outlined in section 8.2.2 the location algorithm does not perform well when the user is located outside of the points that are defined in the plots.json file. There is no clear solution to this problem: adding more points to the plots.json will help to reduce the discrepancy between the user's actual location and the one shown on the piste map. However, it should be noted that this is an edge case, as it is unlikely for users to be in a position where they are outside of the points.

8.3.2 Routing

Currently the routing algorithm chooses the route that has the "safest" path meaning that it will default to blue or green pistes over red or black. Users may not wish this and would prefer to have more black runs in their route. This could be added to the algorithm by dynamically altering the weights of the edges in the graphs at search time. It would also be possible to eliminate specific types of pistes so that a user would never be presented with a route that contained those types.

Depending on the size of the underlying graph there may be a need to bring in a level of pre-processing. Currently the largest graph that it has been tested on has 300 nodes and fewer than 700 edges between them. This is a sparse graph meaning that any search of it will be fairly quick. If the graphs were to become bigger then incorporating a pre-processed element may help with lookup times. However, any pre-processing should be done on the device if user preferences have been implemented, otherwise the pre-processing could be done beforehand.

8.3.3 Brute Force

Currently the brute force approach to finding the closest node in both the location and routing algorithms is so sufficiently quick, that it does not warrant immediate concern. However, if the underlying graphs become significantly large then an alternative approach such as an r-tree or kd-tree should be implemented to take advantage of the speed benefits associated with those types of trees.

8.3.4 GPS Functionality

Fixing the GPS functionality on Android is a primary concern. Many users may launch an application in airplane mode, which would have the GPS functionality disabled by default. For it to work on some devices and not others may mean that due to fragmentation in the Android device market, not all devices can be supported. Further investigation into this issue is required.

8.3.5 Tracking

During the testing phase of the showcase applications it was found that to test the algorithms a tracking component had to be created. This tracking component had full access to the data from the device's GPS system and it would allow for the user's daily activities to be tracked. Meaning that they could see an image of the piste map with all of the routes and runs that they have taken that day. It could be further extended by the inclusion of a leader board, where users could upload their data and the fastest skier or greatest distance travelled in a day could be compared.

References

[1] Making of: interactive, mobile piste map by LAAX, https://magazin.unic.com/en/2012/02/16/making-of-interactive-mobile-piste-map-by-laax/, February 2012

[2] GPS on ski map by Maprika, https://itunes.apple.com/gb/app/gps-on-ski-map-by-maprika/id404686699?mt=8, March 2017

[3] Dragons' Den stars lose £230,000 in 'Satski' satnav fraud, http://www.techworld.com/news/apps-wearables/dragons-den-stars-lose-230000-in-satski-satnav-fraud-3338472/, February 2012

[4] FATMAP Homepage, https://fatmap.com/, May 2017

[5] National Trust Deploys Laser Survey Aircraft from Bluesky to Map Shropshire Estate in 3D, https://www.bluesky-world.com/single-post/2017/04/12/National-Trust-Deploys-Laser-Survey-Aircraft-from-Bluesky-to-Map-Shropshire-Estate-in-3D, April 2017

[6] Since You Asked, Here's How Google Maps Really Works, https://www.forbes.com/sites/haroldstark/2017/04/26/since-you-asked-heres-how-google-maps-really-works/ , April 2017

[7] Bing Maps new routing engine, https://blogs.bing.com/maps/2012/01/05/bing-maps-new-routing-engine, April 2012

[8] Delling, D. Goldberg, A.V. Pajor, T. and Werneck R.F. Customizable Route Planning in Road Networks, *J. Transportation Science*, 51(2):566-591, May 2015

[9] Sedgewick, R. and Wayne, K *Algorithms*, Fourth Edition. Addison-Wesley Professional, 2011

[10] Byer, O. Lazebnik, F. and Smeltzer, D.L. *Methods for Euclidean Geometry*, First Edition. The Mathematical Association of America, 2010

[11] Ryan, P. J. *Euclidean and non-Euclidean Geometry*, Cambridge University Press, 1986.

[12] Dijkstra, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. □

[13] Gutman, R. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In Workshop on Algorithm Engineering and Experiments (ALENEX), pages 100–111, 2004.

[14] Hart, P. E. Nilsson, N. J. and Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on System Science and Cybernetics*, 4(2):100–107, 1968.

[15] Williams, J. W. J. Heapsort. *Communications of the ACM*, 7:347–348, June 1964.

[16] Fredman, M. L. and Tarjan, R. E. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, July 1987.

[17] Bellman, R. On a routing problem. *Quart. Appl. Math.* 16**:** 87–90 1958.

[18] Ford Jr., L.R. Network Flow Theory Paper P-923. Santa Monica, California: RAND Corporation. August 1956

[19] Moore, E.J. The shortest path through a maze. *Proc. Internat. Sympos. Switching Theory* 1957, Part II. Cambridge, Mass.: Harvard Univ. Press. pages. 285–292, 1959

[20] Floyd, R.W. Algorithm 97: Shortest Path. *Communications of the ACM*. 5 (6): 345, June 1962.

[21] Warshall, S. "A theorem on Boolean matrices". Journal of the ACM. 9 (1): 11–12 (January 1962)

[22] Johnson, D.B. Efficient algorithms for shortest paths in sparse networks, *Journal of the ACM,* 24 (1): 1–13, 1977,

[23] Cormen, T.H.; Leiserson, C.E.; Rivest, R.L. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001

[24] ReactNative, Homepage, https://facebook.github.io/react-native/, June 2017

[25] Appcelerator Homepage, https://www.appcelerator.com/mobile-app-development-products/, June 2017

[26] PhoneGap, HomePage, https://phonegap.com/, June 2017

[27] Gartner Press Release, http://www.gartner.com/newsroom/id/3609817, February 2017

[28] StackOverFlow, Questions tagged iOS, https://stackoverflow.com/questions/tagged/ios, June 2017

[29] StackOverFlow, Questions tagged Android, https://stackoverflow.com/questions/tagged/android, June 2017

[30] Reddit, iOS forum, https://www.reddit.com/r/ios/, June 2017

[31] Udacity, Homepage, https://www.udacity.com/, June 2017

[32] Kotlin, FAQ, https://kotlinlang.org/docs/reference/faq.html, June 2017

[33] Swift, Homepage, https://developer.apple.com/swift/, June 2017

[34] Swift, Homepage, https://swift.org/, June 2017

[35] Swift is like Kotlin, http://nilhcem.com/swift-is-like-kotlin/, June 2017

[36] Scrum Guides, Homepage, http://www.scrumguides.org/, July 2017

[37] Agile Alliance, Homepage, https://www.agilealliance.org/, July 2017

[38] Sims, C. and Johnson, H.L. *Scrum: a Breathtakingly Brief and Agile Introduction*, First Edition. Dymaxicon, April 2012

[39] Sommerville, I. *Software Engineering*, Ninth Edition. Pearson, March 2010

[40] JIRA Software, Homepage, https://www.atlassian.com/software/jira, June 2017

[41] Klaxon, Github page, https://github.com/cbeust/klaxon, June 2017

[42] SwiftyJson, Github page, https://github.com/SwiftyJSON/SwiftyJSON, June 2017

[43] SwiftPriorityQueue, Github page, https://github.com/davecom/SwiftPriorityQueue, June 2017

[44] TileView, Github page, https://github.com/moagrius/TileView, June 2017

[45] Carthage, Github page, https://github.com/Carthage/Carthage, June 2017

[46] Cocoapods, Homepage, https://cocoapods.org/, June 2017

[47] Gradle, Homepage, https://gradle.org, June 2017

[48] Choosing JSON for building APIs, TechTarget, http://searchmicroservices.techtarget.com/feature/Choose-JSON-for-building-APIs, July 2017

[49] Toast-Swift, Github page, https://github.com/scalessec/Toast-Swift, July 2017

[50] Axway Appcelerator Alloy Documentation, http://docs.appcelerator.com/platform/latest/#!/guide/Alloy_Concepts July 2017

[51] Google Play Services, Homepage, https://developers.google.com/android/guides/overview , June 2017

[52] Kotlin Blog, Kotlin on Android: now official, https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/ , June 2017

[53] iOS Adoption, Apteligent https://data.apteligent.com/ios/, July, 2017

[54] Android Adoption, Apteligent https://data.apteligent.com/android/, July 2017

[55] MVC, Apple Documentation, https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html, July 2017

[56] MVVM, Microsoft Documentation, https://msdn.microsoft.com/en-gb/library/hh848246.aspx, July 2017

[57] Axway Appecelerator Titanium Documentation, http://docs.appcelerator.com/platform/latest/#!/guide/Titanium_Platform_Overview, July 2017

Appendix 1 – Statistical Results

Wilcoxon Signed Rank CI: difference of x coordinates

Method

η: median of difference of x coordinates

Descriptive Statistics

Sample	N	Median	CI for η	Achieved Confidence
difference of x coordinates	500	0.646733	(-0.042941, 1.47718)	95.00%

Wilcoxon Signed Rank Test: difference of x coordinates

Method

η: median of difference of x coordinates

Descriptive Statistics

Sample	N	Median
difference of x coordinates	500	0.646733

Test

Null hypothesis	$H_0 : \eta = 0$
Alternative hypothesis	$H_1 : \eta \neq 0$

Sample	N for Test	Wilcoxon Statistic	P-Value
difference of x coordinates	500	68485.00	0.070

Wilcoxon Signed Rank CI: difference of y coordinates

Method

η: median of difference of y coordinates

Descriptive Statistics

Sample	N	Median	CI for η	Achieved Confidence
difference of y coordinates	500	0.0537117	(-0.262412, 0.403257)	95.00%

Wilcoxon Signed Rank Test: difference of y coordinates

Method

η: median of difference of y coordinates

Descriptive Statistics

Sample	N	Median
difference of y coordinates	500	0.0537117

Test

Null hypothesis	$H_0 : \eta = 0$
Alternative hypothesis	$H_1 : \eta \neq 0$

Sample	N for Test	Wilcoxon Statistic	P-Value
difference of y coordinates	500	63665.00	0.748

Wilcoxon Signed Rank CI: distance between points

Method

η: median of distance between points

Descriptive Statistics

Sample	N	Median	CI for η	Achieved Confidence
distance between points	500	7.97937	(7.14172, 8.94656)	95.00%

Wilcoxon Signed Rank Test: distance between points

Method

η: median of distance between points

Descriptive Statistics

Sample	N	Median
distance between points	500	7.97937

Test

Null hypothesis	$H_0 : \eta = 0$
Alternative hypothesis	$H_1 : \eta \neq 0$

Sample	N for Test	Wilcoxon Statistic	P-Value
distance between points	500	125250.0	0.000